SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2019, Vol. 62 192103:1–192103:22
https://doi.org/10.1007/s11432-018-9821-9

Generative API usage code recommendation with parameter concretization

Chi CHEN[1,2,3], Xin PENG[1,2,3*], Jun SUN[4], Zhenchang XING[5], Xin WANG[1,2,3], Yifan ZHAO[1,2,3], Hairui ZHANG[1,2,3] & Wenyun ZHAO[1,2,3]

[1]*School of Computer Science, Fudan University, Shanghai 201203, China;*
[2]*Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China;*
[3]*Shanghai Institute of Intelligent Electronics & Systems, Shanghai 200433, China;*
[4]*Pillar of Information System Technology and Design, Singapore University of Technology and Design, Singapore 487372, Singapore;*
[5]*Research School of Computer Science, Australian National University, Acton ACT 2601, Australia*

Received 10 July 2018/Revised 20 October 2018/Accepted 27 February 2019/Published online 30 July 2019

Abstract Many programming languages and development frameworks have extensive libraries (e.g., JDK and Android libraries) that ease the task of software engineering if used effectively. With numerous library classes and sometimes intricate API (application programming interface) usage constraints, programmers often have difficulty remembering the library APIs and/or using them correctly. This study addresses this problem by developing an engine called DeepAPIRec, which automatically recommends the API usage code. Compared to the existing proposals, our approach distinguishes itself in two ways. First, it is based on a tree-based long short-term memory (LSTM) neural network inspired by recent developments in the machine-learning community. A tree-based LSTM neural network allows us to model and reason about variable-length, preceding and succeeding code contexts, and to make precise predictions. Second, we apply data-flow analysis to generate concrete parameters for the API usage code, which not only allows us to generate complete code recommendations but also improves the accuracy of the learning results according to the tree-based LSTM neural network. Our approach has been implemented for supporting Java programs. Our experimental studies on the JDK library show that at statement-level recommendations, DeepAPIRec can achieve a top-1 accuracy of about 37% and a top-5 accuracy of about 64%, which are significantly better than the existing approaches. Our user study further confirms that DeepAPIRec can help developers to complete a segment of code faster and more accurately as compared to IntelliJ IDEA.

Keywords code recommendation, API, deep learning, data dependency, parameter concretization

Citation Chen C, Peng X, Sun J, et al. Generative API usage code recommendation with parameter concretization. Sci China Inf Sci, 2019, 62(9): 192103, https://doi.org/10.1007/s11432-018-9821-9

1 Introduction

In software development tasks, developers often encounter unfamiliar features, for which they lack the necessary knowledge to handle. Many code recommendation techniques have been developed to help developers in such situations. Some techniques [1–3] take natural language descriptions of the developers' needs as input and use information retrieval techniques or deep learning techniques to search for a code relevant to the input descriptions. Many other techniques [4–6] further utilize code information such as API usage, program structure (e.g., data or control flow), and code changes in the code that developers partially implement for recommending code to complete the partial implementation. In this study, we focus on the code recommendation task.

* Corresponding author (email: pengxin@fudan.edu.cn)

Recent years have witnessed the growth of generative code recommendation approaches based on a key observation of code naturalness [7]. Generative approaches learn a certain statistical language model of coding patterns from the existing code. The model can be as simple as an N-gram-based model [8] or a model that incorporates graph-based code structure [4] or change-history information [6]. An N-gram-based model treats the source code as token-based sequences and takes previous $N-1$ tokens as input to produce a list of candidates for the next token as output. A graph-based model treats the source code as graphs and takes a set of context (sub)graphs of the current code as input to produce a list of candidate APIs as output. A change-based model treats code changes as a bag of fine-grained atomic changes and source code as a set of code tokens. Such a model takes code changes and code context as input to produce a list of candidate APIs as output. Theoretically, these generative approaches can generate code that is previously unseen in the existing code.

However, the abovementioned approaches have only limited modeling and generation capabilities. N-gram- and change-based models only consider the sequential context of code in tokens preceding the current editing location. However, program structures such as data or control flows are also important context information. Moreover, developers often follow a non-sequential order of editing, which suggests the expansion of the code context to include the code after the requested location [4]. The graph-based model considers program structures but treats all contexts equally and cannot effectively model the variable-length code context.

These limitations on modeling and generation make the adoption of existing generative code recommendation approaches difficult. For example, current techniques cannot generate a complete line of code, including an API call with concrete parameters or the assignment of the return value of the API call to a local variable. As another example, current techniques can generate an if token, but cannot generate an if(){} statement skeleton and then if-condition and if-body.

This paper presents a novel approach, called DeepAPIRec for generative API usage code recommendation, which tackles the limitations of existing generative code recommendation methods. Our approach is based on a tree-based long short-term memory (LSTM) neural network that organizes the rich code information in a tree-based structure, which is invertible back to the source code. Our tree-based structure can be incrementally expanded as some code elements are generated to enable the progressive generation of a long, complete line of code. Furthermore, the tree-based structure, together with LSTM's superior ability to preserve long-distance dependencies, allows our approach to model and reason about variable-length, preceding and succeeding code contexts. Besides the tree-based LSTM model, our approach uses a statistical parameter model of data dependency to determine the concrete parameters of the API calls recommended by the tree-based LSTM model.

We conduct a series of experimental studies targeting Java APIs to evaluate the accuracy and effectiveness of DeepAPIRec in recommending API usage code. The results show that DeepAPIRec achieves a top-1 accuracy of about 37% and a top-5 accuracy of about 64%, which are significantly better than the existing approaches, at statement-level recommendations. Furthermore, our sensitivity analysis shows that the proportion of the number of API calls in the context and those to be predicted have significant influence on the prediction accuracy of DeepAPIRec. However, the position of the missing code has little effect on the effectiveness of DeepAPIRec. Finally, we conduct a user study with 16 students and 5 tasks. The results show that DeepAPIRec can help developers complete the programming tasks more accurately and efficiently.

This study makes the following main contributions.

- Proposal of an API usage code recommendation approach that combines the advantages of tree-based LSTM and statistical parameter models for continually generating a complete line of code.
- Implementation of our approach in a deep learning system with GPU acceleration that supports the efficient training and inference of tree-based LSTM networks based on TensorFlow1) and Fold library2).
- Experimental evaluation of the accuracy of DeepAPIRec for statement prediction, analysis of key

1) https://github.com/tensorflow/tensorflow/.
2) https://github.com/tensorflow/fold/.

```
1: public byte[] sign(String message, String digestAlgorithm, PrivateKey pk)
            throws GeneralSecurityException {
2:    byte[] messageByte = message.getBytes();
3:    String signMode = null;
4:    if(pk == null){
5:        pk = getPrivateKey("RSA");
6:        String encryptionAlgorithm = pk.getAlgorithm();
7:        signMode = combine(digestAlgorithm, encryptionAlgorithm);
8:    }else{
9:        String encryptionAlgorithm = pk.getAlgorithm();
10:       signMode = combine(digestAlgorithm, encryptionAlgorithm);
11:   }
12:   Signature sig = Signature.getInstance(signMode);
13:   sig.initSign(pk);
14:   sig.update(messageByte);
15:   return sig.sign();
16:}
```

Figure 1 Example of API usage code recommendation.

factors that influence the accuracy of statement prediction, and measurement of the usability of DeepAPIRec.

The remainder of the paper is structured as follows. Section 2 presents the motivation of our study with an example. Section 3 describes the deep learning model used in our approach. Section 4 presents the proposed approach. Section 5 discusses the evaluation of the proposed approach. Section 6 discusses related work. Section 7 presents the conclusion and recommendations for future studies.

2 Motivation

Figure 1 shows an example of Java code that combines a digest algorithm (which creates a hash value from a message), such as MD5, with an encryption algorithm (which encrypts the hash value using a private key), such as RSA, to sign a message. We assume that the developer, Bob, knows how to sign a message by converting the message into a byte array, specifying the signature algorithm, and signing the message. Using his knowledge, Bob writes a partial implementation, such as the code, in black font (lines 1–11) in Figure 1. This method takes a message to be signed, the name of a digest algorithm digestAlgorith, and a private key pk as input. Line 2 coverts the string message into a byte array. Lines 3–11 specify the signature algorithm according to the given digest algorithm digestAlgorithm and private key pk.

After these preparatory steps, it is time to sign the message. Unfortunately, Bob does not know how to sign the message, i.e., he does not know which API(s) he should use to complete this program. Our DeepAPIRec can recommend appropriate APIs with the proper parameter concretization for Bob to use. To use DeepAPIRec, Bob simply marks line 12 as a hole \$hole\$, and then requests the DeepAPIRec's recommendation.

DeepAPIRec consists of a deep learning model for statement prediction (called the statement model) and a statistical model for parameter concretization (called the parameter model). The statement model uses tree-based LSTM to learn API usage and program structure patterns from the AST-based program representations of code in a large code base. Based on the API usage and program structure in the partial implementation and the API usage patterns that it learns, the statement model infers that the code to be filled in the \$hole\$ is most likely a call to the API java.security.Signature.getInstance(java .lang.String) in order to get the Signature object. In fact, this is exactly what Bob needs to do in line 12.

To make the API call recommendation more complete, DeepAPIRec further uses its parameter model to find the most appropriate parameter(s) that Bob can use to make the API call. The parameter model analyzes the data dependency to compute the dependency probability of each variable or object that occurs before $hole$ for the current recommendation. In this example, it is inferred that java.security.Signature.getInstance(java.lang.String) should be called with the variable signMode. Finally, DeepAPIRec generates a complete line of the code statement for the $hole$ with not only the API call and the appropriate parameter but also the variable declaration and assignment: Signature signature = Signature.getInstance(signMode).

