8-2022

Holistic combination of structural and textual code information for context based API recommendation

Chi CHEN
Fudan University

Xin PENG
Fudan University

Zhengchang XING
Australian National University

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Xin WANG
Fudan University

See next page for additional authors

Holistic Combination of Structural and Textual Code Information for Context based API Recommendation

Chi Chen, Xin Peng, *Member, IEEE,* Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao

Abstract—Context based API recommendation is an important way to help developers find the needed APIs effectively and efficiently. For effective API recommendation, we need not only a joint view of both structural and textual code information, but also a holistic view of correlated API usage in control and data flow graph as a whole. Unfortunately, existing API recommendation methods exploit structural or textual code information separately. In this work, we propose a novel API recommendation approach called APIRec-CST (API Recommendation by Combining Structural and Textual code information). APIRec-CST is a deep learning model that combines the API usage with the text information in the source code based on an API Context Graph Network and a Code Token Network that simultaneously learn structural and textual features for API recommendation. We apply APIRec-CST to train a model for JDK library based on 1,914 open-source Java projects and evaluate the accuracy and MRR (Mean Reciprocal Rank) of API recommendation with another 6 open-source projects. The results show that our approach achieves respectively a top-1, top-5, top-10 accuracy and MRR of 60.3%, 81.5%, 87.7% and 69.4%, and significantly outperforms an existing graph-based statistical approach and a tree-based deep learning approach for API recommendation. A further analysis shows that textual code information makes sense and improves the accuracy and MRR. The sensitivity analysis shows that the top-k accuracy and MRR of APIRec-CST are insensitive to the number of APIs to be recommended in a hole. We also conduct a user study in which two groups of students are asked to finish 6 programming tasks with or without our APIRec-CST plugin. The results show that APIRec-CST can help the students to finish the tasks faster and more accurately and the feedback on the usability is overwhelmingly positive.

Index Terms—API, recommendation, deep learning, data flow, control flow, text

1 Introduction

In modern software development, developers heavily rely on APIs (Application Programming Interfaces). When developers do not know which API(s) to use for a desired feature, automatic API recommendation is an important way to help developers find the needed APIs effectively and efficiently. In general, API recommendation methods learn explicit or implicit API usage patterns from a large code base and then match partially written code with the patterns to recommend APIs. Existing methods differ in the types of code information they model and how they model code information.

Source code contains two core types of information: structural and textual. Structural code information, such as control and data flow, represents program logic which can be captured using a graph representation; textual code information, such as code comments, method names, variable names, reflects the semantics of the code in natural language. Take the code snippet in Fig. 1 as an example. Note that the correct API statement at line 8 should be $hashCode = str.hashCode()$. The method name "computeHashCode" and the variable name "hashCode" reflect the intent of this method (assuming the proper tokenization of these names). The method body uses multiple APIs which implement three pieces of correlated program logics: 1) use a reader to read contents from a file line by line (line 3/4/5/6/11/12); 2) compute the hash code of the content (line 8); 3) add the hash value into a created list (line 2/7/9). These program logics can be modeled in a control and data flow graph as shown in Fig. 5. Note that variable names (e.g., "path", "result", "rd", "br", "str", "hashCode") are helpful for the understanding of relevant structural program logics.

For effective API recommendation, we need not only a joint view of both structural and textual code information, but also a holistic view of correlated API usage in control and data flow graph as a whole. Unfortunately, existing API recommendation methods exploit structural or textual code information separately. Based on the observation of linguistic naturalness of source code [1], many approaches [1], [2], [3], [4] have been proposed that rely on statistical language models for code auto-completion and API recommendation. The adopted statistical language models can be simple or enhanced n-gram model [1], [2], [3], [4] or complex deep learning models (e.g., Recurrent Neural Network (RNN)) [5], [6], [7]. No matter which types of statistical language models to use, these approaches treat code as a sequence of text tokens (which may sometimes be enriched with simple syntactic information such as program construct keywords and data types), but do not exploit

- *X. Peng is the corresponding author.*
- *C. Chen, X. Peng, X. Wang, Y. Zhao and W. Zhao are with the School of Computer Science and the Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, and Shanghai Institute of Intelligent Electronics & Systems, China.*
- *Z. Xing is with the Australian National University, Australia.*
- *J. Sun is with the Singapore Management University, Singapore.*

structural code information of source code. As such, they cannot properly model the long-range dependencies between correlated but far-away API usage due to the limitation of the length of a sequence.

To overcome the limitation of token-sequence-based API recommendation, another important line of API recommendation methods [8], [9] analyze control and data flow graph for recommending APIs. However, these methods usually base their recommendation on the enumeration of control and data flow subgraphs, but lack a holistic view of the overall program logic. Consider the code snippet in Fig. 1. Fig. 3 shows nine control-and-data-flow subgraphs for this code snippet. Assume developers do not know the "java.lang.String.hashCode" API to be used at line 8. Unfortunately, existing methods recommend "java.io.BufferedReader.readLine" based on the fourth subgraph in Fig. 3 or "while" based on the sixth subgraph. Different subgraphs are treated independently for recommending relevant APIs. As smaller subgraphs usually appear more frequently than larger subgraphs, APIs from smaller subgraphs that capture only a partial aspect of the overall program logic often overshadow APIs from larger subgraphs that capture more holistic view of the program logic.

In this work, we propose a novel API recommendation approach called APIRec-CST (**API Rec**ommendation by **C**ombining **S**tructural and **T**extual code information), which addresses the limitation of independent modeling of structural and textual code information and the lack of holistic reasoning of code structure in existing API recommendation approaches. APIRec-CST is a deep learning model that combines the API usage with the text information in the source code based on an API Context Graph Network and a Code Token Network. As such, it can simultaneously learn structural and textual features for API recommendation. APIRec-CST uses an API context graph to model API usage in a control and data flow graph for the entire method, rather than independent partial subgraphs as in existing methods [8]. Our API context graph contains the holistic semantics of the API usage in the source code around the location for API recommendation. From this API context graph, the API Context Graph Network learns to extract informative structural features for API recommendation. The textual code information in the source code, such as method names, parameter names and variable names, is processed as a bag of code tokens which is fed into the Code Token Network to infer the developer's intent jointly with the API Context Graph Network.

We conduct a series of experiments to evaluate the effectiveness of APIRec-CST. Our results show that APIRec-CST significantly outperforms an existing graph-based statistical approach and a tree-based deep learning approach for API recommendation. The overall top-1 accuracy of APIRec-CST is about 60.3%, the top-5 accuracy is about 81.5%, the top-10 accuracy is about 87.7% and the MRR is about 69.4%. In addition, our analysis shows that textual code information makes sense and improves the accuracy and MRR. The sensitivity analysis shows that the top-k accuracy and MRR of APIRec-CST are insensitive to the number of APIs to be recommended in a hole. The results of our user study with 18 students and 6 programming tasks show that APIRec-CST can help the students finish the tasks faster and more accurately and the feedback on our tool's usability is overwhelmingly positive.

```
1:  public List<Integer> computeHashCode(String path) throws Exception{
2:      List<Integer> result = new ArrayList<>();
3:      FileReader rd = new FileReader(path);
4:      BufferedReader br = new BufferedReader(rd);
5:      String str = null;
6:      while((str = br.readLine()) != null){
7:          int hashCode;
8:          $hole$;
9:          result.add(hashCode);
10:     }
11:     br.close();
12:     rd.close();
13:     return result;
14: }
```

Fig. 1. Example of Computing HashCode of Content from File

```
1:  public List<Integer> getIntegerScore(String path) throws Exception{
2:      List<Integer> result = new ArrayList<>();
3:      FileReader rd = new FileReader(path);
4:      BufferedReader br = new BufferedReader(rd);
5:      String str = null;
6:      while((str = br.readLine()) != null){
7:          int score;
8:          $hole$;
9:          result.add(score);
10:     }
11:     br.close();
12:     rd.close();
13:     return result;
14: }
```

Fig. 2. Example of Getting Integer Score from File

The main contributions of this work are as follows:

- We propose an API recommendation approach called APIRec-CST that combines structural and textual code information in the source code by jointly learning a graph-based deep learning model and a token-based deep learning model for effective API recommendation.
- We implement APIRec-CST as a tool that supports the efficient model training and API inference with GPU acceleration.
- We evaluate the effectiveness of APIRec-CST for recommending APIs with both automatically constructed test instances and real programming tasks.

2 MOTIVATION

We use the code examples in Fig. 1 and Fig. 2 to motivate the need for holistic combination of structural and textual code information for API recommendation. The example in Fig. 1 is to implement a method to compute the hash code of the content from a file line by line and then adds the computed hash code into a list. The developer has written the code he knows and needs help to complete the remaining code. The line marked as *hole* is the location that the developer requests the recommendation of proper APIs for computing the hash code of the content of a string.

We can see that this program contains rich structural code information (i.e., multiple APIs and control and data flow among these APIs). We can get many subgraphs of

different sizes according to control and data flow, such as the nine subgraphs shown in Fig. 3. Note that each subgraph is labeled with a serial number for the convenience of discussion. We do not list all the subgraphs for the code in Fig. 1 due to the space limitation. As we can see, each subgraph reflects partial program logic (semantics). For example, the seventh subgraph reflects the semantics of creating readers for reading a file. As another example, the fifth subgraph reflects the semantics of reading contents line by line. None of the subgraphs (including those not listed in the paper) independently can reflect the expected semantics (i.e., computing the hash code of a string) at the location of *hole*.

If the developer uses existing tools such as GraLan [8] that recommends APIs based on such subgraphs, he cannot get the correct API recommendation. Table 1 lists the top-10 recommendations by GraLan. The first column is the ranking of each recommendation. The second column lists the ten recommendations. The third column is the serial number of the subgraphs in Fig. 3 used as the parent graph based on which the corresponding recommendation is generated. In GraLan, each subgraph is considered as a context parent graph to generate child graphs (each child graph has one more node than its parent graph and the extra node is considered as a candidate API recommendation). From Table 1, we can see that the top-10 recommendations by GraLan are generated based on partial program semantics and thus miss the correct recommendation.

In order to recommend the correct API, we need a holistic view of the overall program logic in the entire method. Hence, we represent the API usage in a whole control and data flow graph called API context graph (as shown in Fig. 5) instead of subgraphs for the entire method. The API context graph is a directed graph (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges, which models the entire API usage of the source code. Each node in N represents an API method call, an API field access, a variable declaration, an assignment, a control unit or a hole. Each edge represents a flow relationship (such as control flow and data flow) between two nodes. The API context graph not only contains all semantics in subgraphs, but also integrates these semantics as a whole. The details of how to construct an API context graph will be introduced in Section 4.1. From the API context graph, we can see that it contains the following two major semantics: semantics-1) use a reader to read contents from a file line by line; semantics-2) add a value into a created list. Since these semantics are in one entire graph, they can be integrated to infer the semantics at the hole.

When observing these two semantics in a holistic view, we can find that the declared $String$ variable "str" is just used to store the content from the file but not used any more in semantics-1. Furthermore, the declared int variable "hashCode" is not assigned a value in semantics-2. In addition, there lack of APIs to connect semantics-1 and semantics-2 to make the program logic complete. From this holistic view, we can infer that the semantics at the *hole* is to get a value of int type based on some kind of processing of a variable of $String$ type. Note that the subgraph can be a whole graph in GraLan, but the larger a graph is, the less frequent it may occur in the training data which

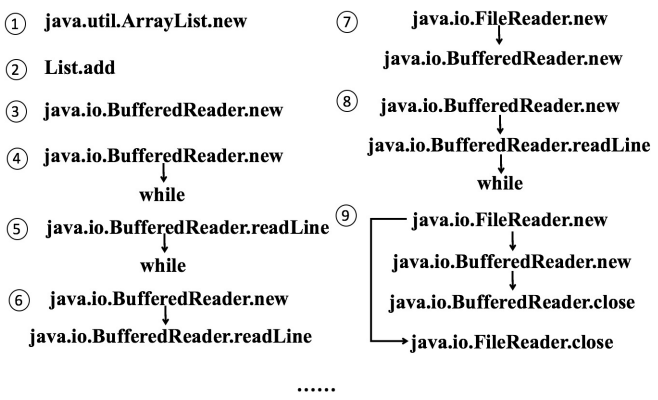

Fig. 3. Control-and-Data-Flow Subgraphs of the Code in Fig. 1

TABLE 1
Top-10 Recommendations by GraLan [8] for the Code Snippet in Fig. 1

Rank	Recommendation	Parent Graph
1	java.util.List.add	1
2	java.util.ArrayList.new	2
3	java.io.BufferedReader.readLine	4
4	java.io.BufferedReader.new	5
5	while	6
6	java.io.BufferedReader.close	8
7	if	2
8	for	2
9	java.util.ArrayList.add	1
10	java.io.InputStreamReader.new	3

may cause the data sparsity issue. Our deep learning model learns a vector representation for each entire graph based on an information diffusion mechanism of all nodes and edges. In this way, each entire graph that has a distinct semantics will have a meaningful vector representation, no matter how large the graph is and how frequent the graph occurs in the code base. As such, our model does not suffer from the data sparsity issue.