Of course, this line of code does not yet complete the program. Bob can continue to mark the next line as $hole$ and request DeepAPIRec's recommendation again. DeepAPIRec can continually recommend code signature.initSign(pk) for line 13, signature.update(messageByte) for line 14, and byte[] byte_array = signature.sign() for line 15. The correct answer for lines 12–15 in the original source code is shown in Figure 1. We can see that these lines, as recommended by DeepAPIRec, successfully help Bob to complete the program.

Note that DeepAPIRec attempts to recommend a complete line of code statement with the appropriate syntactic structure and constructs. In fact, as the statement model learns program structure patterns from AST-based program representations, it can also recommend program constructs such as an if(){} statement skeleton. This is a significant improvement in code recommendation techniques as compared to the existing techniques, which recommend only next tokens or next API calls.

3 Deep learning models

Our study was enabled by recently developed techniques in deep learning [9–12], such as the tree-structured long short-term memory (Tree-LSTM) networks, in particular, developed in 2015 [11]. Tree-LSTM networks improve the ordinary LSTM networks by supporting tree-based semantic representations. It has been shown that Tree-LSTM outperforms the existing learning models, including LSTM, when the data have a natural tree structure [11]. We state that programs naturally have a tree structure, i.e., the syntax tree, which makes Tree-LSTM an ideal choice for encoding and learning API usage programs. Here, we provide a high-level overview of the relevant deep learning techniques and models.

LSTM is a kind of recurrent neural network (RNN) that is well-suited to classifying, processing, and predicting time series with long-term dependencies among data. When LSTM is used for text processing, all elements (called words) in the input sequence and the output need to be included in a vocabulary. To be processed by LSTM, all words need to be encoded into vectors by word embedding, which maps words to vectors of real numbers. A more detailed description of LSTM can be found at web3).

LSTM has been successfully applied to various sequence learning tasks such as speech recognition, music composition, handwriting recognition, and text generation. However, the source code in a program has a well-defined syntax and many levels of semantics that have tree-based structures in nature [4]. These tree-based syntax and semantics cannot be fully captured by the linear chain structures in sequential LSTM. There has been evidence [11] that tree-structured models, such as Tree-LSTM, outperform sequential LSTM on some text analysis tasks that require understanding syntactic properties of sentences (e.g., semantic relatedness prediction and sentiment classification).

Tree-LSTM is a generalization of LSTM to tree-structured network topologies [11]. The state of a Tree-LSTM unit is composed from an input vector and the states of all its child units. Tree-LSTM has two variants, i.e., Child-Sum Tree-LSTM and N-ary Tree-LSTM. Figure 2 shows a Child-Sum Tree-LSTM network unrolled by the tree nodes. The network contains a series of Tree-LSTM units, each of which is a repeating module of a neural network. For each tree node j in an input tree, the Tree-LSTM unit reads its value and maps it to a vector represented by $x_{t,j}$. Then, the input $x_{t,j}$, the hidden states (i.e., $h_{t-1,j}, \ldots, h_{t-1,k}$), and memory cells (i.e., $c_{t-1,j}, \ldots, c_{t-1,k}$) from all child units are fed into the current Tree-LSTM unit to generate a new hidden state $h_{t,j}$ and memory cell $c_{t,j}$. Once the hidden state $h_{n,1}$ of the root Tree-LSTM unit is generated, it is used to produce the output through the softmax function or other functions. The internal structure of a Tree-LSTM unit is shown in the dotted box in Figure 2. Unlike an LSTM unit, a Tree-LSTM unit takes multiple hidden states and memory cells as input from

3) http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

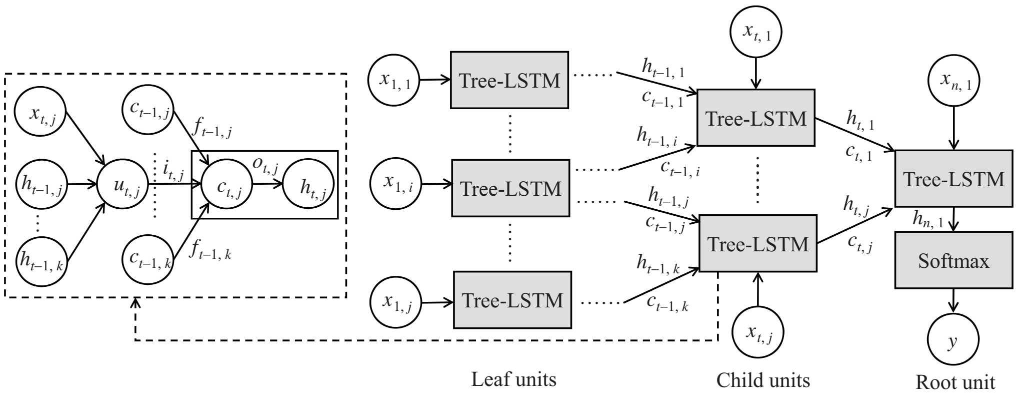

Figure 2 Tree-structured LSTM network (Child-Sum Tree-LSTM network).

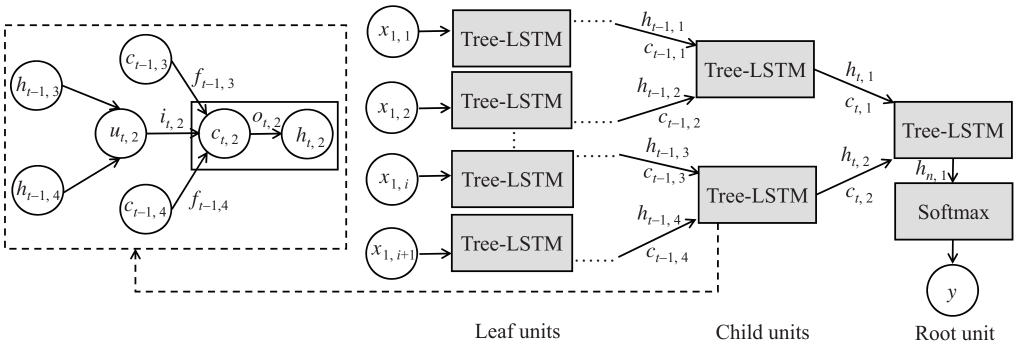

Figure 3 Tree-structured LSTM network (N-ary Tree-LSTM network and $N = 2$).

all its child units. Figure 3 shows an N-ary Tree-LSTM network, which is similar to the Child-Sum Tree-LSTM network, unrolled by the tree nodes. Following are the differences: (1) each parent node has at most N ordered child nodes; (2) only leaf nodes can take inputs.

Child-Sum Tree-LSTM is suited for dependency trees with high branching factor, where each node can take an input and the child nodes are unordered. N-ary Tree-LSTM is suited to constituency trees with limited branching factor, where only the leaf nodes can take inputs and the child nodes are ordered [11]. The code trees in our approach reflect the structures of programs, thus may have high branching factor considering the branches, and the non-leaf nodes can represent code elements (e.g., API calls and control units). On the other hand, the child nodes are ordered, for example, to represent the condition expression, the true branch, the false branch of an if statement, and the rest part of the program. Therefore, we combine Child-Sum Tree-LSTM with N-ary Tree-LSTM as the learning model of programs.

4 Approach

In this section, we present details of our approach, which is referred to as DeepAPIRec. DeepAPIRec is a data-driven approach for API usage code recommendation. The ultimate goal of DeepAPIRec is to generate and recommend a complete API usage code according to the context in order to reduce the programmers' effort. Our aim is not to simply generate relevant API call sequences and let the programmer work out how to structure these calls and identify the right parameters, but rather to generate a series of statements that can be variable declarations, assignments, structural statements, such as if and while, and API calls with concrete parameters.

To use DeepAPIRec for help, a developer needs to provide some code, which includes some API calls or variable declarations of API classes, before or after the requested location as context. When requested, DeepAPIRec recommends a statement of different types, including variable declarations, assignments,

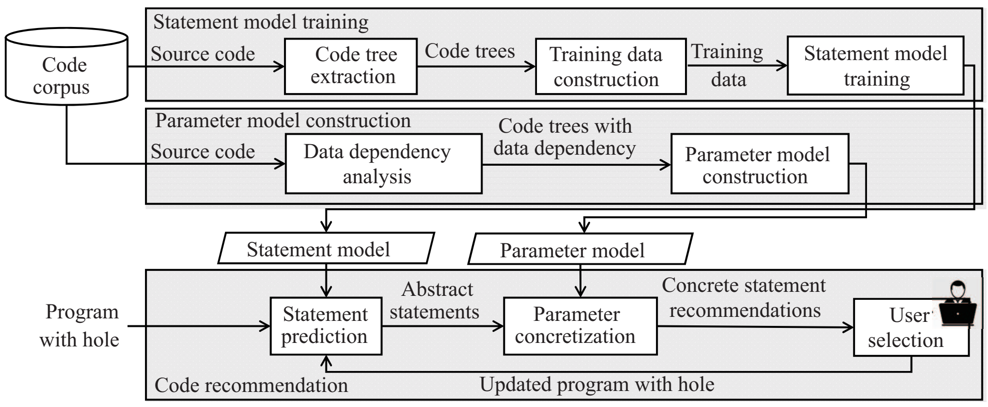

Figure 4 Overall workflow of DeepAPIRec.

object creations, API method calls or field accesses, and control units. DeepAPIRec recommends simple statements that do not include sub-statements. For example, instead of recommending FileInputStream fileInputStream = new FileInputStream(new File(path)), DeepAPIRec recommends two simple statements File file = new File(path) and FileInputStream fileInputStream = new FileInputStream(file) to implement the same functionality. When recommending control structures, DeepAPIRec first recommends the control unit, then its condition, and finally its body. For example, for the control structure if(file.exists()){file.delete();}, DeepAPIRec first recommends if(){}, then file.exists(), and finally, file.delete().

The overall workflow of DeepAPIRec is shown in Figure 4, which includes three phases, i.e., statement model training, parameter model construction, and code recommendation. The statement model training phase is designed to predict the next statement. It preprocesses the source code of the code corpus, prepares a set of training data, and then applies Tree-LSTM based deep learning to train a statement model for statement prediction. The parameter model construction phase aims to complete the prediction by generating concrete parameters if an API call is generated in the previous phase. It analyzes the source code from the code corpus, systematically extracts data dependency, and then constructs a statistical parameter model for parameter concretization. Based on these two models, the code recommendation phase takes a program with a hole as input and generates a list of ranked code recommendations with concrete parameters. After the user selects a recommended statement, the code recommendation process continues and recommends further statements based on the updated program.

Note that the statement model is trained by the abstract representations of statements, which neglect concrete variable names. The parameter model is independently constructed from the statement model on the basis of concrete statements. In the code recommendation phase, the statement model is used to predict abstract statements and the parameter model is used to generate concrete parameters for the abstract statements by identifying and evaluating compatible variables or objects from the code context.

4.1 Statement model training

This phase aims to obtain a model that allows us to precisely predict the next statement on the basis of the surrounding programming context. Given a code base containing considerable of code corpus, during this phase, we first extract a code tree from each method in the code base, generate a set of training data from the extracted code trees, and finally, train a model by deep learning.

4.1.1 *Code tree extraction*

Code trees are representations of the code samples in the code base and are the raw data used to train Tree-LSTM networks in a later step. Each node in a code tree represents an API call, a control unit, a variable declaration, or an assignment. Other types of statements, such as assert statements, client

Table 1 Code tree extraction rules

Type	Root	Child nodes and subtrees
if	If	Condition subtree for the condition expression Then subtree for the true branch ElseIf/Else subtree for the false branch Successor subtree for the rest part of the program

while/do
for/foreach
	While/DoWhile
	For/Foreach
		Condition subtree for the condition expression
		Body subtree for the loop body
		Successor subtree for the rest part of the program

switch	Switch	Selector subtree for the selector statement a series of Case subtrees, each for a case branch Default subtree for the default statement Successor subtree for the rest part of the program
try/catch	Try/Catch	Try subtree for the try block Catch subtree for the catch block Finally subtree for the finally block Successor subtree for the rest part of the program

Table 2 Representations of statements in a code tree

Statement type	Representation	Example
Var. Declaration	[Full Class Name] Variable	`String s;` → java.lang.String Variable
Var. Declaration with Constant Assignment	[Full Class Name] = Constant	`String s = "xy";` → java.lang.String = Constant
Var. Declaration with Null Assignment	[Full Class Name] = Null	`String str = null;` → java.lang.String = Null
Var. Declaration with Object Creation	[Full Class Name].new ([Full Parameter Types])	`File file = new File("path");` → java.io.File.new(java.lang.String)
API Method Call	[Full Method Name] ([Full Parameter Types])	`builder.append("path");` → java.lang.StringBuilder.append(java.lang.String)
API Field Access	[Full Field Name]	`System.out;` → java.lang.System.out

method calls and field accesses, are neglected for now. We extract code trees from the abstract syntax tree (AST) and the control flow of the source code. Given a program, we construct a code tree systematically as follows.

• If the first statement is of a type listed in the first column of Table 1 and includes API calls in its body or condition expression, we create a root node, child nodes, and subtrees according to the rules shown in Table 1. For example, if the first statement is an *if* statement, we create a root node IF and the following child nodes and subtrees: the Condition subtree for the condition expression and a leaf node ConditionEnd to represent the end of the condition expression; the Then subtree for the true branch; a child node ElseIf or Else and its subtree for the false branch together with a leaf node ControlEnd to represent the end of the control unit; and the Successor subtree for the rest part of the program. Note that ControlEnd node will be added as the leaf node of the Then subtree if there is no ElseIf or Else subtree.

• If the first statement is a condition or expression that includes API calls, we create a root node for its first API call and a subtree for the rest part of the program.

• Otherwise, we create a statement node according to Table 2 and create a subtree of the statement node for the rest part of the program.

Note that each statement is represented in an abstract way in the code tree, e.g., we neglect concrete variable names and record only the fully qualified class, method, or field names of API calls. The representations of different kinds of statements are shown in Table 2. An example code tree is shown in Figure 5, which represents the code (mark line 12 as hole) shown in Figure 1.

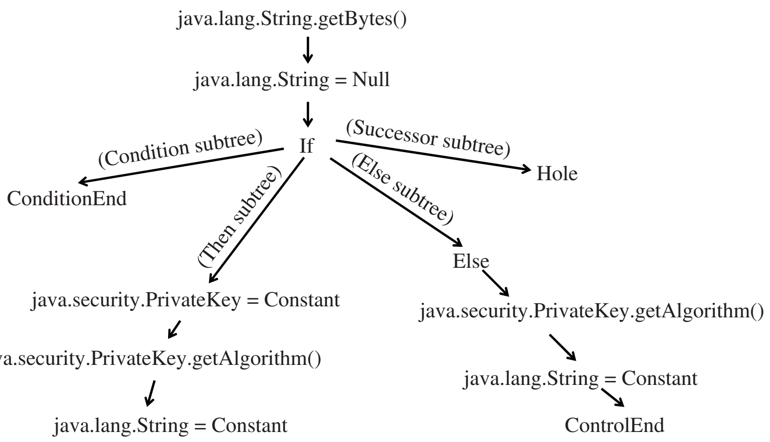
java.lang.String.getBytes()
java.lang.String = Null
If
(Condition subtree)
ConditionEnd
(Successor subtree)
Hole
(Then subtree)
(Else subtree)
Else
java.security.PrivateKey = Constant
java.security.PrivateKey.getAlgorithm()
java.lang.String = Constant
java.security.PrivateKey.getAlgorithm()
java.lang.String = Constant
ControlEnd

Figure 5 Example of code tree.

In terms of implementation, we use a syntax parser to determine whether a method call or field access is an API call and precisely recognize the target API method or field for program element embedding. We use the reflection mechanism provided in Java to obtain a complete list of API methods and fields from the target API libraries (e.g., the JDK library). When parsing the source code, we add the target libraries into the classpaths. Thus, given a method call or field access from the source code, we can obtain the fully qualified name of the target class from the parser and try to match the target method or field with the methods or fields in the API list. Given a method call, we consider not only the method name but also the compatibility of the parameter list. If the target method or field matches, the method call or field access is regarded as an API call; otherwise, it is regarded as a client method call or field access. For cascading method calls or field accesses, such as java.lang.System.out.println (java.lang.String), we first recognize its first part (e.g., java.lang.System.out) and determine whether it is an API call. If it is an API call, we obatain its return type (e.g., java.io.PrintStream) using the reflection mechanism, and then use the return type as a target class to recognize the remainder of the method call or field access (e.g., java.io.PrintStream.println (java.lang.String)).

4.1.2 *Training data construction*

With the extracted code trees, we then construct training samples. Each training sample is a pair consisting of an incomplete code tree (i.e., the context) and a code tree node to be generated (i.e., target node). For each tree obtained from the code base, we remove a part of its nodes (i.e., the hole) and replace it with a special node Hole at the position. The resultant tree and the root of the removed nodes form a training sample. Note that, although we generate only one node at a time in our approach, the hole can include multiple statements (and thus, multiple nodes).

Given a code tree tree, a node node in the tree, and a constant HS representing the number of nodes in the hole, we produce a single training sample according to Algorithm 1. If the current node is a control node (i.e., IF, While, DoWhile, For, Foreach, Switch, and Try/Catch), all of the node's subtrees, except for the Successor subtree, are removed. Otherwise, only the current node is removed. Then, the root node of the remaining subtree is considered as the current node and the above procedure is performed iteratively until the number of removed nodes reaches or exceeds HS or no more nodes are left. Finally, node is replaced with a Hole node and the resultant tree is returned, and together with node, forms a training sample.

To construct a set of training samples, we perform the above algorithm for each node in a code tree with different hole sizes ranging from 1 to MaxHS-1 (where MaxHS is the number of nodes in the code tree). To enable the statement model to predict if a program is complete (after filling the hole with some statements), we introduce a training sample (tree, End), where End is a special node denoting that the tree is complete for every code tree tree.

Algorithm 1 Training sample generation

```
Input: tree, node, HS;
Output: tree;
 1: count=0, curr=node;
 2: while count is less than HS and curr is not Null do
 3:     Let old be curr;
 4:     if curr is If, While, DoWhile, For, Foreach, Switch, or Try/Catch then
 5:         for each child of curr do
 6:             if child is the node of Successor subtree then
 7:                 count=count+1;
 8:                 curr=child;
 9:             else
10:                 Remove child from tree;
11:             end if
12:         end for
13:     else
14:         Set curr to be child of curr;
15:         count=count+1;
16:     end if
17:     Remove old from tree;
18: end while
19: Replace node in tree with Hole.
```

4.1.3 *Statement model training*

Next, we aim to apply deep learning techniques to construct a statement model based on the training samples constructed in the last step. Our statement model combines the advantages of Child-Sum Tree-LSTM and N-ary Tree-LSTM. The model's overall structure is a Child-Sum Tree-LSTM network, whereas its transition equations are adapted from N-ary Tree-LSTM to fit the arbitrary number of children of a node. The vocabulary of the model includes all API methods and fields (including cascading API calls) that occur in the training samples.