However, we still cannot recommend the exact API needed at the $hole$ in Fig. 1, if we just consider the structural code information in this example. This is because we cannot decide what kind of processing should be performed on the variable of $String$ type. Let us see the code snippet in Fig. 2. The developer needs to implement a method to read scores stored in a file, convert each score to an integer and add it into a list for further use. We can see that the code in Fig. 2 is structurally very similar to the code in Fig. 1, because the program logics for reading file and list addition are the same. The API context graph of the code in Fig. 2 is the same as that of the code in Fig. 1, but the expected APIs at *hole* are different. If the developer requests API recommendation for these two code snippets, we should distinguish the different intents in the two code snippets. To that end, textual code information in code becomes very useful for inferring code intents. In Fig. 1, the method name "computeHashCode" and variable name "hashCode" imply that the processing on the variable of $String$ type is likely relevant to hash code processing. In Fig. 2, the method name "getIntegerScore" and variable name "score" can imply that the processing on the variable of $String$ type is likely relevant to String-Integer conversion.

To sum up, a joint view of both structural and tex-

TABLE 2
Top-10 Recommendations by APIRec-CST for the Code Snippet in Fig. 1

Rank	Recommendation
1	if
2	java.lang.String.hashCode
3	java.util.ArrayList.add
4	java.lang.String.indexOf
5	java.lang.Integer.parseInt
6	java.lang.String.length
7	while
8	java.lang.String.trim
9	java.lang.String.indexOf
10	java.lang.String.split

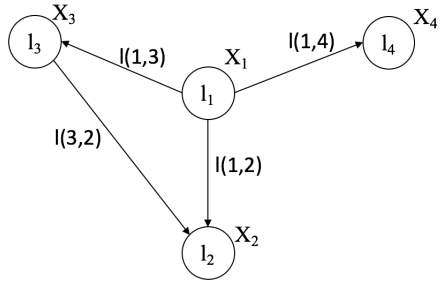

Fig. 4. Graph Example

tual code information and a holistic view of correlated API usage in control and data flow graph of the entire method is required for effective API recommendation. When combining structural and textual code information and learn API usage in control and data flow graph of an entire method from a holistic view, our proposed approach (named APIRec-CST) successfully recommends the correct API $java.lang.String.hashCode$ for the code in Fig. 1 and the correct API $java.lang.Integer.parseInt$ for the code in Fig. 2 separately. Take the code in Fig. 1 as an example, Table 2 shows the top-10 recommendations provided by APIRec-CST (Note that the parameter type of the fourth recommendation is $java.lang.String$, while the parameter type of the ninth recommendation is int). We can see that most of the recommendations are related to the semantics of getting a value of int type based on some kind of processing of a variable of $String$ type, which benefits from a holistic view of correlated API usage in control and data flow graph of an entire method. Among these recommendations, $java.lang.String.hashCode$ is successfully recommended as the most possible processing, which benefits from combining structural and textual code information.

3 Background

In this work, we adopt Graph Neural Networks (GNNs), in particular, Gated Graph Neural Networks (GG-NNs) [10], for API recommendation. An API usage can be naturally represented in the form of a graph where the nodes represent APIs and edges represent control/data flow between nodes. Furthermore, the nodes and edges can be labeled with additional context information, e.g., the nodes can be labeled with API calls and the edge labels can be used to distinguish control flow and data flow.

GNNs are a neural network model which take graph structures as inputs. GNNs are based on an information diffusion mechanism and work effectively for a variety of graphs, e.g., directed or undirected graphs and cyclic or acyclic graphs. In GNNs, each node of the graph is represented as a corresponding unit in the neural network, and the connectivity among the units is the same as the connectivity among nodes in the graph. The unit captures the current state of a node and is used to compute the next state of the node when activated. The units update their states and exchange information until they reach a stable equilibrium [11]. The state of a node is composed of the label of the node, the labels of its incoming and outgoing edges and the states and labels of neighbor nodes with a parametric function. Formally, a state $\mathbf{x}_n$(t) at the tth iteration of a node n is defined as follows [11].

$$\mathbf{x}_n(t) = \mathbf{f}_w(\mathbf{l}_n, \mathbf{l}_{co[n]}, \mathbf{x}_{ne[n]}(t-1), \mathbf{l}_{ne[n]}), \quad (1)$$

where $\mathbf{f}_w$ is a parametric function, $\mathbf{l}_n$ is the label of node n, $\mathbf{l}_{co[n]}$ are the labels of edges containing node n, $\mathbf{x}_{ne[n]}$(t-1) are the states of nodes in the neighborhood of node n at the $(t-1)$th iteration, and $\mathbf{l}_{ne[n]}$ are the labels of nodes in the neighborhood of node n. In this way, each node can get a node representation. Take the graph in Fig. 4 as an example. The state $\mathbf{x}_1$ of node 1 at time t is computed as $\mathbf{x}_1$(t)=$\mathbf{f}_w$($\mathbf{l}_1$,$\mathbf{l}_{(1,2)}$,$\mathbf{l}_{(1,3)}$,$\mathbf{l}_{(1,4)}$,$\mathbf{x}_2$(t-1),$\mathbf{x}_3$(t-1),$\mathbf{x}_4$(t-1),$\mathbf{l}_2$,$\mathbf{l}_3$,$\mathbf{l}_4$), where $\mathbf{l}_1$ is the label of node 1, $\mathbf{l}_{(1,2)}$, $\mathbf{l}_{(1,3)}$, $\mathbf{l}_{(1,4)}$ are the labels of edges connected with node 1, $\mathbf{x}_2$(t-1), $\mathbf{x}_3$(t-1), $\mathbf{x}_4$(t-1) are the states of the neighboring nodes (i.e., node 2, node 3 and node 4) of node 1 at time t-1 and $\mathbf{l}_{(1,2)}$, $\mathbf{l}_{(1,3)}$, $\mathbf{l}_{(1,4)}$ are the labels of these neighbors of node 1. The state of a node is connected with other nodes in the graph as nodes can communicate with each other based on the information diffusion mechanism. Through training, GNNs can be applied for subgraph matching, mutagenesis, and web page ranking [11].

GG-NNs [10] are based on GNN. The difference is that GNNs apply Almeida-Pineda algorithm [12], [13] for computing gradients, whereas GG-NNs apply back-propagation through time with Gated Recurrent Units [14] for computing gradients. GG-NNs use a soft attention mechanism to decide which nodes are more relevant to compute the final vector representation of the graph. The graph level representation vector $\mathbf{x}_g$ is computed as follows [10]. Specifically, the soft attention mechanism takes the states of the nodes (units) as input and computes the weight of each node (unit) through neural networks and a sigmoid function based on updating the parameters of the neural networks during training to give higher weights to the nodes (units) that make more contribution to predicting the correct API.

$$\mathbf{x}_g = tanh\left(\sum_{n\in N} \sigma\Big(i(\mathbf{x}_n(t), \mathbf{l}_n)\Big) \odot tanh\Big(j(\mathbf{x}_n(t), \mathbf{l}_n)\Big)\right), \quad (2)$$

where σ(i($\mathbf{x}_n$(t),$\mathbf{l}_n$)) works as a soft attention mechanism, i and j are neural networks taking as input the concatenation of $\mathbf{x}_n$(t) and $\mathbf{l}_n$ and output real-valued vectors [10], and $\odot$ is element-wise multiplication.

To get a graph representation, GNNs require creating a dummy super node which is connected to all other nodes by a special type of edge [10]. Doing so in our context may destroy the structural code information of the source code itself. In addition, the soft attention mechanism of GG-NNs

can help us to identify which nodes (i.e., APIs) in the API context graph are more important for API recommendation. In GG-NNs, the final representation of a graph is the accumulated information of each node with its importance computed through the soft attention mechanism. In this way, the final representation of a graph is a holistic representation of all nodes. Therefore, we choose GG-NNs as our deep neural networks to learn the features of API context graphs from a holistic view. More details of GNNs and GG-NNs can be referred to [10], [11].

4 APPROACH

In this section, we present the detailed design of APIRec-CST. It takes a program with a hole as input, and outputs a ranked list of API recommendations for filling the hole.

4.1 Program Representation

Given a program with a hole, APIRec-CST first constructs an API context graph and a bag of code tokens. The API context graph is a graph representation of structural code information of the user-provided program, whereas the code tokens (including the method name, parameter names and variable names) capture the textual code information.

An API context graph is a directed graph (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. Each node in N represents an API method call, an API field access, a variable declaration, an assignment, a control unit or a hole. Furthermore, each node is labeled differently according to its type. Table 3 shows how each type of node is labeled. Specifically, if the type of a node is a variable declaration initialized with an API method call, the label of the node is processed as the API method call. For example, the label of $int\ length = str.length()$; is $java.lang.String.length()$. Similarly, if the type of a node is a variable declaration initialized with an API field access, the label of the node is processed as the API field access. The assignment statement is processed the same as the variable declaration statement. We use a special node labeled with $Hole$ (called hole node hereafter) to represent the hole. There is an edge $(n, n') \in E$ if and only if one of the following conditions is satisfied.

- There is a direct control flow from n to n';
- There is a direct data flow from n to n';
- n' is the hole node and n is a node representing the preceding statement in the program or n is the hole node and n' is a node representing the subsequent statement in the program.

Given an edge (n, n'), we say that n is the parent node of n' and n' is the child node of n. In APIRec-CST, the edges in an API context graph are distinguished by labeling them with different types, i.e., an edge is labeled *control flow* (Type c) if there is direct control flow and no direct data flow; an edge is labeled *data flow* (Type d) if there is direct data flow and no direct control flow; an edge is labeled *control and data flow* (Type cd) if there are both direct control flow and direct data flow; and an edge is labeled *special flow* (Type s) if its source node or target node is the hole. Note that the *special flow* edge makes sure that the hole node is connected to its context.

Given a program, APIRec-CST systematically builds the API context graph statically. First, APIRec-CST builds the AST (i.e., Abstract Syntax Tree) of the program. Then it creates nodes and edges in the API context graph for each statement in the program based on the AST in the following way.

- If the statement is an API method call, an API field access, a variable declaration or an assignment, a node is created according to the corresponding node type in Table 3. Note that if the parameter of an API method call is also an API method call or an API field access, APIRec-CST first creates a node for the parameter.
- If the current statement is an expression that includes several API method calls or API field accesses, APIRec-CST creates a node for each API method call or API field access one by one.
- If the current statement is a control statement, APIRec-CST creates a node for the control unit according to its type and several other nodes together with edges connecting them as shown in Table 4. For example, if the current statement is a *while* statement, APIRec-CST first creates a *While* node, a *Condition* node, and a *Body* node. Two Type c edges are introduced, one from the *While* node to the *Condition* node and the other from the *While* node to the *Body* node.

Next, we systematically analyze the control and data dependencies between the nodes (i.e., between the corresponding statements) and introduce the edges accordingly. Take the *while* statement as an example. A Type c edge is added from the *Condition* node to the first node created for the condition expression and a Type c edge is added from the *Body* node to the first node created for the loop body. In addition, a Type c edge is added from the *While* node to the first node representing the statement following the loop. If the program contains a hole, a Type s edge is added from the node representing the statement preceding the hole to the hole node and a Type s edge is added from the hole node to the node representing the statement succeeding the hole. To analyze the data dependencies, if a node represents a variable declaration statement or an assignment statement, which means that this node contains a variable or an object that can be used as a receiver or a parameter in other API calls, we store its variable or object. Then if the stored variable or object of a node is used as a receiver or a parameter in another node representing an API call, a Type d edge is added from the previous node to the latter node. Specifically, if there already exists control flow between these two nodes, a Type cd edge is added between these two nodes.

For instance, the API context graph for the program shown in Fig. 1 is shown in Fig. 5, where solid lined triangle arrows represent edges labeled with control flow; dashed lined triangle arrows represent edges labeled with data flow; solid lined diamond arrows represent edges labeled with control and data flow; and dotted lined triangle arrows represent edges labeled with special flow. We can see that different from the graph used in GraLan, each edge is given a type in our API context graph and the structure of our API context graph is closely related to the program structure. In

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3074309, IEEE Transactions on Software Engineering

TABLE 3
Labels of Different Types of Nodes in API Context Graphs

Node Type	Label	Example
Vari. Decl.	[Full Class Name].Declaration	`String str;` → java.lang.String.Declaration
Vari. Decl. with Constant Assignment	[Full Class Name].Constant	`String str = "str";` → java.lang.String.Constant
Vari. Decl. with Null Assignment	[Full Class Name].Null	`String str = null;` → java.lang.String.Null
Vari. Decl. with Object Creation	[Full Class Name].new([Parameter Types])	`File file = new File(path);` → java.io.File.new(java.lang.String)
API Method Call	[Full Method Name]([Parameter Types])	`builder.append("str");` → java.lang.StringBuilder.append(java.lang.String)
API Field Access	[Full Field Name]	`System.out;` → java.lang.System.out
Cascading API Method Call (API Field Access)	[Full Method Name]([Parameter Types]) .[Method Name]([Parameter Types])	`builder.append("str").toString();` → java.lang.StringBuilder.append(java.lang.String).toString()
	[Full Field Name] .[Method Name]([Parameter Types])	`System.out.println("str");` → java.lang.System.out.println(java.lang.String)
Nested API Method Call (API Field Access)	[Full Method Name]([Parameter Types]) [Full Method Name]([Parameter Types])	`writer.write(sb.toString());` → java.lang.StringBuilder.toString() java.io.FileWriter.write(java.lang.String)
	[Full Field Name] [Full Method Name]([Parameter Types])	`label.setForeground(Color.blue);` → java.awt.Color.blue javax.swing.JLabel.setForeground(java.awt.Color)
Control Unit	[Control Unit Name]	`if` → If

TABLE 4
API Context Graph Nodes and Edges for Control Statements

Control Statement Type	Node of Control Unit	Nodes and Edges
if statement	**If**	a **Condition** node and a Type c edge from **If** node to **Condition** node
		a **Then** node and a Type c edge from **If** node to **Then** node
		a **ElseIf/Else** node and a Type c edge from **If** node to **ElseIf/Else** node
while/do *for/foreach* statement	**While/DoWhile** **For/Foreach**	a **Condition** node and a Type c edge from **While/DoWhile/For/Foreach** node to **Condition** node
		a **Body** node and a Type c edge from **While/DoWhile/For/Foreach** node to **Body** node
switch statement	**Switch**	a **Selector** node and a Type c edge from **Switch** node to **Selector** node
		a series of **Case** nodes and Type c edges from **Switch** node to each **Case** node
		a **Default** node and a Type c edge from **Switch** node to **Default** node
try statement	**Try**	a series of **Catch** nodes and a Type c edge from **Try** node to the first **Catch** node
		Type c edges connecting **Catch** nodes in order (such as from first to second, from second to third)
		a **Finally** node and a Type c edge from the last **Catch** node to **Finally** node

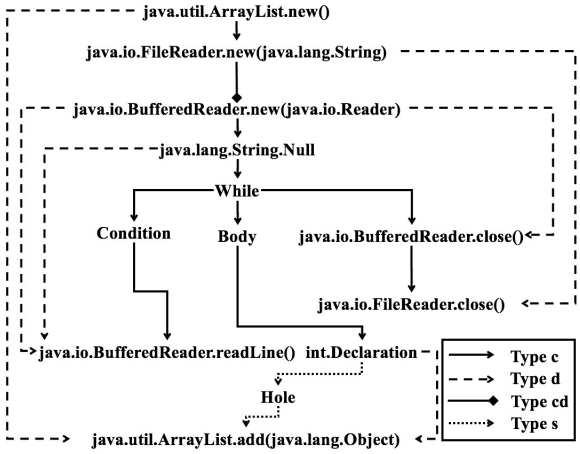

Fig. 5. API Context Graph of the Source Code in Fig. 1

addition, although the program contains a hole, our API context graph is still a connected graph that contains all related structure information, but in GraLan, a graph is not a connected graph but consists of several context subgraphs around the hole.

Indeed, our approach can involve multiple libraries. As long as import needed libraries, our approach can build the API context graph containing API calls from multiple libraries (e.g., APIs in the imported libraries).

The bag of code tokens consists of tokens of the method name, parameter names and variable names. As mentioned before, it captures the textual code information, which is useful for API recommendation. The bag of code tokens is collected as follows. First, APIRec-CST systematically extracts the method name, parameter names and variable names based on the AST of the program. Note that APIRec-CST only extracts the names of parameters and variables whose types are included in the target library (e.g., JDK). Second, because developers often use compound or nonstandard word as names, the extracted names are split as tokens.

APIRec-CST adopts a simple and efficient rule-based method for splitting names into atomic tokens. First, the numbers in a name are pruned. For example, "file2" becomes "file" afterwards. When a number is found inside in a name which is used for composing tokens, the name is split into multiple tokens according to the position of the number in the name. For example, "int2string" is split into "int" and "string". Second, the name is split into multiple tokens using the two special characters "_" and "$" that are often used in naming. For example, "file_name" is split into "file" and "name". Third, each token is further split according to camel case [15]. For example, "fileName" is split into "file" and "name". Next, each token is processed by lemmatization. For example, "files" is converted to "file". Lastly, APIRec-CST post-processes the tokens by removing duplicated tokens as well as tokens which are meaningless, e.g., one character such as "i" and "j". In general, only those tokens which are in the GloVe vocabulary [16] are deemed meaningful. The GloVe vocabulary contains 400K

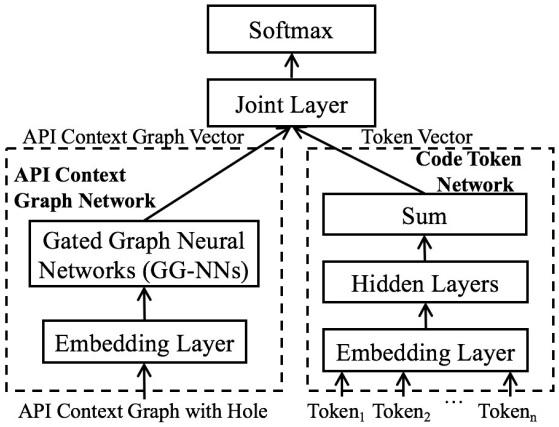

Fig. 6. The Overall Architecture of APIRec-CST

most frequent unique tokens obtained from Wikipedia and Gigaword. The GloVe vocabulary is indeed used to determine meaningful tokens in our approach. In fact, it allows us to filter out 16,175 tokens, which include examples such as "lfsra" (not a word or common abbreviation), "inteface" (typo), "settting" (typo) and "nodesize" (violating the rule of camel case).

For instance, the bag of code tokens obtained from the program shown in Fig. 1 includes "compute", "hash", "code", "path", "result", "rd", "br" and "str".

4.2 Architecture

Given a program with a hole represented in the form of an API context graph and a bag of code tokens, the task of APIRec-CST is to predict what should be for filling the hole. APIRec-CST is designed to solve the task based on deep learning techniques. Fig. 6 shows the overall architecture of APIRec-CST, which consists of two main components i.e., the API Context Graph Network and the Code Token Network, as well as a joint layer. The API Context Graph Network learns an API context graph vector based on a given API context graph. It consists of an embedding layer and GG-NNs. The Code Token Network learns a token vector based on a given bag of code tokens. It consists of an embedding layer, multiple hidden layers and a sum operation. The joint layer is designed to combine the API context graph vector and token vector and output a joint vector. The softmax function is then used to compute the probabilities of each candidate APIs based on the joint vector. We introduce each component and the joint layer in the following.

API Context Graph Network The API Context Graph Network takes as input an API context graph (with a hole to be filled) and outputs a vector. The API context graph is processed as a set of nodes and edges and fed into the network. An embedding layer is first used to embed the node label of each node into an individual vector which is then used as the initial vector of the node annotation in GG-NNs. The embedding layer maintains an embedding matrix to map the label of each node into an individual vector and the embedding matrix is updated through training. The label of each node is considered as a single token instead of being split into several tokens. Then the nodes and edges are passed into GG-NNs to get an API context graph vector.

In order to get the API context graph vector, GG-NNs first compute the state of each node and the state from the last time step is used as the node representation. The overall process of computing the state of each node is introduced in Section 3 and the details can be referred to [10], [11]. Afterwards the API context graph vector is computed based on the node representations with a soft attention mechanism to decide which nodes are relevant to the current API context graph. The detailed equation of computing the API context graph vector can be found in Section 3 and [10].

Code Token Network The Code Token Network takes as input the bag of code tokens and outputs a vector. To obtain the output token vector, an embedding layer is first used to embed each of the code tokens into an individual vector. Subsequently, the information of each token is encoded in the form of a vector which can be learnt during training optimized by trainable parameters. We consider the code tokens as a bag of words, because we need to avoid the influence of ordering among them. First, developers often use compound words for naming method names, variable names, and parameter names. For example, a method name is "readFile" and another method name is "fileRead". These two method names are split as "read file" and "file read" separately. If the order among tokens is not ignored and apply a sequence neural network (such as LSTM) for learning, the vector representations of "read file" and "file read" are not the same. However, in fact, "read file" and "file read" reflect the same semantics and the vector representations of "read file" and "file read" should be the same. Thus, we think it is better to ignore the order among tokens and treat code tokens as a bag of words. Second, we take all tokens into consideration, aiming to capture the semantics of each token and then integrate them together. It means that as long as a token exists, we can capture its semantics no matter its order among all tokens. Take the code snippet in Fig. 1 as an example. Suppose that the method name "computeHashCode" is changed to "compute", then "hash" and "code" will appear after "str" instead of before "str". However, no matter "hash" and "code" appear after "str" or before "str", the semantics of hash code in fact exists in all the tokens. Thus, we think it is better to ignore the order among tokens and treat code tokens as a bag of words. We use multiple fully connected layers as hidden layers to capture higher-level semantic information among the code tokens. Then we sum all the vector representation of each token output by the last hidden layer as the final embedding of all the code tokens named token vector.

Joint Layer The joint layer takes as input the API context graph vector and the token vector and outputs a joint vector. Suppose that the API context graph vector is a d^A-dimensional vector and the token vector is a d^T-dimensional vector. The joint layer first combines the d^A-dimensional vector and the d^T-dimensional vector as a d^{A+T}-dimensional concat vector. Then the d^{A+T}-dimensional concat vector is used to compute the final joint

Algorithm 1 Training Instance Construction

Input: $API\ context\ graph\ without\ a\ hole,\ node,\ hole\_size,\ code\_tokens$
Output: $API\ context\ graph\ with\ a\ hole,\ remaining\ code\ tokens,\ ground\ truth$
1: $count = 0, curr = node$
2: **while** $count$ is less than $hole\_size$ and $curr$ is not $Null$ **do**
3: let old be $curr$
4: **if** $curr$ is **If**, **While**, **Do**, **For**, **Foreach**, **Switch**, or **Try then**
5: **for** each $child$ with edge type of $Type\ c$ or $Type\ cd$ of $curr$ **do**
6: **if** $child$ represents the statement outside the control scope **then**
7: $count = count + 1$
8: $curr = child$
9: **else**
10: remove $child$ and its subgraph
11: remove all edges connected to $child$ and nodes in its subgraph
12: **end if**
13: **end for**
14: **else**
15: set $curr$ to be child with edge type of $Type\ c$ or $Type\ cd$ of $curr$
16: $count = count + 1$
17: **end if**
18: remove old and all edges connected to old
19: **end while**
20: replace $node$ with **Hole**
21: get remaining code tokens related to the $API\ context\ graph\ with\ a\ hole$
22: set $groundtruth$ to be the label of $node$

vector through a fully connected layer using *tanh* as the activation function. The fully connected layer is designed to further learn the joint semantics of the structural code information (in the form of the API context graph vector) and textual code information (in the form of the token vector) in a holistic way. The joint vector output by the joint layer is used as the final vector for the softmax function.

Softmax Function In deep neural networks, the softmax function is usually used to map a vector to a normalized probability distribution over fixed size classes that needed to be predicted. The classes are then ranked based on their probabilities. If we consider each API as a class, the API recommendation task can be considered as a classification task. What we need to do is to compute the probability of each API and then get the top N APIs as the recommendations. Thus, the softmax function is a natural choice. It takes as input the joint vector, and outputs a normalized probability over all APIs.

4.3 Training Corpus Construction

To train the models in APIRec-CST, we require a large set of training instances. A training instance is a triple consisting of an API context graph (with a hole), a corresponding bag of code tokens and ground truth (the expected label of the hole node (i.e., an API call)). To construct training instances, we first collect a large code base and then parse the methods one by one. For each method, we construct its corresponding API context graph (without a hole) and obtain the bag of code tokens. Afterwards, we systematically replace a set of nodes from the API context graph with a hole node. The resultant API context graph (with a hole), the remaining code tokens and the ground truth (which is the label of the first removed node) form a training instance.

The details of the algorithm for constructing a training instance is shown in Algorithm 1. The inputs are an API context graph without a hole, the corresponding bag of code tokens, a node $node$ in the graph and a constant $hole\_size$. Intuitively, $node$ is the starting node to be removed and the label of $node$ is used as the ground truth for the training instance, and $hole\_size$ is the number of nodes to be removed in the hole. Specifically, when a node represents a control unit (e.g, if or while), the node itself and all its child nodes in the condition subgraph and all its child nodes in the body subgraph are considered as a whole, which means that though there are several nodes, the number is considered as 1. We treat them as a whole to guarantee the syntax completeness of the source code. Other types of nodes are considered as single nodes.