To feed a code tree to a Tree-LSTM network, we need a vector representation for each node of the tree. Therefore, we map each word in the vocabulary to an N-dimensional (e.g., 50) vector of real numbers by randomly initializing each of its dimensions with a value in a specific range (e.g., from -0.05 to 0.05). In this way, the code tree (i.e., the code generation context) in each training sample can be transformed into an embedded code tree, i.e., each node of the tree is an embedded word (vector).

To train the statement model, we feed all training samples to the Tree-LSTM network. For each training sample, the embedded code tree is propagated forward through the network from the input layer to the output layer to generate the output. The structure information of the embedded code tree is encoded during the forward propagation process. The Tree-LSTM network is dynamically unrolled according to the embedded code tree in a top-down way to reflect the structure of the code tree. For example, for the if node in the code tree shown in Figure 5, a Tree-LSTM unit is assigned with four child units representing the four child nodes respectively. Once the output is generated, the Tree-LSTM network starts backpropagation to update the parameters. The gradient descent optimization algorithm adopted in this study is the adaptive gradient algorithm (AdaGrad[4]), a modified stochastic gradient descent with per-parameter learning rate [13]. The above process continues until all training samples are processed.

4.2 Parameter model construction

In this phase, we aim to obtain a model that allows us to generate concrete parameters for a predicted API call based on the context. We construct the parameter model by analyzing the data dependency between the recommended API and the preceding program statements. Intuitively, the parameters can be either newly created or previously created objects. In the latter case, a data dependency would be formed. Analyzing data dependency in the code base not only allows us to predict the parameters for the API call but may also help improve statement prediction (i.e., a predicted statement whose parameters cannot be concretized is unlikely to be correct). Indeed, the rationale of the parameter model is a heuristic designed

4) A brief introduction to AdaGrad can be seen at https://en.wikipedia.org/wiki/Stochastic_gradient_descent#AdaGrad/.

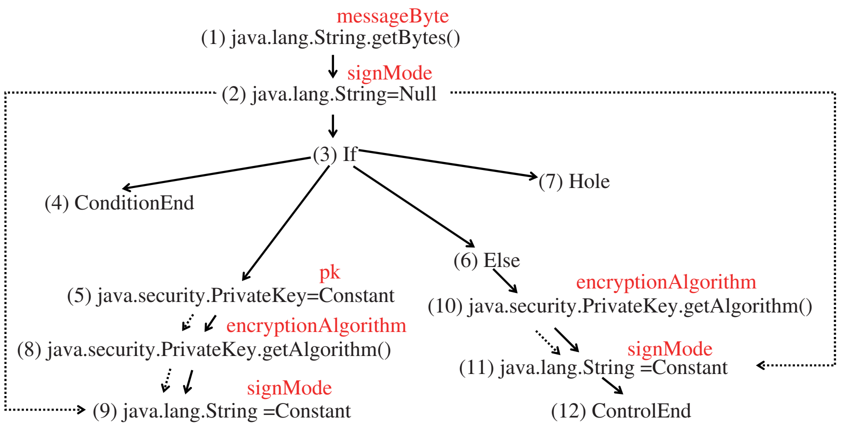
messageByte
(1) java.lang.String.getBytes()
signMode
(2) java.lang.String=Null
(3) If
(4) ConditionEnd
(7) Hole
(6) Else
pk
(5) java.security.PrivateKey=Constant
encryptionAlgorithm
(10) java.security.PrivateKey.getAlgorithm()
encryptionAlgorithm
(8) java.security.PrivateKey.getAlgorithm()
signMode
(11) java.lang.String =Constant
signMode
(9) java.lang.String =Constant
(12) ControlEnd

Figure 6 (Color online) Example: code tree with data dependency.

to maximize the connection between the predicted statement and the remainder of the code snippet. Despite having no evidence that such as heuristic is universally effective, we believe it to be so for the following reasons. First, the main purpose of the parameter model is to concretize parameters, and so, make connections (i.e., data dependency) with the remainder of the code snippet, thereby facilitating our concretizing of the parameters. Second, this heuristic is remotely related to [14], which suggests that the words in many natural languages are strongly connected with its neighbors. Third, the following experimental results suggest the feasibility of our heuristic.

4.2.1 *Data dependency analysis*

To construct the parameter model, we first identify data dependency between nodes in the previously extracted code trees. That is, if there is a direct data dependency between two nodes, we add a (dashed) edge between them. Additionally, if a node represents a statement that is a variable declaration or assignment statement, we label the node with the name of the variable being declared or assigned. Figure 6 shows the code tree with data dependency for the program shown in Figure 1. The variable name of each node (if the node contains a variable name) is marked in red.

4.2.2 *Model construction*

Intuitively, the parameter model is used to predict the data dependency between a statement and its preceding statements, thereby allowing us to predict the parameters of an API call. We build the parameter model by "counting" the data dependencies between two nodes in a path in the code tree as follows. Let Dependency(n, end, p), where p is a path, n is a node in the path, and end is the last node of the path, be the number of times a path p is observed in all code trees such that there is a data dependency from node n to end. Intuitively, p is the context in which the data dependency occurs. For instance, given the path (1)–(2)–(3)–(7) in Figure 6, the dependency count between nodes 2 and 7 is the number of times the data dependency occurs in such a path. Assume that, in the code corpus, the path (1)–(2)–(3)–(7) occurs 100 times, of which 90 times there is a data dependency between nodes 2 and 7, then the dependency count between nodes 2 and 7 is 90.

Furthermore, where p is a path, let subpaths(p) be the set of suffixes of p defined as follows: subpaths(p) $= \{\pi | \exists q.\ q \cdot \pi = p \wedge \text{length}(\pi) > 2\}$, where $\cdot$ is the sequence concatenation operator. For instance, subpaths(p), where p is (1)–(2)–(3)–(7) is a set containing (1)–(2)–(3)–(7) and (2)–(3)–(7). Note that subpaths(p) contains only paths that have a length of more than 2. This is because we would like to count the data dependencies between two nodes in a certain context (i.e., the path). A path that is too short (i.e., with a length of 2 or less) is not a credible context (e.g., the context would be too common).

Given a code tree with a hole and a recommended statement rec, we compute a score for data dependency as follows. We first fill the hole with rec. Afterward, we obtain all paths from the root of the tree to rec, denoted as Paths. Next, we build a set of paths AllPaths, which represent the different contexts a data dependency might occur as follows: AllPaths = {subpaths(p) | $p \in$ Paths}. The total number of data dependencies from any node n in the code tree to rec in any context can be calculated as follows:

$$\text{total(rec)} = \Sigma_n \Sigma_{p\in\text{AllPaths}} \text{Dependency}(n, \text{rec}, p). \tag{1}$$

Then, we compute the probability of having a data dependency from a node n to rec as follows:

$$\text{dp}(n, \text{rec}) = \frac{\Sigma_{p\in\text{AllPaths}} \text{Dependency}(n, \text{rec}, p)}{\text{total(rec)}}, \tag{2}$$

where the numerator is the accumulated dependency count over all contexts (i.e., paths in AllPaths) for a particular node n. Last, we recommend parameters of rec according to the node on which rec is most likely to have a data dependency, i.e., the one with the largest dp(n, rec).

For the example in Figure 1, we assume that the statement java.security.Signature.getInstance(java. lang.String) has been recommended. We first use the statement to replace the Hole node in the code tree shown in Figure 6. To recommend parameters, we compute the dependency probabilities between (1) and (7), (2) and (7), and (3) and (7) by first identifying all contexts, which are paths (1)–(2)–(3)–(7) and (2)–(3)–(7). Then, for each path p, we obtain the dependency counts Dependency$(1, 7, p)$, Dependency$(2, 7, p)$, and Dependency$(3, 7, p)$. The total dependency total(rec) is the sum of the dependency counts of all paths and nodes. If dp$(2, 7)$ is the largest of all data dependency probabilities, then we would recommend signMode as the parameter for the recommended statement.

A detailed algorithm for parameter recommendation is shown in Algorithm 2. Given a program with hole prog and the top N recommendations provided by the statement model, we convert prog to a code tree g and respectively append each recommendation as a node appendNode to g in order to replace the Hole node. We represent each new code tree containing appendNode as g'. To recommend parameters, we compute the dependency probability between each node and appendNode respectively. We obtain Paths (which is the set of paths that reach appendNode from the root node of g') at line 4 and build AllPaths by the loop from lines 5 to 8. We initialize a map map(node,count) to store the dependency counts of appendNode, where the key represents a node n other than appendNode and the value represents the dependency count between n and appendNode. Next, for each path p in AllPaths, we obtain Dependency(n, appendNode, p) between each node n and appendNode, and then, update map accordingly. For each appendNode, we obtain its corresponding map from counts and add all counts in map as total to represent the total dependency count of appendNode. Then, for each node in map for appendNode, we use the result of the count of each node divided by total as that node's dependency probability, which is used to recommend parameters.

Algorithm 2 Parameter recommendation algorithm

```
Input: Program with Hole prog, Recommendations recs;
Output: counts;
 1: pairs(appendNode, $g'$) $\leftarrow$ appendRecommendations(recs, prog);
 2: counts $\leftarrow$ {};
 3: for each pair(appendNode, $g'$) in pairs(appendNode, $g'$) do
 4:     Paths $\leftarrow$ paths that can reach appendNode from root node in $g'$;
 5:     Let AllPaths be an empty set;
 6:     for each path in Paths do
 7:         add getSubPaths(path) in AllPaths;
 8:     end for
 9:     Init map(node,count) to store dependency count and add it in counts;
10:     for each $p$ in AllPaths do
11:         for each node $n$ except appendNode in $p$ do
12:             update($n$, map, Dependency($n$, appendNode, $p$));
13:         end for
14:     end for
15: end for
```

Note that the parameter model can be used to rerank the recommended statements. Intuitively, if a recommended statement (i.e., a node) can be easily connected with the preceding statements through data dependency, it is likely to be a correct recommendation. Let recs be a set of statements recommended by the statement model. For each recommended statement rec, we can obtain one probability from the statement model (for the likelihood of being the right statement), e.g., $\alpha(\text{rec})$. We can calculate a probability $\beta(\text{rec})$ from the parameter model as follows:

$$\beta(\text{rec}) = \frac{\text{total(rec)}}{\Sigma_{\text{rec} \in \text{recs}}}. \tag{3}$$

Intuitively, $\beta(\text{rec})$ measures the recommended statement's connection to the preceding context through data dependency. The higher the connection is, the more likely that the recommended statement is the correct one. We then rank the recommended statements recs according to a weighted sum of these two probabilities: $0.65 \times \alpha(\text{rec}) + 0.35 \times \beta(\text{rec})$, where the coefficients are determined through empirical experiments.

4.3 Code recommendation

The code recommendation phase takes a program with a hole (i.e., a missing segment of the program) as input and recommends a list of statements to fill the hole in an iterative and interactive way. In each iteration, a set of ranked recommendations, to be checked and selected by the user, is generated for the next statement. The code recommendation phase first uses statement prediction to generate abstract statements in which variables and parameters are abstracted into types and classes, and then uses parameter concretization to instantiate abstract statements into concrete ones.

The statement prediction step transforms the input program into a code tree, and then maps it to an embedded code tree by following the same processes in statement model training. The step then feeds the embedded code tree into the statement model and obtains the top N recommended abstract statements together with their probabilities.

The parameter concretization step instantiates each of the recommended abstract statements into a concrete statement. Given an abstract statement, the step first recognizes the types and classes that need to be concretized in it. For each type or class, the step identifies compatible variables or objects from the code context and regards them as the candidates. For each candidate, the step uses the parameter model to compute the dependency probability from the current statement to it. The variable or object that has the highest dependency probability is chosen to concretize the type or class. If there are no candidates for the type or class, the step directly uses the type or class name as a parameter and leaves the initialization to the user. After all N recommended abstract statements are instantiated, the step reranks the recommended statements according to their statement prediction and parameter concretization probabilities.

Finally, the reranked top N concrete statements are returned as recommendations for the user to check and select.

5 Evaluation

To evaluate the accuracy and effectiveness of DeepAPIRec for API usage code recommendation, we conduct a series of experimental studies to answer the following three research questions:

RQ1 (Prediction accuracy): How accurate is DeepAPIRec in predicting the next statement as compared to state-of-the-art approaches?

RQ2 (Sensitivity analysis): How do different factors such as hole position and context proportion affect the accuracy of statement prediction?

RQ3 (Usability analysis): How can statement prediction be used by developers to complete a segment of code?

Table 3 First token prediction accuracy of NC N-gram (%)

Project	Top-1	Top-3	Top-5	Top-10
Galaxy (978)	9.1	21.2	31.8	46.9
Log4j (3703)	5.3	12.9	17.9	27.4
JGit (8718)	2.8	10.0	14.8	22.7
Froyo-Email (2329)	10.4	27.1	34.0	44.3
Grid-Sphere (2860)	4.1	18.4	32.5	43.0
Itext (5716)	5.5	14.9	20.8	31.3

5.1 Implementation and data

The AST analysis in code tree extraction is implemented using JavaParser[5]. The combination of Child-Sum and N-ary Tree-LSTMs is implemented on the basis of TensorFlow (version 1.0.0) and its Fold library (version 0.0.1). The Fold library provides the implementation of dynamic computation graphs [15] for the creation of TensorFlow models that consume structured data and support GPU acceleration.

Our experimental studies focus on JDK 1.8, which has 17173 API classes and 137134 API methods/fields. To prepare the training data, we collect a large corpus of Java projects from GitHub according to the following criteria: the projects have more than 800 stars[6] and have been forked at least once from 2014 to 2017. Our training data are extracted from Java methods. We filter out the methods that are too small (having less than 2 lines of code), too large (larger than 200 KB), or do not include JDK API invocations. We obtain 348478 methods with 3652852 lines of code from 472452 source files of 738 projects for the construction of training samples. For each method, we use Algorithm 1 to generate a series of training samples at all the possible positions of the code and with different hole sizes from 1 to 5. Finally, we generate 7669339 training samples with different sizes of holes from all the methods.

The training of the statement model is conducted on a server with Intel Xeon E5-2620 2.1 GHz (16 threads and 128 GB RAM) and two Nvidia 1080Ti GPUs running on Ubuntu 16.04.2 LTS. When we train the Tree-LSTM network, we set the dimension of the input vector to 50, hidden size to 300, epoch to 50, and learning rate to 0.05, while using dropout with a value of 0.75 to prevent overfitting. In each epoch, we set the batchsize of parallel computation to 256.

We evaluate the accuracies of different approaches on the latest versions (accessed in January 2018) of the following six projects from GitHub: Galaxy, Log4j, JGit, Froyo-Email, Grid-Sphere, and Itext. These projects have been used in previous studies [4, 6, 7], and are not included in our training data. The test data are prepared in a manner similar to the preparation of the training data. The accuracy is measured by the percentage of test cases in which the correct results are included in the Top-N recommendations. In the evaluation, we consider only statements that include APIs in the training data.

5.2 Prediction accuracy (RQ1)

To answer RQ1, we test the statement prediction accuracy of DeepAPIRec on the six projects and compare the results with the state-of-the-art approaches whose implementations are available.

One of the baseline approaches is Nested-Cache N-gram (NC N-gram) in its maintenance setting [16], which is an enhanced statistical language modeling approach for the source code. Note that NC N-gram is a token-based approach that needs to recommend a statement by iteratively predicting each token. Table 3 shows the accuracy of NC N-gram in predicting the first token of a statement, where the number of test samples extracted from each project is given after the project name. Considering that most statements include 5–10 tokens, this accuracy is too low for statement prediction. To provide a fair comparison, we adapt NC N-gram, which can be called by the NC N-gram-Statement, as a statement-based prediction approach. NC N-gram-Statement represents each statement in an abstract way as an abstract representation used in DeepAPIRec (see Table 2) and treats the whole statement as a token.

5) https://github.com/javaparser/javaparser/.

6) Starring is a GitHub feature that lets users bookmark repositories and stars indicate the approximate level of interest.

Table 4 Statement prediction accuracy (%)

Project	Model	Top-1	Top-3	Top-5	Top-10
Galaxy (978)	NC N-gram-Statement	16.7	32.6	41.7	51.8
	LSTM	16.4	19.1	21.7	29.1
	Tree-LSTM	23.6	44.4	52.2	63.0
	DeepAPIRec	25.4	45.3	53.2	63.0
Log4j (3703)	NC N-gram-Statement	25.3	42.4	52.0	58.4
	LSTM	23.2	25.8	27.2	32.5
	Tree-LSTM	39.6	54.7	60.8	72.6
	DeepAPIRec	40.2	56.1	61.7	72.6
JGit (8718)	NC N-gram-Statement	25.2	46.9	56.9	67.4
	LSTM	24.6	30.6	35.0	43.6
	Tree-LSTM	37.7	59.1	67.4	77.7
	DeepAPIRec	38.8	59.2	67.6	77.7
Froyo-Email (2329)	NC N-gram-Statement	32.5	52.1	60.6	70.8
	LSTM	19.3	25.1	30.8	38.3
	Tree-LSTM	33.7	55.6	65.2	75.4
	DeepAPIRec	36.1	57.7	66.8	75.4
Grid-Sphere (2860)	NC N-gram-Statement	25.3	43.3	53.5	60.9
	LSTM	21.2	33.6	37.1	42.0
	Tree-LSTM	34.0	55.2	63.7	71.6
	DeepAPIRec	35.7	54.4	64.9	71.6
Itext (5716)	NC N-gram-Statement	28.3	44.6	54.3	63.1
	LSTM	23.3	27.2	29.5	35.7
	Tree-LSTM	34.6	52.5	60.8	70.8
	DeepAPIRec	36.0	53.3	62.4	70.8

The other baseline approach is an LSTM-based code prediction approach. It is implemented by ourselves according to the LSTM approach described in [17] and the LSTM reference implementation on the TensorFlow website. Similarly, it uses the abstract statement representation as defined in Table 2. For training the LSTM approach, we set the number of hidden layers to 3, the hidden size to 300, and the learning rate to 0.05. We also include a variant of DeepAPIRec without parameter concretization (called Tree-LSTM) for comparison to evaluate the influence of the reranking of the Top-10 recommended statements on the statement prediction results. All prediction models used for comparison are trained with the same training data of DeepAPIRec.