Algorithm 1 uses a variable $count$ to count the number of nodes that have been removed. Whenever $count$ reaches $hole\_size$ or there are no more nodes to be removed, the algorithm terminates. Initially, we set $curr$ (which is the current node to be removed) to be $node$. If $curr$ is not a control node (like *if* or *while*), we identify its (unique) child node $child$ through an edge of *Type c* or *Type cd*, remove the current node $curr$ from the graph and set $curr$ to be $child$. Note that whenever a node is removed, so are its incoming and outgoing edges. The reason why we choose the child node following edges of *Type c* or *Type cd* is that we remove nodes according to the control flow in the source code. As a result, the remaining context graph is still well-formed from a control flow point of view. If $curr$ is a control node, all of its subgraphs in its control scope are removed, i.e., we remove all its subsequent nodes through control flow representing a statement in the control scope (e.g., all statements in the loop body if $curr$ is a *while* node). For instance, if we remove the control node labeled with $While$ in Fig. 1, all nodes representing the API call at line 6/7/8/9 are also removed, which are the ones labeled with $Condition$, $java.io.BufferedReader.readLine()$, $Body$, $int.Declaration$, $java.lang.String.hashCode()$ and $java.util.ArrayList.add(java.lang.Object)$. Then, we set $curr$ to be the first subsequent node outside of the control scope. Note that API calls in the condition and body of a control unit are defined in the control scope (such as $java.io.BufferedReader.readLine()$ and $java.util.ArrayList.add(java.lang.Object)$ in Fig. 1), while the first API call that does not belong to the condition and body of a control unit is defined outside of the control scope (such as $java.io.BufferedReader.close()$ in Fig. 1). Thus, for a control unit, there is only one child node outside of the control scope. When we remove a control unit through Algorithm 1 at line 5-13, the child node outside of the control scope of this control unit is always the last to be visited (nodes in the condition and body will be first visited), so $curr = child$ will only be executed one time at last.

For example, Fig. 5 is an API context graph with a hole that is produced from the code in Fig. 1. In this example, the input of node $node$ is the node with label $java.lang.String.hashCode()$ representing the statement of $hashCode = str.hashCode()$; at line 8. The input of hole size $hole\_size$ is set to be 1. The input of code tokens $code\_tokens$ are all the tokens extracted in the original complete code. The remaining code tokens are those tokens in the remaining source code, which are "compute", "hash", "code", "path", "result", "rd", "br" and "str". For another instance, if all but line 2 and 3 are removed in Fig. 1, the remaining code tokens become "compute", "hash", "code", "path", "result", and "rd". The ground truth of this train-

ing instance is $java.lang.String.hashCode()$. We further illustrate an example whose hole size is more than 1 in Fig. 7 to explain Algorithm 1. In this example, the first API context graph is the input API context graph without a hole, the input of node $node$ is the node with label $While$, the input of hole size $hole\_size$ is 2 and the input of code tokens $code\_tokens$ are "compute", "hash", "code", "path", "result", "rd", "br" and "str". At the beginning, $count$ is 0, $curr$ is the node with label $While$, and old is $curr$. Since $curr$ is a control unit, all its child nodes (except the child node outside of the control scope) and corresponding subgraphs are removed (i.e., its condition node and condition subgraph, and its body node and body subgraph). All edges that connected with the removed nodes are also removed. Thus, we get the second API context graph in Fig. 7. At this time, though we have removed several nodes, they all belong to the same control unit, so $count$ is added from 0 to 1. At the same time, $curr$ is the node with label $java.io.BufferedReader.close()$ which is the child node with Type c of the node with label $While$. Since $count = 1$ is less than hole size 2 and $curr$ is not null, next we need to continue to remove nodes. When continue to remove nodes, since $curr$ is the node with label $java.io.BufferedReader.close()$, which is not a control unit, we set $curr$ be its child node with Type c (which is the node with label $java.io.FileReader.close()$) and then we directly remove the node with label $java.io.BufferedReader.close()$ (which is stored as old) and all edges connected with it. Thus, we get the third API context graph in Fig. 7. At this time, $count$ is added from 1 to 2, which is not less than hole size 2, so we stop removing nodes. Finally, we add a hole node at the position of the input node $node$. Thus, we can get the fourth API context graph with a hole in Fig. 7 as output. At the same time, the remaining code tokens are "compute", "hash", "code", "path", "result", "rd", "br" and "str", and the ground truth is the label of the input node $node$, which is $While$.

To systematically construct a set of training instances, for each API context graph and code tokens constructed from a method in the code base, the above algorithm is applied with each node in the graph as the starting node to be removed and different hole sizes. Note that the hole size can range from 1 to $Max-1$ where Max is the total number of nodes in the API context graph. Though the hole size can range from 1 to Max-1 where Max is the total number of nodes in the API context graph, if we do not limit the hole size, Algorithm 1 will be slow and may suffer from the data explosion problem. Thus, as in [17], we limit the max hole size to 5 in our experiments.

5 Evaluation

The purpose of APIRec-CST is recommending APIs based on given code context by combining structural and textual code information. We develop an implementation of APIRec-CST for JDK 1.8, which has 17,173 API classes and 137,134 API methods/fields. The implementation uses JavaParser [18] to parse source code into ASTs and Java reflection mechanism to recognize API invocations in source code. We remark that our current implementation supports nearly 70% of the different types of statements that JavaParser provide, which cover almost all fundamental computational program logics. The fundamental computational logics of a program mainly consist of expression statements as well as control statements. For expression statements, we cover all the most commonly used statements, such as variable declaration (including object creation), assignment, literal expression, method call expression and field access expression. For control statements, we cover all types of control units (i.e., if-elseif-else, while, for, foreach, doWhile, try, and switch). Statements that do not affect the fundamental computational logics of a program are ignored as of now, such as assert statement and throw statement. In our future work, we will expand our approach to cover more statement types. The lemmatization of code tokens is implemented using Stanford CoreNLP [19]. The deep learning architecture is implemented using TensorFlow 1.14 [20] and GG-NNs reference implementation [21]. Based on the implementation, we conduct a series of experimental studies to answer the following research questions.

RQ1 (API Prediction Accuracy): How accurate is APIRec-CST in predicting the next API compared with state-of-the-art approaches for context-based API recommendation?

RQ2 (Contribution of Textual Code Information): How much does textual code information contribute to the API recommendation?

RQ3 (Sensitivity Analysis): How does hole size (i.e., number of APIs to be recommended in a hole) affect the accuracy of APIRec-CST?

RQ4 (Effectiveness in Real Tasks): How effective are developers when accomplishing programming tasks using APIRec-CST ?

All the data of the experimental studies can be found in our replication package [22].

5.1 Training Details

We create a large corpus from GitHub by crawling all the Java projects that have 1000 stars or more. In this way we obtain 1,914 Java projects, which include 944,783 source files, 7,279,321 methods, and 68,319,916 lines of code.

We randomly select 90% of the Java projects as training set and the remaining 10% projects as validation set. For methods in the files of each project in the training or validation set, we apply Algorithm 1 to create a set of training instances or validation instances. To ensure efficiency we filter out the files that are larger than 200 KB and the methods that have no JDK API invocations. The reason for filtering files that are larger than 200 KB is that parsing large files using JavaParser [18] is quite time consuming. Large files may halt JavaParser when parsing these files due to its limitation in scalability. That is why they are removed in our approach. Note that most of files (i.e, 99.9993% of them) have a size smaller than 200KB and we expect filtering those large files has minimum effect. The reason for filtering methods that have no JDK API invocations is that we focus on JDK library. When creating training/validation instances containing only preceding context, we do not limit the hole size (i.e., $hole\_size$); when creating training/validation instances containing both preceding and succeeding contexts, we limit the hole size to 5 or less to avoid data explosion.

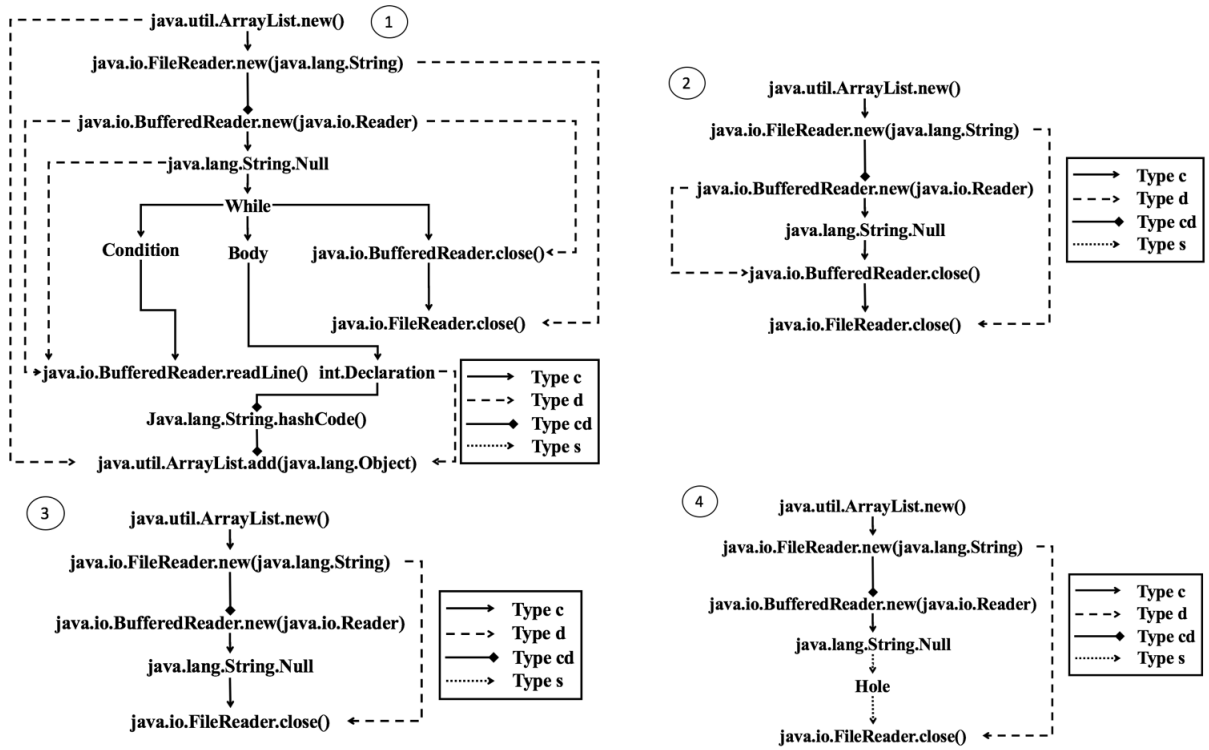

Fig. 7. Example of API Context Graphs during Each Step in Algorithm 1

We also filter out training/validation instances that have no API invocation in the context. Finally we obtain 6,627,591 training instances and 482,186 validation instances.

Based on the training data and validation data, we train an API recommendation model using a server with Intel Xeon E5-2620 2.1GHz (16 threads and 128GB RAM) and two Nvidia 1080Ti GPUs running on Ubuntu 16.04. We set **embedding size** of each embedding layer to 300, the number of hidden layers to 3, **hidden size** of each hidden layer to 300, **dropout** to 0.75, **learning rate** to 0.005, and **batch size** to 256. We conduct serval trial experiments with different hyper parameters and the above hyper parameters achieve the best performance. After each epoch in the training, APIRec-CST evaluates the current model using the validation instances. If the prediction accuracy does not increase in five successive epochs, the training process ends and the last best model is used as the result.

5.2 API Prediction Accuracy (RQ1)

We compare APIRec-CST with existing approaches for solving the same problem. We adopt two approaches that are most related to ours as baseline approaches in this evaluation. One is GraLan [8], which is a state-of-the-art graph-based statistical model for API recommendation and the other is Tree-LSTM [17], which is a state-of-the-art deep learning model using tree-based structure for API recommendation. We reimplement GraLan based on the description of the approach in [8] and the extraction of graph representation from code in [23]. We set the parameter value for GraLan's theta to 4 and we set the parameter value for GraLan's delta to 4. The reason for setting theta to 4 is that according to Nguyen et al. [8], when theta is 4, the top-10 accuracy achieved by GraLan (86.3%) is close to the best accuracy (87.1%). The reason for setting delta to 4 is that according to Nguyen et al. [8], when delta is 4, the top-10 accuracy achieved by GraLan (85.8%) is close to the best accuracy (87.1%). Considering the performance of predicting time, we set theta to 4 and delta to 4. The implementation of Tree-LSTM is directly obtained from the authors of [17]. APIRec-CST, GraLan and Tree-LSTM are trained with the same training data. We choose six open-source Java projects as the test data: Galaxy [24], Log4j [25], JGit [26], Froyo-Email [27], Grid-Sphere [28], and Itext [29]. These projects are chosen based on the following criteria: widely used as test data in previous researches on API recommendation (e.g., [9], [30]); not included in the training data or validation data. Following the same procedure of training/validation instance construction (which means that test instances contain the situations of just preceding context and both preceding and succeeding context with different hole sizes), we create 14,986 test instances from the test data.