Table 4 shows the results of the accuracy evaluation. The number of test samples extracted from each project is given below the project name. Note that this evaluation does not consider the accuracy of the parameters. Table 4 shows that DeepAPIRec outperforms the other three approaches, with Top-1, Top-3, Top-5, and Top-10 accuracies of 25.4%–40.2%, 45.3%–59.2%, 53.2%–67.6%, and 63.0%–77.7%, respectively. NC N-gram is worse than the other approaches, indicating that token-based prediction performs badly for statement prediction. NC N-gram-Statement is much better than LSTM and the accuracy of DeepAPIRec is more than 10 percentage points higher than that of NC N-gram-Statement in most cases. The accuracy of DeepAPIRec is higher than that of Tree-LSTM in all cases except for the Top-3 accuracy of Grid-Sphere, indicating that the reranking included in the parameter concretization can maintain or even improve the accuracy of statement prediction.

As NC N-gram-Statement and LSTM can consider only the code context preceding the hole, we also test these approaches with test samples that have only the code context preceding the hole. Table 5 shows the results. When considering only the code context preceding the hole, DeepAPIRec still outperforms all the other three approaches.

We further evaluate the accuracy of parameter concretization by DeepAPIRec. For each test case, if the correct statement is included in the Top-10 recommendations, then the parameters that are variables or objects (i.e., not constants) in the statement are considered for accuracy evaluation. The accuracy

Table 5 Statement prediction accuracy when only considering preceding context (%)

Project	Model	Top-1	Top-3	Top-5	Top-10
Galaxy (209)	NC N-gram-Statement	12.4	30.6	41.1	52.2
	LSTM	11.5	15.3	20.6	30.1
	Tree-LSTM	23.0	42.6	51.2	60.3
	DeepAPIRec	23.4	43.1	51.7	60.3
Log4j (829)	NC N-gram-Statement	22.0	39.7	49.7	55.9
	LSTM	21.1	23.2	25.9	32.7
	Tree-LSTM	33.4	47.3	53.3	66.7
	DeepAPIRec	33.3	49.9	54.2	66.7
JGit (2256)	NC N-gram-Statement	24.6	47.1	57.3	67.6
	LSTM	22.2	28.7	32.4	40.3
	Tree-LSTM	32.2	54.3	62.5	73.6
	DeepAPIRec	33.6	54.6	63.3	73.6
Froyo-Email (566)	NC N-gram-Statement	29.9	50.0	59.7	69.8
	LSTM	16.1	22.6	28.1	35.9
	Tree-LSTM	27.6	50.0	58.7	70.0
	DeepAPIRec	30.4	51.8	61.5	70.0
Grid-Sphere (683)	NC N-gram-Statement	21.7	42.6	54.9	63.8
	LSTM	17.9	30.3	35.9	42.2
	Tree-LSTM	32.9	53.9	63.3	69.8
	DeepAPIRec	34.3	52.7	64.6	69.8
Itext (1546)	NC N-gram-Statement	29.0	44.6	54.1	62.8
	LSTM	22.1	26.3	29.8	37.3
	Tree-LSTM	33.4	51.3	59.1	66.6
	DeepAPIRec	34.7	51.4	60.0	66.6

Table 6 Parameter concretization accuracy (%) of DeepAPIRec

Galaxy	Log4j	JGit	Froyo-Email	Grid-Sphere	Itext
68.1	86.3	82.3	68.1	79.1	61.4

of parameter concretization is calculated as the percentage of correctly concretized parameters of all considered parameters. Table 6 shows the accuracy of the parameter concretization of DeepAPIRec in the six projects. The accuracy is 61.4%–86.3% and Itext has the lowest accuracy. After analyzing the data, we find that the low accuracy of Itext is mainly caused by the following two problems. One is that some parameters are declared as the general type java.lang.Object, so identifying matched objects for them is difficult. The other problem is the presence of more than one parameter of the same type in some API invocations, so DeepAPIRec currently cannot differentiate among these parameters.

5.3 Sensitivity analysis (RQ2)

Hole position and context proportion are two key factors for the accuracy of code prediction. To answer RQ2, we conduct experiments to investigate the influence of each of these two factors on the accuracy of the code prediction of DeepAPIRec. For the hole position, we define the line number of the position of the hole as currentLineNumber and the total number of lines of the method as totalLineNumber. Then, the hole position is defined as currentLineNumber/totalLineNumber to be the relative position of the hole in the method. We categorize each test sample into one of the five hole position ranges ((0, 1/5], (1/5, 2/5], (2/5, 3/5], (3/5, 4/5], (4/5, 1]) according to the relative position of the hole. For context proportion, we define the amount of APIs to be predicted as unknown and that already known as known. Then, the context proportion is defined as known/(unknown+known). We categorize each test sample into one of the five context proportion ranges ((0, 1/5], (1/5, 2/5], (2/5, 3/5], (3/5, 4/5], (4/5, 1]).

The results of the sensitivity analysis on the hole position are shown in Figure 7. There is no obvious

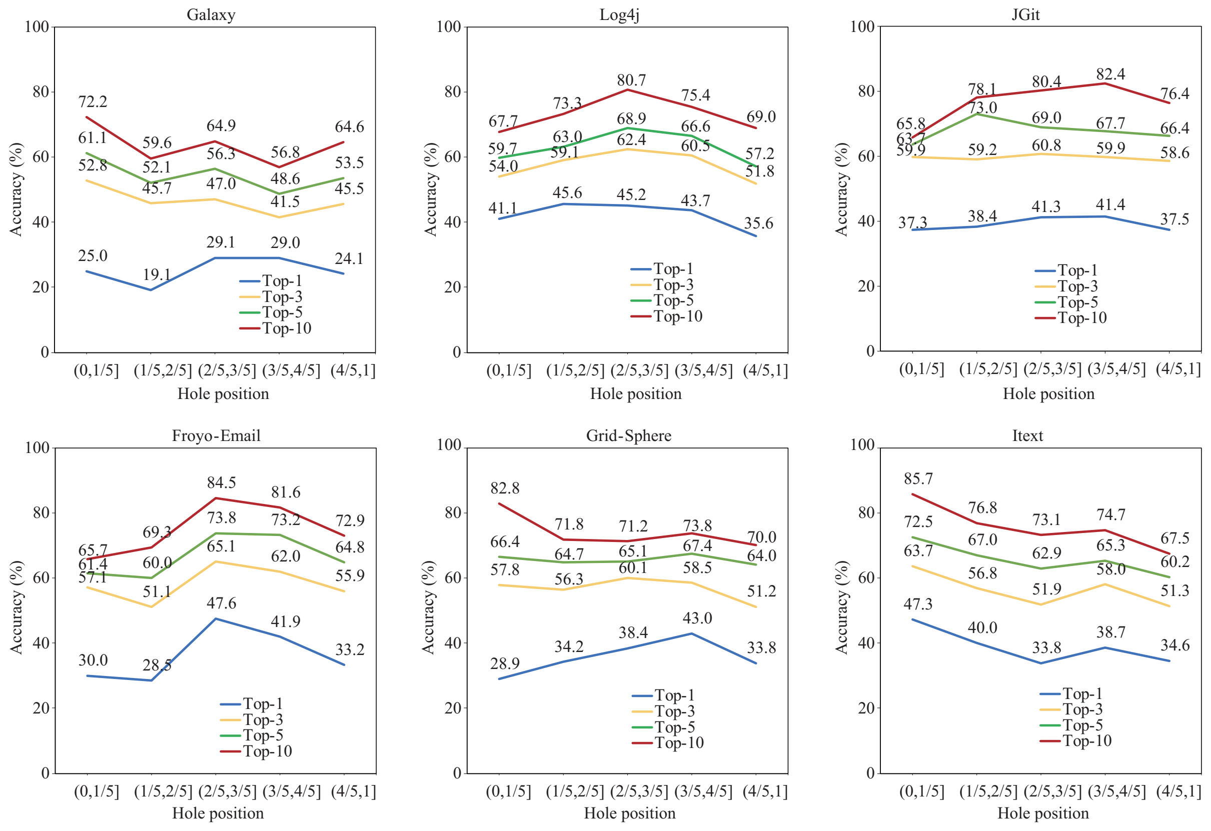

Figure 7 (Color online) Sensitivity analysis on hole position.

positive or negative correlation between the hole position and accuracy. Since we take the whole program (with the hole) as input, we can leverage its whole context information regardless of the hole position. Thus, we believe that the accuracy is insensitive to the hole position.

The results of the sensitivity analysis on the context proportion are shown in Figure 8. The accuracy of the code prediction usually can be improved with an increase in the context proportion when the context proportion is low (e.g., lower than 50%). However, the accuracy fluctuates when the context proportion is high (e.g., higher than 50%). This trend indicates that more context information can benefit code prediction but can also create noise for the prediction.

5.4 Usability analysis (RQ3)

To answer RQ3, we conduct a user study to compare the effectiveness and usability of DeepAPIRec and IntelliJ IDEA in helping developers to complete a segment of code. To provide a fair comparison with the code recommendation of IntelliJ IDEA, we integrate DeepAPIRec with a lightweight code editor that supports only basic code editing and syntax highlighting.

The tasks used in our study are chosen from Stack Overflow. We search for questions with the tag “Java” and order the questions by votes. For each question, we check if the answers include a suggested code segment that uses JDK APIs and has 4–10 lines of code. If a question has multiple suggested code segments that meet the requirements, we manually choose the best segment. Duplicated questions are eliminated. The top five questions that meet the requirements are selected as the tasks of our study and the corresponding code segments are used as the golden set. Finally, we have the following tasks: Convert InputStream to String (T1), Create ArrayList from array (T2), Iterate HashMap (T3), Convert stack trace to String (T4), Generate an md5 hash (T5). For each task, the question is used as the task description and the first two lines of code are given as the initial context.