To confirm the effect of our GraLan implementation, we compare the API recommendation accuracy of our implementation on the six projects with that of the GraLan implementation by Liu et al. [9] based on the results they report in [9]. We use the top-k accuracy, which is the ratio of the number of test instances whose top-k recommendations include the expected API, as a metric. To compute the top-k accuracy, we first need to obtain top-k API recommendations from GraLan. As mentioned in Section 2, in GraLan, each subgraph is considered as a context parent graph to generate child graphs (each child graph has one more node than its parent graph and the extra node is considered as a candidate API recommendation). So, given a source code

with a hole, GraLan first extracts subgraphs around the hole as context parent graphs and then uses these context parent graphs to generate child graphs ordered by probabilities. Finally, the API of the one more extra node of each child graph compared with its corresponding parent graph is used as the recommendation. All the API recommendations are ordered by their probabilities. Thus, we can obtain top-k API recommendations to compute the top-k accuracy. The comparison shows that: our implementation achieves a top-1 (top-10) accuracy of 19.6-41.6% (73.4-80.9%), while their implementation achieves a top-1 (top-10) accuracy of 22.4-33.6% (73.9-80.6%); in terms of top-1 accuracy, our implementation is better than theirs on 4 projects and worse than theirs on 2 projects; in terms of top-10 accuracy, our implementation is better than theirs on 4 projects and worse than theirs on 2 projects. The results show that the performance of these two implementations is comparable. Note that the performance of GraLan is sensitive to the count of each subgraph appeared in the training data. Our training data is different from the training data used in [9], which explains why the performance of our implementation of GraLan is different from their original.

We compare the top-K accuracies and MRR (Mean Reciprocal Rank) of APIRec-CST, GraLan and Tree-LSTM for predicting the next API. MRR is a summary metric for top-K accuracies that averages the inverse of the ranks of each recommendation, which ranges from 0 to 1 [31]. For example, a MRR of 0.25 means that the correct recommendation is to appear at the fourth position on average. The results are shown in Table 5. In the table, the number of test instances of each project is shown after the project name and the best accuracy and MRR values are in boldface. We can see that APIRec-CST achieves much higher top-1, top-5, and top-10 accuracy than GraLan and Tree-LSTM. For the six projects, APIRec-CST's top-1, top-5, and top-10 accuracy is 50.6-66.4% (58.6% on average), 67.7-87.1% (81.4% on average), and 79.2-92.5% (87.9% on average), respectively; GraLan's top-1, top-5, and top-10 accuracy is 19.6-41.6% (31.5% on average), 60.5-71.4% (64.5% on average), and 73.4-80.9% (77.6% on average), respectively; Tree-LSTM's top-1, top-5, and top-10 accuracy is 39.3-52.6% (46.7% on average), 62.9-75.6% (70.4% on average), and 75.6-82.6% (79.3% on average), respectively. We can see that APIRec-CST also achieves much higher MRR than GraLan and Tree-LSTM. APIRec-CST's MRR is 58.4-74.2% (68.4% on average), GraLan's MRR is 37.4-53.8% (45.3% on average) and Tree-LSTM's MRR is 51.4-61.7% (56.7% on average). Furthermore, we conduct Mann-Whitney U test to determine whether the improvements in top-1, top-5, top-10 accuracy and MRR between APIRec-CST and the other two approaches are statistically significant. If the p-value is less than 0.05, the improvement is considered to be significant. The p-value of top-1, top-5 and top-10 accuracy between APIRec-CST and GraLan are 0.003, 0.004 and 0.004 respectively. The p-value of top-1, top-5 and top-10 accuracy between APIRec-CST and Tree-LSTM are 0.015, 0.023 and 0.010 respectively. The p-value of MRR between APIRec-CST and GraLan is 0.003 and the p-value of MRR between APIRec-CST and Tree-LSTM is 0.007. We can see that all the improvements are significant.

Since source code can only contain preceding context or contain both preceding and succeeding context, we further separately evaluate the top-k accuracies and MRR of GraLan, Tree-LSTM and APIRec-CST when source code only contains preceding context and contains both preceding and succeeding context. The results are shown in Table 6 and Table 7 separately. We can see that no matter source code contains only preceding context or contains both preceding and succeeding context, APIRec-CST achieves the best top-k accuracies and MRR.

We further explain the reasons why APIRec-CST achieves the best performance. First, APIRec-CST is based on the hypothesis that we should leverage and combine the information from different aspects in the source code as much as possible, which treats the API usage in the source code as a whole and combines structural and textual code information. In this way, APIRec-CST can obtain more complete, diverse and overall information in the source code that is related to the API needed to be predicted from our API context graph and code tokens. Second, APIRec-CST is based on the hypothesis that deep learning models are stronger than traditional models in terms of the ability of feature representation, extraction and learning. Deep learning models are implicit models that represent, extract and learn features in an implicit way. Features are represented, extracted and learnt as vectors through parameters during training automatically. Thus, there is no need for deep learning models to define and extract features explicitly and manually to avoid missing something important or just suitable for some cases. For example, the subgraphs used in GraLan can be seen as manually defined and extracted features, which miss the holistic view of structural code information and can not identify the important information in the subgraphs. Benefiting from applying GG-NNs, APIRec-CST can automatically identify the important information in the structural code information through training. Meanwhile, the Code Token Network uses several hidden layers to abstract the high-level semantics in the textual code information and then used to jointed with GG-NNs. All the features are implicit vector representations in a large feature vector space. Through carefully training, these features can be trained as well as possible. Thus, APIRec-CST can better characterize and learn features.

In our work, we use the top-k accuracy and MRR for the evaluation. These two metrics are widely used in the API recommendation task in the previous works, such as [3], [4], [7], [8], [9], [17], [30]. We agree that there is a gap from these two metrics to the usefulness in practice, i.e. whether an improved accuracy translates to usefulness improvement of the recommendation in practice and by how much.

According to [32], [33], it is not always the case that the more the choices, the better the results. That is, developers may be overwhelmed by a recommendation list containing too many API recommendations. As top-k accuracy (usually k is set to 10 as the max value) is widely used in previous works, we assume that developers prefer to look up the correct API recommendation within 10 recommendations in practice. Therefore, the improvement of top-10 accuracy can help developers find their desired APIs in more cases when they are not willing to look up their desired APIs out of 10 recommendations. We conduct an experiment to count the number of cases where APIRec-CST recommends the correct

TABLE 5
API Recommendation's Top-K Accuracy and MRR (%)

Project	Model	Top-1	Top-5	Top-10	MRR
Galaxy (473)	GraLan	29.4	60.5	73.4	42.2
	Tree-LSTM	39.3	68.7	76.7	51.4
	APIRec-CST	**51.0**	**81.6**	**88.2**	**63.6**
JGit (4530)	GraLan	41.6	71.4	79.1	53.8
	Tree-LSTM	52.6	75.6	81.2	61.7
	APIRec-CST	**66.4**	**85.1**	**89.5**	**74.2**
Froyo-Email (1537)	GraLan	23.0	62.8	78.9	40.7
	Tree-LSTM	51.6	74.5	82.6	61.1
	APIRec-CST	**63.7**	**86.0**	**91.3**	**73.5**
Grid-Sphere (1847)	GraLan	36.5	66.5	80.9	48.6
	Tree-LSTM	48.0	72.4	80.6	58.3
	APIRec-CST	**62.0**	**87.1**	**92.5**	**72.8**
Itext (4444)	GraLan	19.6	64.3	75.7	37.4
	Tree-LSTM	46.0	68.1	75.6	55.4
	APIRec-CST	**57.9**	**80.7**	**86.8**	**67.6**
Log4j (2155)	GraLan	38.6	61.3	77.5	48.9
	Tree-LSTM	42.4	62.9	79.0	52.2
	APIRec-CST	**50.6**	**67.7**	**79.2**	**58.4**
Overall	GraLan	31.7	66.0	77.9	45.9
	Tree-LSTM	48.1	70.8	79.1	57.7
	APIRec-CST	**60.3**	**81.5**	**87.7**	**69.4**

TABLE 6
API Recommendation's Top-K Accuracy and MRR: with Preceding Context Only (%)

Project	Model	Top-1	Top-5	Top-10	MRR
Galaxy (239)	GraLan	24.3	60.7	72.8	39.5
	Tree-LSTM	34.7	67.4	75.3	48.0
	APIRec-CST	**50.2**	**81.6**	**87.9**	**62.9**
JGit (2399)	GraLan	38.6	69.1	77.9	50.7
	Tree-LSTM	44.3	72.3	77.4	55.6
	APIRec-CST	**60.2**	**82.1**	**87.1**	**69.5**
Froyo-Email (824)	GraLan	22.7	60.4	77.8	38.9
	Tree-LSTM	43.9	70.0	79.2	54.9
	APIRec-CST	**58.9**	**82.9**	**89.8**	**69.4**
Grid-Sphere (1167)	GraLan	33.6	65.6	79.9	46.1
	Tree-LSTM	47.2	73.1	80.0	58.0
	APIRec-CST	**59.8**	**87.3**	**92.6**	**71.5**
Itext (2511)	GraLan	17.7	61.4	73.7	35.2
	Tree-LSTM	42.4	65.9	73.0	52.2
	APIRec-CST	**51.6**	**77.5**	**84.3**	**62.6**
Log4j (887)	GraLan	26.9	59.0	73.2	40.3
	Tree-LSTM	35.4	59.0	71.9	45.9
	APIRec-CST	**44.3**	**67.6**	**78.5**	**54.4**
Overall	GraLan	28.0	63.9	76.2	42.5
	Tree-LSTM	42.8	68.6	75.9	53.5
	APIRec-CST	**55.2**	**79.9**	**86.4**	**65.7**

TABLE 7
API Recommendation's Top-K Accuracy and MRR: with Both Preceding and Succeeding Context (%)

Project	Model	Top-1	Top-5	Top-10	MRR
Galaxy (234)	GraLan	34.6	60.3	73.9	45.0
	Tree-LSTM	44.0	70.1	78.2	54.9
	APIRec-CST	**51.7**	**81.6**	**88.5**	**64.2**
JGit (2131)	GraLan	45.1	74.0	80.3	57.3
	Tree-LSTM	62.0	79.3	85.5	68.6
	APIRec-CST	**73.3**	**88.5**	**92.1**	**79.6**
Froyo-Email (713)	GraLan	23.3	65.6	80.2	42.8
	Tree-LSTM	60.4	79.7	86.5	68.2
	APIRec-CST	**69.3**	**89.6**	**93.0**	**78.2**
Grid-Sphere (680)	GraLan	41.6	68.2	82.8	52.8
	Tree-LSTM	49.4	71.3	81.5	58.8
	APIRec-CST	**65.9**	**86.6**	**92.2**	**75.1**
Itext (1933)	GraLan	22.2	68.0	78.3	40.3
	Tree-LSTM	50.6	71.0	78.9	59.6
	APIRec-CST	**66.2**	**84.9**	**90.0**	**74.2**
Log4j (1268)	GraLan	46.8	63.0	80.5	54.9
	Tree-LSTM	47.3	65.7	**84.0**	56.6
	APIRec-CST	**55.0**	**67.7**	79.7	**61.2**
Overall	GraLan	36.1	68.5	79.8	49.8
	Tree-LSTM	54.2	73.4	82.9	62.4
	APIRec-CST	**66.2**	**83.4**	**89.2**	**73.7**

API in top-10 recommendations, whereas GraLan and Tree-LSTM fail to do so. The total number is 682 (about 4.6% of all the 14,986 cases), which means that developers would fail to find the correct implementation even after examining 10 recommendation using existing approaches. As a result, the improvement of top-k accuracy can indeed translate in usefulness improvement in practice.

In addition, the usefulness of the recommendation generated by APIRec-CST is partially evidenced through the user study with the master students on real-world programming tasks as we reported in Section 5.5. How to further evaluate the usefulness of the recommendations in practice will be our future direction.

To sum up, APIRec-CST can achieve high top-k accuracies and MRR and performs much better than GraLan and Tree-LSTM. Since APIRec-CST represents correlated API usage in control and data flow graph of an entire method from a holistic view and introduces textual code information. APIRec-CST can leverage semantics in both preceding and succeeding context. Thus, no matter the source code only contains preceding context or contains both preceding and succeeding context, APIRec-CST can treat the context in the source code as a whole and leverage all the information in the context.

5.3 Contribution of Textual Code Information (RQ2)

APIRec-CST mainly relies on the structural code information embedded in the API Context Graph Network and at the same time leverages the textual code information embedded in the Code Token Network. Though textual information contains textual semantics that reflect the intent of a developer, textual information can not reflect usages among of APIs. For the API recommendation task, the usages and semantics of structural information (e.g., API usage) in the source code plays a leading role to predict the API at a hole, because the API at a hole is closely connected with the APIs in the context. Thus we assume that the main contribution is made by structural information. To verify our assumption, we further train a variant model based on the same training/validation data that only uses textual information called APIRec-TO, which only includes one network (i.e., Code Token Network). We evaluate APIRec-TO on the same test data. To evaluate the contribution of textual code information when combined with structural code information, we derive a variant of APIRec-CST that uses structural code information only (called APIRec-SO), which only includes one network (i.e., API Context Graph Network). We use APIRec-SO to train an API recommendation model based on the same training/validation data and evaluate the model with the same test data.