We recruit 16 students, of which 2 are doctorate and 14 are master degree students, from our university. We ask them to complete a pre-experiment survey about their experience of Java. We then group them

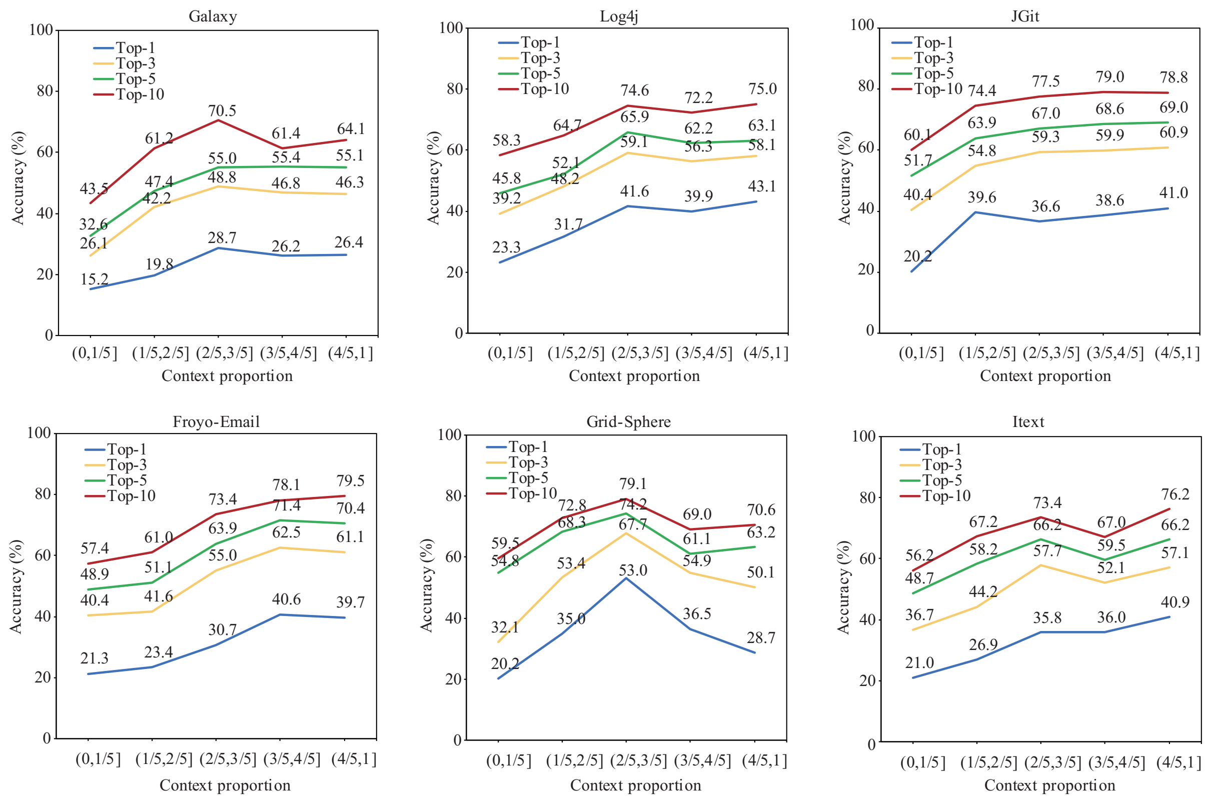

Figure 8 (Color online) Sensitivity analysis on context proportion.

into two groups (G1 and G2) of eight each according to their level of experience to ensure that the overall levels of the abilities of both groups are equivalent. We assign G1 to use DeepAPIRec and G2 to use IntelliJ IDEA.

The participants of the two groups are requested to complete the five tasks following the order of T1 to T5. For each task, they are given 15 min, after which they need to submit the task with its current implementation and move to the next task. They are not allowed to change the initial context (the two lines of code) or search the Internet for help but the participants in G2 can use the API reference documentation embedded in IntelliJ IDEA. After finishing each task, all participants are requested to evaluate the task's difficulty level (easy, medium, or hard). For each task, we compare the performances of both groups in terms of accuracy and completion time. The accuracies of the submitted results are evaluated with scores ranging from 0 to 10 by two Java experts. The scores are based on the consistency of the results with the golden set as the experts consider API calls, control units, and other minor elements (e.g., parameters), and syntax errors. In a few cases, the results are different from the golden set but are still accurate. These results are thus evaluated based on the alternative implementation strategies. Detailed descriptions and records of the tasks and results can be found in our replication package[7)].

The descriptive statistics of the usability evaluation by completion time and accuracy are presented in Tables 7 and 8, respectively. As the data are not in a normal distribution, we use the Mann-Whitney U test in our statistical analysis. We can see that the participants who use DeepAPIRec complete the tasks in significantly less time for T1, T2, T4, and T5 with $p < 0.05$ and in comparable time for T3, and their submitted results are significantly more accurate for T1 and T4 with $p < 0.05$ and comparable for T2, T3, and T5. According to the evaluation of the participants, T1 and T4 are hard; T3 and T5 are medium; and T2 is easy. We can see that DeepAPIRec is significantly better than IntelliJ IDEA for harder tasks and can help developers complete a segment of code significantly faster for most of the other tasks.

7) https://deepapirec2018.wixsite.com/deepapirec.

Table 7 Results of usability evaluation by completion time (s)

Task	Group	Average	Min	Max	Median	Standard deviation	p
T1	DeepAPIRec	365.0	84	566	348	152.22	0.033
	IntelliJ IDEA	665.3	220	900	731.5	255.23	
T2	DeepAPIRec	86.0	58	125	82	23.43	0.012
	IntelliJ IDEA	171.6	58	257	190.5	62.43	
T3	DeepAPIRec	209.8	76	390	194.5	102.74	0.113
	IntelliJ IDEA	358.1	46	900	345	245.23	
T4	DeepAPIRec	208.3	40	574	164.5	166.26	0.020
	IntelliJ IDEA	478.5	114	900	490.5	235.19	
T5	DeepAPIRec	142.3	34	279	143.5	74.53	0.004
	IntelliJ IDEA	379.9	124	900	332	210.19	

Table 8 Results of usability evaluation by accuracy

Task	Group	Average	Min	Max	Median	Standard deviation	p
T1	DeepAPIRec	6.1	3	10	5.5	2.47	0.019
	IntelliJ IDEA	3.1	0	7	2.5	2.42	
T2	DeepAPIRec	9.5	8	10	10	0.71	0.134
	IntelliJ IDEA	9.9	9	10	10	0.33	
T3	DeepAPIRec	8.1	5	10	9	2.03	0.478
	IntelliJ IDEA	7.9	3	10	8	2.20	
T4	DeepAPIRec	8.0	1	10	10	3.12	0.025
	IntelliJ IDEA	4.3	0	10	5	2.99	
T5	DeepAPIRec	8.1	6	10	8.5	1.90	0.136
	IntelliJ IDEA	9.4	6	10	10	1.32	

Note that the average accuracies of the participants using DeepAPIRec are lower than those of the participants using IntelliJ IDEA for T2 and T5. After analyzing the data, we obtain the following findings. The current code editor for DeepAPIRec has no syntax checking, so the results submitted by some participants using DeepAPIRec have syntax errors, such as missing return statements. This problem can be solved by integrating DeepAPIRec with existing IDEs, such as Eclipse and IntelliJ IDEA. On the other hand, some participants using DeepAPIRec neglect the checking and changing of wrongly recommended parameters. This problem implies the necessity of reminding the user to check the suggested parameters beacuse DeepAPIRec recommends a statement as a whole and may recommend correct API calls but with the wrong parameters.

After the experiment, we conduct a group interview with the participants to obtain their feedback and suggestions. The general feedback is that DeepAPIRec can accurately recommend the required API calls, and in most cases, the right API calls are included in the top ten recommendations. Hence, DeepAPIRec can help when the user has no idea of which APIs to use. In contrast, IntelliJ IDEA requires the user to provide the class/object names before recommending the methods/properties. The participants agree that the statements recommended by DeepAPIRec are complete, and in many cases, they can use the recommended statements with little, or even no changes. In many cases, DeepAPIRec can accurately recommend a series of successive statements so that the user needs only to examine and choose the right recommendations. A key to a successful recommendation is the choice of the right statement from the recommendations. The participants mainly rely on the names of the recommended API calls to decide their relevance to the given task. Some participants suggest that easy-to-understand explanations of the recommended statements be provided to allow them to better choose the right statements.

5.5 Threats to validity

The threats to the external validity of our studies mainly lie in the fact that the training samples, test samples, and tasks are still limited. All of the training and test samples are obtained from open source Java projects, so the results may not be generalized to industrial projects or projects of different languages or sizes. Our current studies focus on the JDK library. The results may not be applicable for other libraries. The test samples in our studies are produced by removing code elements at arbitrary positions, so they may not fully reflect the characteristics of code prediction scenarios in the real-world. The tasks used in the user study are simple tasks independent of complex project contexts, so the results may not be generalized to tasks that are a part of a project.

The threats to the internal validity lie in our implementation of the experiments and the data statistics. The baseline approaches used in our studies may not be properly implemented or trained. The results of the user study are evaluated subjectively by two experts. To reduce bias, the two experts are required to reach an agreement in each evaluation.

6 Related work

The vast accumulation of source code and related documentation (i.e., big data in software development) has boosted data-driven software automation [18] for a variety of tasks such as code recommendation [4, 6, 8], program generation [19] and optimization [20, 21]. Besides the approaches mentioned in the previous sections, this study is related to the following studies.