The results are shown in Table 8. In Table 8, the left value in the Difference column is the difference between APIRec-CST and APIRec-TO and the right value is the difference between APIRec-CST and APIRec-SO. We can see that the top-1, top-5 and top-10 accuracies as well as MRR of APIRec-TO are quite lower than APIRec-SO and APIRec-CST. The results indicate that structural code information plays the leading role for API recommendation. We can also see that APIRec-SO achieves good top-1, top-5, and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3074309, IEEE Transactions on Software Engineering

TABLE 8
Contribution of Textual Code Information(%)

Project	Model	Top-1	Difference	Top-5	Difference	Top-10	Difference	MRR	Difference
Galaxy (473)	APIRec-TO	19.2		51.6		64.7		32.2	
	APIRec-SO	46.9	+31.8/+4.1	76.3	+30.0/+5.3	82.2	+23.5/+6.0	58.9	+31.4/+4.7
	APIRec-CST	**51.0**		**81.6**		**88.2**		**63.6**	
JGit (4530)	APIRec-TO	15.4		40.9		51.3		24.8	
	APIRec-SO	61.7	+51.0/+4.7	83.8	+44.2/+1.3	88.6	+38.2/+0.9	71.2	+49.4/+3.0
	APIRec-CST	**66.4**		**85.1**		**89.5**		**74.2**	
Froyo-Email (1537)	APIRec-TO	18.9		43.7		60.9		29.4	
	APIRec-SO	58.8	+44.8/+4.9	82.4	+42.3/+3.6	88.9	+30.4/+2.4	68.7	+44.1/+4.8
	APIRec-CST	**63.7**		**86.0**		**91.3**		**73.5**	
Grid-Sphere (1847)	APIRec-TO	13.2		42.2		58.4		24.7	
	APIRec-SO	57.7	+48.8/+4.3	83.1	+44.9/+4.0	90.6	+34.1/+1.9	69.0	+48.1/+3.8
	APIRec-CST	**62.0**		**87.1**		**92.5**		**72.8**	
Itext (4444)	APIRec-TO	13.1		39.3		51.1		23.1	
	APIRec-SO	56.3	+44.8/+1.6	78.8	+41.4/+1.9	84.4	+35.7/+2.4	65.7	+44.5/+1.9
	APIRec-CST	**57.9**		**80.7**		**86.8**		**67.6**	
Log4j (2155)	APIRec-TO	12.0		27.3		39.8		19.1	
	APIRec-SO	48.5	+38.6/+2.1	**71.4**	+40.4/-3.7	**83.7**	+39.4/-4.5	58.3	+39.3/+0.1
	APIRec-CST	**50.6**		67.7		79.2		**58.4**	
Overall (14986)	APIRec-TO	14.4		39.3		51.9		24.2	
	APIRec-SO	56.9	+45.9/+3.4	80.0	+42.2/+1.5	86.7	+35.8/+1.0	66.8	+45.2/+2.6
	APIRec-CST	**60.3**		**81.5**		**87.7**		**69.4**	

TABLE 9
API Recommendation's Top-K Accuracy and MRR of APIRec-SO and APIRec-CST with the Increase of API Number in the Context(%)

Number of APIs	Model	Top-1	Top-5	Top-10	MRR
1	APIRec-SO	36.2	65.9	72.5	48.5
	APIRec-CST	**41.7**	**69.1**	**75.7**	**53.0**
2	APIRec-SO	53.3	73.4	83.5	62.3
	APIRec-CST	**55.2**	**77.7**	**84.9**	**65.0**
3	APIRec-SO	51.6	78.5	83.6	62.9
	APIRec-CST	**57.2**	**81.2**	**86.9**	**67.5**
4	APIRec-SO	59.1	79.7	86.7	68.3
	APIRec-CST	**62.0**	**84.4**	**88.8**	**71.4**
5	APIRec-SO	59.2	84.9	89.8	70.4
	APIRec-CST	**63.0**	**86.1**	**91.7**	**72.7**
6	APIRec-SO	61.0	82.6	**88.8**	70.1
	APIRec-CST	**62.2**	**83.4**	88.6	**71.3**
7	APIRec-SO	54.6	79.9	86.5	65.3
	APIRec-CST	**59.6**	**82.5**	**89.2**	**69.3**
8	APIRec-SO	**62.4**	81.5	89.4	70.9
	APIRec-CST	61.8	**84.2**	**90.7**	**71.6**
9	APIRec-SO	54.7	80.8	**89.4**	66.0
	APIRec-CST	**58.5**	**81.1**	87.9	**68.1**
10	APIRec-SO	**64.1**	84.4	**90.3**	72.8
	APIRec-CST	63.9	84.4	89.9	**72.9**
11	APIRec-SO	58.6	**84.2**	91.9	69.7
	APIRec-CST	**63.3**	84.0	**92.6**	**72.5**
12	APIRec-SO	60.2	81.5	87.8	68.8
	APIRec-CST	**62.3**	**81.8**	**88.1**	**70.6**
13	APIRec-SO	54.9	81.0	89.0	66.3
	APIRec-CST	**59.2**	**82.5**	**89.6**	**69.5**
14	APIRec-SO	65.3	**85.8**	89.8	73.7
	APIRec-CST	**67.7**	84.2	**90.6**	**74.6**
15	APIRec-SO	59.7	82.5	89.4	70.0
	APIRec-CST	**64.3**	**85.2**	**90.5**	**73.9**
15+	APIRec-SO	65.4	**85.6**	**91.3**	73.7
	APIRec-CST	**69.0**	83.3	89.6	**75.2**

top-10 overall accuracy (56.9%, 80.0%, and 86.7%) on the six projects, but the accuracy is lower than that of APIRec-CST (60.3%, 81.5%, and 87.7%). The top-1 overall accuracy achieves a 3.4% improvement, the top-5 overall accuracy achieves a 1.5% improvement and the top-10 overall accuracy achieves a 1.0% improvement when textual code information is added. For each test project, the top-k accuracy of adding textual code information achieves different degrees of improvement. The improvement of the top-1 accuracy ranges from 1.6% to 4.9%, the improvement of the top-5 accuracy ranges from 1.3% to 5.3%, and the improvement of the top-10 accuracy ranges from 0.9% to 6.0%. We can also see that the overall MRR is improved by 2.6% when textual code information is added. For each test project, the improvement of MRR ranges from 0.1% to 4.8%. The top-5 and top-10 accuracy of Log4j project decrease when textual code information is added. It is because that textual code information maybe contains noise that negatively influences the API recommendation results. In our future work, we will try to better process the noise in textual code information.

TABLE 10
Number of Positive and Negative Test Cases when Adding Textual Information (%)

Project	Postive Test Instances	Negative Test Instances
Galaxy (473)	131 (27.7%)	64 (13.5%)
JGit (4530)	823 (18.2%)	514 (11.3%)
Froyo-Email (1537)	318 (20.7%)	180 (11.7%)
Grid-Sphere (1847)	362 (19.6%)	216 (11.7%)
Itext (4444)	829 (18.7%)	599 (13.5%)
Log4j (2155)	376 (17.4%)	472 (21.9%)

For each test project, we further conduct an experiment to count the number of test instances where APIRec-CST ranks the correct API higher than APIRec-SO (which we call positive test instances) and otherwise (which we call negative test instances) to understand the significance of the improvement by introducing textual information. The numbers and ratios of positive test instances and negative test instances for each project are shown in Table 10. From the table, we can see that the number and ratio of positive test instances of each project (except Log4j) is higher than negative test instances. The improvement (i.e., the ratio of the positive test instances) of each test project is significant, which is about 20.4% on average. The negative test instances are usually caused by the lack of in-depth understanding of the context relevant to the current position. First, the code tokens in the textual information mislead the implementation consideration due to the lack of in-depth understanding of the context. For example, in the first case in Fig. 11 the code tokens "get" and "all" in the method name mislead APIRec-CST to recommend for (iteratively adding all the elements in a loop) and $java.util.ArrayList.addAll$. How-

ever, the developer chooses to add all the levels one by one using $java.util.ArrayList.add$ as they are defined as a set of enumeration values. Second, the code tokens in the textual information are more relevant to the APIs after the current position. For example, in the second case in Fig. 11 the code token "write" in the method name makes APIRec-CST miss the correct API $java.lang.Class.getName$ in the top-5 recommendations. In fact, the recommended APIs related to object writing are the core APIs but will be used later after the current position.

To further understand the contribution of textual code information, we analyze its influence with an increasing number of APIs in the context. We divide all the test data into 16 subsets according to the number of APIs in the context (1-15 and above 15). For each subset, we calculate the difference of the top-1, top-5, top-10 accuracy and MRR of APIRec-CST and APIRec-SO. Note that the difference computed for 15+ is the overall difference for the number of APIs in the context above 15, including 16, 17, 18 and so on. The results are shown in Fig. 8 (the original top-k accuracies and MRR are shown in Table 9). The dotted lines are the zero lines and the points above the lines indicate positive contribution of textual code information, which mean that APIRec-CST achieves higher accuracy and MRR than APIRec-SO. We can see that the contribution of textual code information is positive in most cases. There is no obvious positive or negative correlation between the contribution of textual code information and the number of APIs in the context. This means that the contribution of textual code information is insensitive to the number of APIs in the context. The reason of the nine negative cases in Fig. 8 is also that textual code information maybe contains noise that negatively influences the API recommendation results. Though textual information is useful overall, its quality is difficult to guarantee. That is, it is different in different projects or even in different test instances. It may sometimes bring a negative impact, although not often. Please see the negative examples that we add in Section 5.6 for more details to learn in which situations textual information is noise.

To sum up, textual information indeed makes sense and achieves different levels of improvement for different test projects in terms of top-1 accuracy, top-5 accuracy, top-10 accuracy and MRR. However, textual information may also introduce noises that need to be addressed in our future work.

5.4 Sensitivity Analysis (RQ3)

The test instances (no matter the test instances that contain only preceding context or contain both preceding and succeeding context) in RQ1 are constructed with different hole sizes, which means that the number of APIs to be recommended in a hole is different. Thus, we further conduct an experiment to evaluate how the number of APIs to be recommended in a hole will affect the accuracy of APIRec-CST. For each test project, we vary the number of APIs to be recommended in a hole of test instances in our experiment from 1 to 10+ (i.e., 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and over 10). Therefore, we can obtain 11 subsets of test instances. For each subset of each test project, we evaluate the top-1 accuracy, top-5 accuracy, top-10 accuracy and MRR of APIRec-CST. The results are show in Fig. 9.

We can see that there is no obvious positive or negative correlation between the number of APIs to be recommended in a hole and the top-k accuracy and MRR, which means that the top-k accuracy and MRR of APIRec-CST are insensitive to the number of APIs to be recommended in a hole. Maybe the results are not consistent with our intuition that the larger the number of APIs to be recommended in a hole the lower the top-k accuracy and MRR. For example, the top-k accuracy and MRR curves of JGit and Froyo-Email are relatively flat, which indicates that the top-k accuracy and MRR of JGit and Froyo-Email change lightly when the number of APIs to be recommended in a hole changes. The top-k accuracy and MRR curves of Itext and Log4j are relatively flat when the number is less than 6, however, the curves fluctuate when the number is more than 6. In addition, the top-k accuracy and MRR of Itext and Log4j when the number is more than 6 (or 7) are lower than the top-k accuracy and MRR of Itext and Log4j when the number is less than 6 (or 7) in most cases. The top-k accuracy and MRR curves of Galaxy and Grid-Sphere fluctuate relatively severely, which indicates that there is no correlation between the number of APIs to be recommended in a hole and the top-k accuracy and MRR of Galaxy and Grid-Sphere. The above observations are reasonable and we list some reasons as follows. First, we construct instances with different hole sizes for training, so APIRec-CST can learn semantics as much as possible no matter the hole size is small or large. Second, APIRec-CST is context sensitive, so if the APIs and code tokens in the context are enough for inferring semantics, APIRec-CST can still correctly recommend the correct API even the hole is large. For example, when a developer wants to read contents from a file line by line and he just writes one line code $FileReader\ rd = new\ FileReader(path);$ in the current method "readFile". Though the hole is large in this example, APIRec-CST can still recommend the expected APIs one by one because the APIs ($java.io.FileReader.new(java.lang.String)$) and code tokens ("read" and "file") are enough to infer the semantics of reading contents from a file line by line.

To sum up, the top-k accuracy and MRR of APIRec-CST are insensitive to the number of APIs to be recommended in a hole. In most cases, APIRec-CST achieves comparable top-k accuracy and MRR with different number of APIs to be recommended in a hole when compared with the overall top-k accuracy and MRR of each test project.