Code completion. Traditional code completion approaches [22–24] are focused on completing method calls for given API classes or objects. These approaches mainly consider the strategies of sorting, filtering, and grouping candidate API methods based on the type information. Mandelin et al. [25] proposed an approach to synthesize code fragments for given input and output types by matching API method signatures and mining existing client programs. Their approach includes a type matching technique that can generate parameters for API methods. Unlike our parameter concretization technique, their technique does not consider the relation between the current method call and the code context. Some approaches [26, 27] consider the similarity between the current code context and previous code examples for code completion. Other example approaches include [4, 6, 8, 28–30]. These approaches often begin with the conjecture that the source code is naturally repetitive and predictable [7], so their models learn statistically from local code corpuses. Allamanis et al. [28] trained an N-gram model that uses a large corpus, instead of a small one, of Java code from a wide variety of domains. Nguyen et al. [30] proposed a statistical semantic language model, called SLAMC, that enhances the N-gram model by incorporating semantic information (such as data type and the role of a code token) into code tokens and global technical concerns/functionality using an N-gram topic model. Tu et al. [29] proposed an augmented N-gram model that uses a cache to capture the localized regularities, instead of simply using global regularities, in the source code. Raychev et al. [8] proposed a method that reduces the code completion problem into an NLP problem of predicting the possibility of a sentence and trained a statistical language model based on the sentences extracted from programs. Nguyen et al. [6] proposed an approach based on statistical learning from fine-grained code changes, which leverages the change context and code context to predict API calls. Though there are enhancements in [6, 8, 28–30], these studies do not consider the structure information of the source code. In comparison, DeepAPIRec incorporates the structure information of the source code to better learn the semantic and syntactic information. There also exist approaches that take the structure information into consideration. For example, Nguyen et al. [4] proposed GraLan, a graph-based statistical language model for code completion, which represents source code as graphs and uses Bayesian statistical inference to compute the probability of a new graph given any (sub)graphs as context but treats all contexts equally and cannot effectively model variable-length code context. In comparison, when fed into the Tree-LSTM network, DeepAPIRec can recognize which context is more useful and can effectively model the variable-length code context. In addition, DeepAPIRec applies statistical learning to predict API parameters. By combining statistical learning and deep learning, our method is both generative and statistically accurate. All above approaches are static, but there also exist dynamic approaches. For example, Galenson et al. [31] proposed a dynamic approach, that leverages dynamic information, such as dynamic type and dynamic value to complete the code at runtime.

Mining API usage patterns. This study is related to studies, such as [32–35], on API usage pattern mining. Existing approaches often use deterministic mining algorithms to capture frequent API usage patterns through pair associations, co-occurrences, frequent subsequences, or subgraphs. Zhong et al. [34] proposed MAPO, which first clusters code snippets, and then uses frequent subsequence mining to mine API usage patterns. Nguyen et al. [35] proposed GrouMiner, which is a graph-based approach that represents usage patterns as graphs (called groum) for mining. Wang et al. [33] proposed an approach that uses a two-step clustering strategy for call sequences before and after applying a sequence mining algorithm to mine patterns. Fowkes et al. [32] presented PAM, a near parameter-free probabilistic algorithm that uses a probabilistic model based on a set of API patterns and a set of probabilities to infer the most interesting API patterns. Our approach differs from these studies by capturing API usage patterns with not only API calls but also other relevant statements.

Deep learning. This study is closely related to approaches to applying deep learning for code recommendation and completion. Some recent studies, such as [2, 19, 36–38], have explored the application of deep leaning to analyzing the source code. For example, Mou et al. [19] trained a sequence-to-sequence recurrent neural network, which takes the brief comment of a user intention as input and generates the corresponding code character-by-character. Mou et al. [36] also applied deep learning techniques for other code applications. They proposed a tree-based convolutional network, called TBCNN, to model the source code based on ASTs, which can be used for classifying programs and detecting code snippets of certain patterns. Gu et al. [2] used an RNN encoder-decoder to learn the semantics of a natural language query and generate related API usage sequences. Allamanis et al. [37] proposed an attentional neural network that takes a sequence of code tokens as input and produces an extreme summary of these code tokens by employing convolutions on the code tokens. For defect prediction, Wang et al. [38] used a deep belief network to automatically learn features of tokens extracted from the source code. Our study differs from the abovementioned approaches in terms of neural network models (i.e., Tree-LSTM) and, more importantly, the objective (i.e., statement-level code recommendation).

7 Conclusion

We propose and implement a generative API usage code recommendation approach, called DeepAPIRec, based on a tree-based LSTM neural network. DeepAPIRec can model and reason about variable-length, preceding and succeeding code contexts to make precise predictions and generate concrete parameters in order to provide complete code recommendations. Our experimental studies on the JDK library show that DeepAPIRec can achieve high accuracy for statement-level recommendations. Our user study further confirms that DeepAPIRec can help developers to complete a segment of code faster and more accurately than can IntelliJ IDEA. Our future work will further improve the accuracy and coverage of code recommendations by utilizing developers' knowledge in mind [39] and better trained neural networks.

Acknowledgements This work was supported by National Key Research and Development Program of China (Grant No. 2016YFB1000801), and Shanghai Science and Technology Development Funds (Grant No. 16JC1400801).

References

1 Stylos J, Myers B A. Mica: a web-search tool for finding API components and examples. In: Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, Brighton, 2006. 195–202

2 Gu X D, Zhang H Y, Zhang D M, et al. Deep API learning. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 631–642

3 Raghothaman M, Wei Y, Hamadi Y. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis. In: Proceedings of the 38th International Conference on Software Engineering, Austin, 2016. 357–367

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

5 Nguyen A T, Nguyen T T, Nguyen H A, et al. Graph-based pattern-oriented, context-sensitive source code completion. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 69–79

6 Nguyen A T, Hilton M, Codoban M, et al. API code recommendation using statistical learning from fine-grained

changes. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 511–522

7 Hindle A, Barr E T, Su Z D, et al. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 837–847

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

9 Graves A, Jaitly N, Mohamed A. Hybrid speech recognition with deep bidirectional LSTM. In: Proceedings of IEEE Workshop on Automatic Speech Recognition and Understanding, Olomouc, 2013. 273–278

10 Socher R, Karpathy A, Le Q V, et al. Grounded compositional semantics for finding and describing images with sentences. Trans Association Comput Linguist, 2014, 2: 207–218

11 Tai K S, Socher R, Manning C D. Improved semantic representations from tree-structured long short-term memory networks. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, Beijing, 2015. 1556–1566

12 Zhang X X, Lu L, Lapata M. Top-down tree long short-term memory networks. In: Proceedings of Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego, 2016. 310–320

13 Duchi J C, Hazan E, Singer Y. Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res, 2011, 12: 2121–2159

14 Montemurro M A, Zanette D H. Universal entropy of word ordering across linguistic families. PLoS ONE, 2011, 6: e19875

15 Looks M, Herreshoff M, Hutchins D, et al. Deep learning with dynamic computation graphs. 2017. ArXiv: 1702.02181

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

17 Dam H K, Tran T, Pham T. A deep language model for software code. 2016. ArXiv: 1608.02715

18 Mei H, Zhang L. Can big data bring a breakthrough for software automation? Sci China Inf Sci, 2018, 61: 056101

19 Mou L L, Men R, Li G, et al. On end-to-end program generation from user intention by deep neural networks. 2015. ArXiv: 1510.07211

20 Zhou X, Wu K D, Cai H Q, et al. LogPruner: detect, analyze and prune logging calls in Android apps. Sci China Inf Sci, 2018, 61: 050107

21 Huang G, Cai H Q, Swiech M, et al. DelayDroid: an instrumented approach to reducing tail-time energy of Android apps. Sci China Inf Sci, 2017, 60: 12106

22 Pletcher D M, Hou D Q. BCC: enhancing code completion for better API usability. In: Proceedings of the 25th IEEE International Conference on Software Maintenance, Edmonton, 2009. 393–394

23 Hou D Q, Pletcher D M. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, Cape Town, 2010. 26–30

24 Hou D Q, Pletcher D M. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In: Proceedings of IEEE 27th International Conference on Software Maintenance, Williamsburg, 2011. 233–242

25 Mandelin D, Xu L, BodíR, et al. Jungloid mining: helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, 2005. 48–61

26 Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 213–222

27 Asaduzzaman M, Roy C K, Schneider K A, et al. A simple, efficient, context-sensitive approach for code completion. J Softw Evol Proc, 2016, 28: 512–541

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

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

30 Nguyen T T, Nguyen A T, Nguyen H A, et al. A statistical semantic language model for source code. In: Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Saint Petersburg, 2013. 532–542

31 Galenson J, Reames P, BodíR, et al. CodeHint: dynamic and interactive synthesis of code snippets. In: Proceedings of the 36th International Conference on Software Engineering, Hyderabad, 2014. 653–663

32 Fowkes J M, Sutton C A. Parameter-free probabilistic API mining across GitHub. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 254–265

33 Wang J, Dang Y N, Zhang H Y, et al. Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, 2013. 319–328

34 Zhong H, Xie T, Zhang L, et al. MAPO: mining and recommending API usage patterns. In: Proceedings of the 23rd European Conference on Object-Oriented Programming, Genoa, 2009. 318–343

35 Nguyen T T, Nguyen H A, Pham N H, et al. Graph-based mining of multiple object usage patterns. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 383–392

36 Mou L L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, Phoenix, 2016. 1287–1293

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

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

39 Peng X, Xing Z C, Pan S, et al. Reflective feature location: knowledge in mind meets information in system. Sci China Inf Sci, 2017, 60: 072102