5.5 Effectiveness in Real Tasks (RQ4)

We develop an IntelliJ IDEA plugin for APIRec-CST and conduct a user study in which two groups of participants are asked to complete a set of programming tasks with and without the plugin respectively. Note that the purpose of the user study is not to compare APIRec-CST with other approaches, since we have already answered RQ1. The objective is rather to evaluate whether APIRec-CST can indeed help developers during coding. So, two groups of participants are asked to complete a set of programming tasks with and without using the APIRec-CST's plugin respectively. We derive a set of programming tasks from Stack

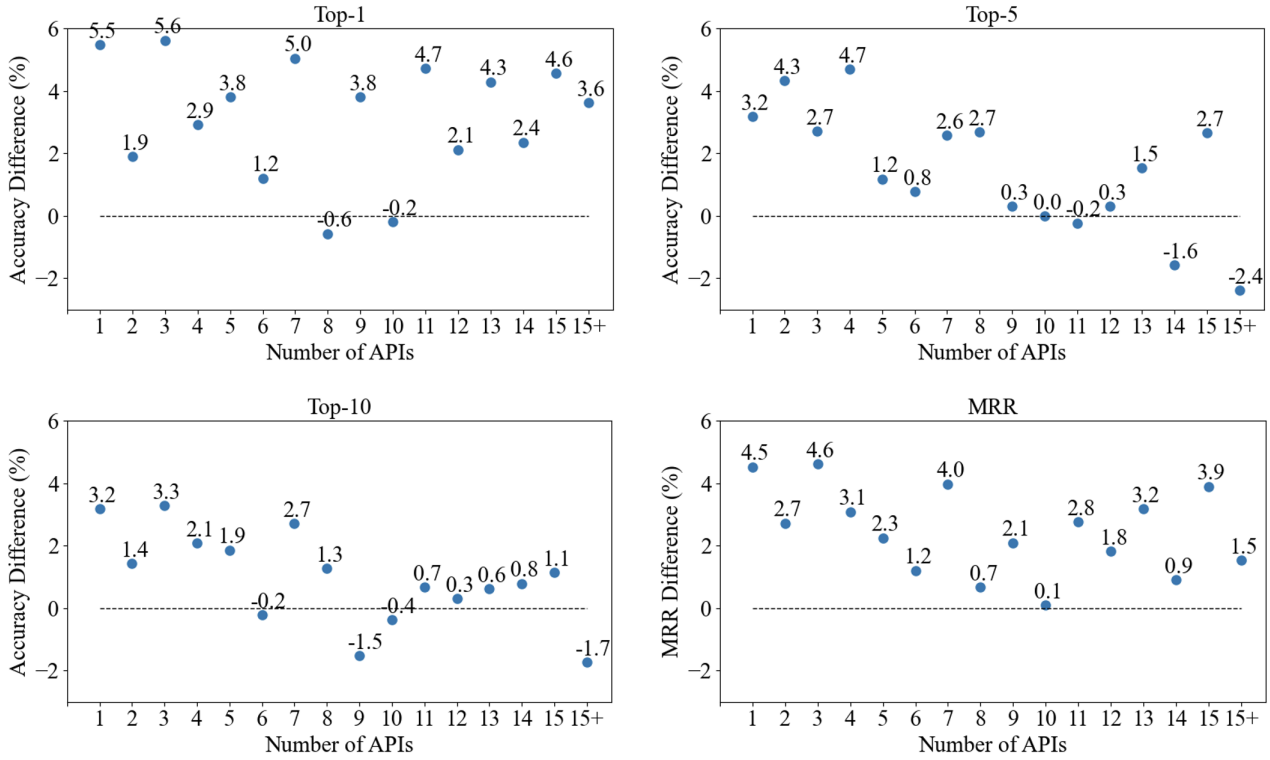

Fig. 8. Textual Code Information Contribution with the Increase of API Number in the Context

Overflow (SO) in the following way. We find the 500 most voted SO questions with the tag "Java" and identify those that can be used as programming tasks. For example, questions about concept explanation such as "Is Java pass-by-reference or pass-by-value?" are eliminated. We then choose those questions that have code snippets in the answers or question bodies that can be used to implement the desired functionalities. We further filter out the questions that have less than four lines of code or are not API intensive. We obtain 44 SO questions as candidates and randomly select the following six as the tasks. For each task we prepare a description based on the corresponding question title and body and design a set of test cases (2-9, 6 on average). We design test cases based on the following two criterion: (1) We just design normal test cases and do not design exception test cases. For example, for T1, we do not design a test case that a file does not exist. (2) We design test cases by considering all situations as much as possible for each task. For example, for T1, we design three normal test cases that are a file containing empty content, a file containing one-line content and a file containing multi lines content. The six tasks are as follows:

T1: How do I create a Java string from the contents of a file [34]

T2: Iterating through a Collection, avoiding ConcurrentModificationException when removing objects in a loop [35]

T3: How can I generate an MD5 hash [36]

T4: How do I invoke a Java method when given the method name as a string [37]

T5: How to read all files in a folder from Java [38]

T6: How can I increment a date by one day in Java [39]

We recruit 18 master students from our school and all of them major in software engineering. Since the tasks require the experience and ability of programming with JDK library, we first ask the participants to objectively evaluate their own experience and ability of programming with JDK library by themselves. We split the experience and ability of programming with JDK library into four levels, which are "poor", "ordinary", "well" and "outstanding". Each participant must choose one of the levels as their experience and ability of programming with JDK library. As a result, 12 participants choose "ordinary" and 6 participants choose "well". Thus, we divide the participants into two groups whose overall abilities are at an equivalent level (each group contains 6 participants with "ordinary" and 3 participants with "well"). We respectively assign G1 to use standard IntelliJ IDEA and G2 to use IntelliJ IDEA with the APIRec-CST plugin. Each participant is asked to complete the six tasks from T1 to T6 independently. They are not allowed to search Internet, but can look up the JDK reference documentation and use the code recommendation feature and other facilities provided by IntelliJ IDEA. The participants in G2 can request the help of the APIRec-CST plugin, which can provide a list of top 10 API recommendations for the current cursor position. The participants are asked to write the method signature even though they have no idea of a task. The method signature is named by the participants themselves, so we ask the participants to use meaningful naming instead of casual naming. For each task each participant is given 20 minutes and if he (she) can not finish the task in time he (she) has to stop and submits his (her) implementation of the task. We record the completion time

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3074309, IEEE Transactions on Software Engineering

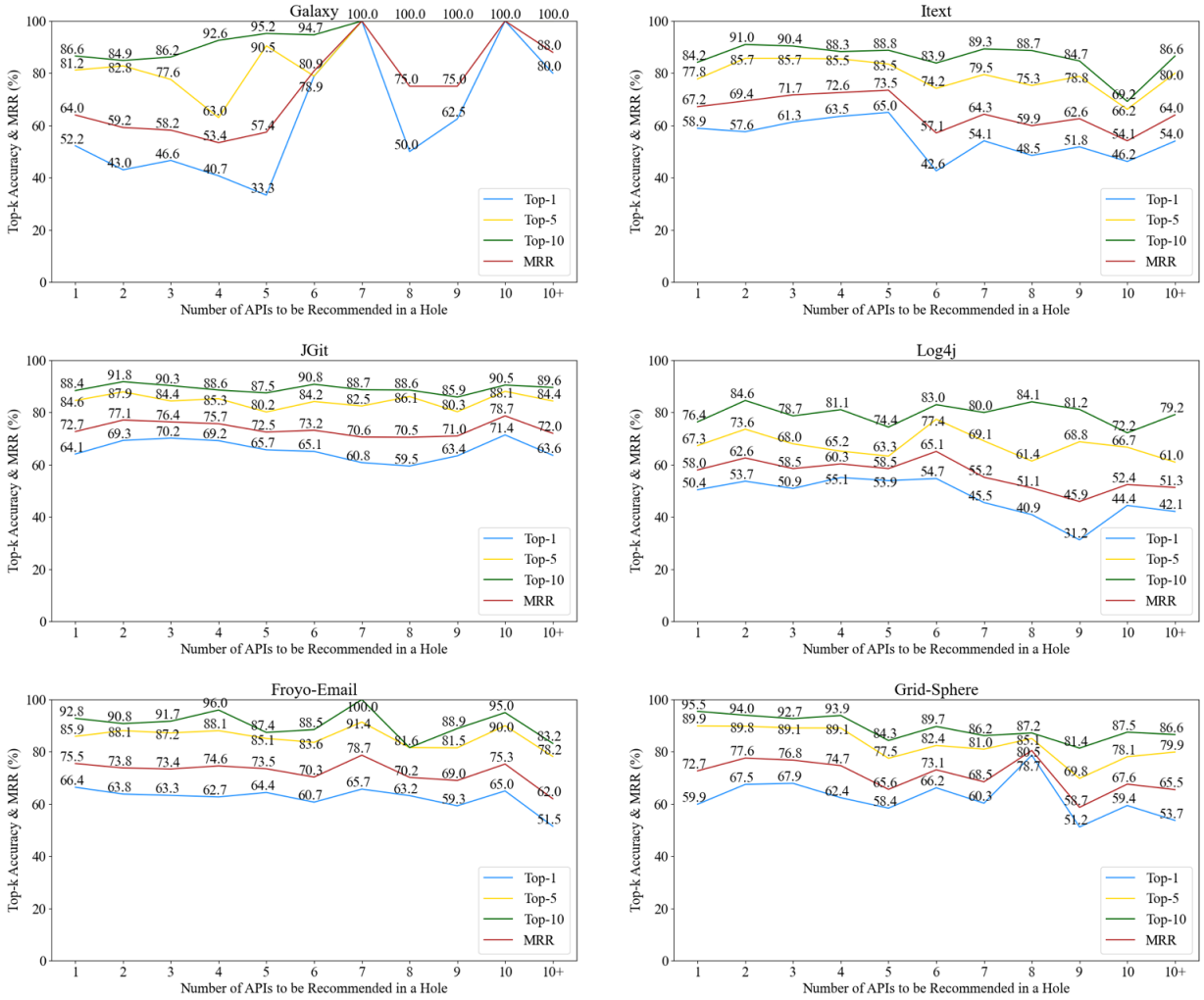

Fig. 9. Sensitivity Analysis on the Number of APIs in a Hole of APIRec-CST

of the participants and test their implementations for each task.

Before formal experiments, we train the participants in G2 on how to use APIRec-CST. We show the participants in G2 how to use APIRec-CST through an example and then let the participants in G2 try to use APIRec-CST based on this example until each of them is skilled at using APIRec-CST. The participants in G2 are enforced to use APIRec-CST when they do not know which API call to use next or have no idea of a task. The participants in G2 can also use APIRec-CST to help to reduce the typing time even if they know what to write next.

We use task completion time and test pass rate as two metrics for evaluation. Task completion time is the time that a participant used to complete a task. Given a submitted implementation of a task, test pass rate is the percentage of test cases passed in the total number of test cases. The results of descriptive statistics analysis of task completion time and test pass rate are shown in Table 11 and Table 12 respectively. On average, the participants in G1 use 665.7-1,173.3 seconds to finish a task, while the participants in G2 use 441.3-708.1 seconds to finish a task; the participants in G1 pass 4-47% test cases, while the participants in G2 pass 68-89% test cases. We can see that APIRec-CST helps the participants finish the tasks faster and more accurately. Furthermore, we evaluate whether the improvements are significant or not. We make a significance test using the Mann-Whitney U test where a difference is thought to be significant if the p-value is less than 0.05. We can see that the participants in G2 significantly outperform the participants in G1 in terms of completion time for three tasks and in terms of test pass rate for five tasks. Furthermore, we count the average number of times that participants in G2 actually use APIRec-CST for each task. On average, participants in G2 actually use APIRec-CST 10 times for T1, 7 times for T2, 6 times for T3, 6 times for T4, 9 times for T5 and 10 times for T6. It indicates that participants in G2 are willing to use APIRec-CST when they need help. To sum up, developers are more effective in terms of task completion time and test pass rate when accomplishing programming tasks using APIRec-CST.

We have an interview with each of the participants in G2 to get their feedback on APIRec-CST. Most of them agree that APIRec-CST provides accurate recommendations which are quite helpful especially when they do not know how to proceed. In most cases, the right API is included

TABLE 11
Completion Time of the Tasks (Seconds)

Task	Group	avg	min	max	median	stan. dev.	p
T1	G1	888.0	485	1200	900	299.83	0.0904
	G2	680.0	188	1200	718	343.13	
T2	G1	671.4	232	1200	480	387.91	0.2677
	G2	562.4	246	1003	449	298.53	
T3	G1	1173.3	960	1200	1200	75.42	0.0001
	G2	441.3	160	703	463	150.04	
T4	G1	1159.6	836	1200	1200	114.39	0.0003
	G2	708.1	431	1154	697	211.87	
T5	G1	665.7	345	1200	558	295.51	0.0924
	G2	475.0	232	746	427	184.56	
T6	G1	1140.0	660	1200	1200	169.71	0.0013
	G2	707.3	255	1200	658	340.67	

TABLE 12
Test Pass Rate of the Tasks

Task	Group	avg	min	max	median	stan. dev.	p-value
T1	G1	0.26	0.00	1.00	0.00	0.41	0.0073
	G2	0.81	0.00	1.00	1.00	0.36	
T2	G1	0.47	0.00	1.00	0.33	0.41	0.1461
	G2	0.68	0.00	1.00	1.00	0.39	
T3	G1	0.11	0.00	1.00	0.00	0.31	0.0008
	G2	0.89	0.00	1.00	1.00	0.31	
T4	G1	0.11	0.00	1.00	0.00	0.31	0.0012
	G2	0.83	0.00	1.00	1.00	0.33	
T5	G1	0.36	0.00	1.00	0.38	0.37	0.0030
	G2	0.89	0.5	1.00	1.00	0.21	
T6	G1	0.04	0.00	0.33	0.00	0.10	0.0013
	G2	0.78	0.00	1.00	1.00	0.42	

in the top 5 recommendations. In extreme cases, APIRec-CST can even provide right APIs when the participants only declare a method (including method name and parameters). This indicates that APIRec-CST can provide useful recommendations by only using textual code information. They also provide suggestions for further improvement. Two common suggestions are recommending arguments for API invocation and providing explanations for the recommended APIs.

5.6 Qualitative Analysis

In RQ1 and RQ2, we perform quantitative analysis on APIRec-CST, thus we list some examples to qualitatively illustrate the advantages of APIRec-CST. In Fig. 10, we list five examples.

The first example is to read contents from a reader of a file line by line. As we can see that GraLan recommends the correct API in the third place, Tree-LSTM recommends the correct API in the second place, and both APIRec-SO and APIRec-CST recommend the correct API in the first place. The first two recommendations of GraLan are due to the irrelevant subgraphs that capture the semantics of list operation. This suggests that though subgraphs in GraLan may capture the semantics at a hole, the recommendations may be over-shadowed by other irrelevant subgraphs. Tree-LSTM is a deep learning model using tree-based structure which includes control flow among APIs but lacks data flow. Tree-LSTM treats source code as code tree, and feed the code tree into the deep learning model. Compared to GraLan, Tree-LSTM also considers the structure information but lack of data flow. Thus, Tree-LSTM performs better than GraLan, although worse than APIRec-SO and APIRec-CST. APIRec-SO and APIRec-CST treat source code as an API context graph which contains the structure information, and apply GG-NNs to learn the semantic in an API context graph using a holistic view. Due to the information diffusion mechanism in GG-NNs, each node itself and its relations to other nodes in the API context graph are integrated and added to the final vector representation of the API context graph. As a result, APIRec-SO and APIRec-CST successfully recommend that the API of the hole should be used to read the next line. From this example, we can see that a holistic view of correlated API usage in control and data flow graph of an entire method can help to improve the ranking of the correct API.

The second example is to draw a BufferedImage given a RenderedImage. As we can see that GraLan and Tree-LSTM fail to recommend the correct API in the top 5 recommendations, whereas APIRec-SO and APIRec-CST successfully recommend the correct API in the first place. In this example, none of the subgraphs in GraLan can capture the real semantics at the hole. Most of GraLan's recommendations are the APIs in $java.util.Hashtable$ because APIs in $java.util.Hashtable$ are closest to the hole and are used as context in subgraphs. Due to the lack of data flow among APIs, Tree-LSTM cannot recommend the correct API. Only with a holistic view of a control and data flow graph of an entire method, can APIRec-SO and APIRec-CST find that all the APIs in the method are prepared to be used as the parameters of the correct API $java.awt.image.BufferedImage.new(java.awt.image.ColorModel, java.awt.image.WritableRaster, boolean, java.util.Hashtable)$ to create a BufferedImage object. From this example, we can see that in some situations, a holistic view of correlated API usage in control and data flow graph of an entire method can help to recommend the correct API.

The third example is to remove an old database file. As we can see that GraLan fails to recommend the correct API in the top 5 recommendations, Tree-LSTM recommends the correct API in the fifth place, whereas APIRec-SO recommends the correct API in the second and APIRec-CST recommends the correct API in the first place. All of the approaches capture the semantics at the hole is to apply an operation to a File object. However, the first three approaches fail to identify which operation should be applied to the File object. APIRec-CST leverages the method name as textual information in which "remove" indicates that the operation is to delete a file. APIRec-CST applies a Code Token Network to embed the textual information to capture the semantics in the textual information and combined (joint) with the structure information. From this example, we can see that the method name is indeed helpful to clarify the semantics.

The fourth example is to set the time in millisecond of a given value. As we can see that GraLan and Tree-LSTM fail to recommend the correct API in the top 5 recommendations, APIRec-SO recommends the correct API in the third place and APIRec-CST recommends the correct API in the first place. Since there is only one JDK API (Calendar) in the method, recommendations of all the approaches are related to the Calendar object. However, the first three approaches cannot recommend the correct API in first place because they are not certain which operation should be applied on the Calendar object. APIRec-CST leverages the parameter

ID	1

Source Code	`public static List readLines(Reader input) throws IOException {`
	`  BufferedReader reader = new BufferedReader(input);`
	`  List list = new ArrayList();`
	`  String line = reader.readLine();`
	`  while (line != null) {`
	`  list.add(line);`
	`  /*hole*/`
	`  }`
	`  return list;`
	`}`

Correct API	java.io.BufferedReader.readLine
GraLan Top-5 Predictions	java.util.List.add, java.util.ArrayList.new, java.io.BufferedReader.readLine, java.io.BufferedReader.new, while
Tree-LSTM Top-5 Predictions	java.util.ArrayList.add, java.io.BufferedReader.readLine, if, java.lang.String.trim, java.lang.Integer.parseInt
APIRec-SO Top-5 Predictions	java.io.BufferedReader.readLine, if, java.util.ArrayList.add, java.util.ArrayList.toArray, java.lang.String.charAt
APIRec-CST Top-5 Predictions	java.io.BufferedReader.readLine, if, while, java.util.ArrayList.add, java.io.BufferedReader.mark

ID	2

Source Code	`public void drawRenderedImage(RenderedImage img, AffineTransform xform) {`
	`  BufferedImage image = null;`
	`  if (img instanceof BufferedImage) {`
	`  image = (BufferedImage)img;`
	`  } else {`
	`  ColorModel cm = img.getColorModel();`
	`  int width = img.getWidth();`
	`  int height = img.getHeight();`
	`  WritableRaster raster = cm.createCompatibleWritableRaster(width, height);`
	`  boolean isAlphaPremultiplied = cm.isAlphaPremultiplied();`
	`  Hashtable<String, Object> properties = new Hashtable<String, Object>();`
	`  String[] keys = img.getPropertyNames();`
	`  if (keys!=null) {`
	`  for (String key : keys) {`
	`  properties.put(key, img.getProperty(key));`
	`  }`
	`  }`
	`  /*hole*/`
	`  }`
	`  drawImage(image, xform, null);`
	`}`

Correct API	java.awt.image.BufferedImage.new
GraLan Top-5 Predictions	if, java.util.Hashtable.put, java.util.Hashtable.new, java.util.Hashtable.get, for
Tree-LSTM Top-5 Predictions	if, java.awt.RenderingHints.KEY_TEXT_ANTIALIASING, for, java.awt.Color.new, java.util.ArrayList.add
APIRec-SO Top-5 Predictions	java.awt.image.BufferedImage.new, java.awt.image.ColorModel.isAlphaPremultiplied, java.util.Hashtable.new, if, java.awt.image.RenderedImage.getProperty
APIRec-CST Top-5 Predictions	java.awt.image.BufferedImage.new, java.awt.image.RenderedImage.copyData, java.awt.image.RenderedImage.getPropertyNames, if, java.awt.image.RenderedImage.getProperty

ID	3

Source Code	`private void removeOldDatabaseFile() {`
	`  String dbpath = settingsService.getRealSettingsPath("database");`
	`  File dbdir = new File(dbpath);`
	`  String[] filenames = dbdir.list();`
	`  String currentVersion = null;`
	`  for (int i = 0; i < filenames.length; i++) {`
	`  if (filenames[i].startsWith("GS")) currentVersion = filenames[i];`
	`  }`
	`  if (currentVersion != null) {`
	`  File f = new File(currentVersion);`
	`  /*hole*/`
	`  }`
	`}`

Correct API	java.io.File.delete
GraLan Top-5 Predictions	if, java.io.File.new, java.io.File.exists, java.io.File.getAbsolutePath, for
Tree-LSTM Top-5 Predictions	if, java.io.File.new, java.io.File.getAbsolutePath, java.io.File.mkdirs, java.io.File.delete
APIRec-SO Top-5 Predictions	if, java.io.File.delete, java.io.File.getAbsolutePath, java.io.File.getName, java.io.File.renameTo
APIRec-CST Top-5 Predictions	java.io.File.delete, java.io.File.renameTo, if, java.io.File.exists, java.io.File.getAbsolutePath

ID	4

Source Code	`private static long dayStartInMillis(long timeInMillis) {`
	`  Calendar cal = new GregorianCalendar(`
	`SystemReader.getInstance().getTimeZone());`
	`  /*hole*/`
	`}`

Correct API	java.util.GregorianCalendar.setTimeInMillis
GraLan Top-5 Predictions	java.util.GregorianCalendar.getTime, java.util.Calendar.set, java.util.GregorianCalendar.new, if, java.util.Calendar.getTime
Tree-LSTM Top-5 Predictions	java.util.GregorianCalendar.setTime, java.util.GregorianCalendar.set, java.util.Date.new, java.util.GregorianCalendar.clear, java.util.GregorianCalendar.new
APIRec-SO Top-5 Predictions	java.util.GregorianCalendar.setTime, java.util.GregorianCalendar.clear, java.util.GregorianCalendar.setTimeInMillis, for, java.util.GregorianCalendar.set
APIRec-CST Top-5 Predictions	java.util.GregorianCalendar.setTimeInMillis, java.util.GregorianCalendar.setTime, java.util.GregorianCalendar.set, java.util.GregorianCalendar.clear, if

ID	5

Source Code	`public void init() {`
	`  File dir = new File(OUT_FOLDER);`
	`  if (dir.exists()) {`
	`  deleteDirectory(dir);`
	`  }`
	`  /*hole*/`
	`}`

Correct API	java.io.File.mkdirs
GraLan Top-5 Predictions	if, java.io.File.new, java.io.File.exists, java.io.File.getAbsolutePath, for
Tree-LSTM Top-5 Predictions	java.io.File.delete, if, java.io.File.new, java.io.File.getAbsolutePath, java.io.File.mkdirs
APIRec-SO Top-5 Predictions	if, java.io.File.new, java.io.File.mkdirs, java.io.File.getAbsolutePath, java.io.FileInputStream.new
APIRec-CST Top-5 Predictions	java.io.File.mkdirs, if, java.io.File.mkdir, java.io.File.new, java.io.File.listFiles

Fig. 10. Positive Examples for Qualitative Analysis

name as textual information in which "time", "in" and "millis" ("in" and "millis" are also in the method name) indicate that the operation is to process time in millisecond. Combined with the semantics in the API context graph, APIRec-CST successfully identifies that the operation is to set the time in millisecond. From this example, we can see that the parameter name is helpful to clarify the semantics.

The last example is to create a new directory (delete the original directory if the directory exists). As we can see that GraLan fail to recommend the correct API in the top 5 recommendations, Tree-LSTM recommends the correct API in the fifth place, APIRec-SO recommends the correct API in the third place and APIRec-CST recommends the correct API in the first place. All of the approaches capture the semantics at the hole is to apply an operation to a File object. However, the first three approaches fail to identify which operation should be applied to the File object, and thus fail to recommend the correct API in the first place. APIRec-CST leverages the variable names as textual information in which "dir" and "folder" indicate that the operation should applied on a directory not a file. Combined with the semantics in the API context graph, APIRec-CST successfully identifies that there lacks a new directory and thus the operation is to create a new directory. From this example, we can see that the variable names are helpful to clarify the semantics.

Since textual information may introduce noise that affect the accuracy of APIRec-CST for API recommendation based on the findings in RQ2, we further list some examples that negatively affect the accuracy of APIRec-CST when introducing textual information. In Fig. 11, we list two examples.

The first example is to get a list that contains all the possible logging levels (such as SEVERE, WARNING and FINER). As we can see that GraLan recommends the correct API in the second place, Tree-LSTM and APIRec-SO recommend the correct API in the first place, and APIRec-CST recommends the correct API in third place. APIRec-CST achieves the worst performance. In this example, the code tokens in the textual information mislead the implementation consideration due to the lack of in-depth understanding of the context. The code tokens "get" and "all" in the method name mislead APIRec-CST to recommend for (iteratively adding all the elements in a loop) and $java.util.ArrayList.addAll$. However, the developer chooses to add all the levels one by one using $java.util.ArrayList.add$ as they are defined as a set of enumeration values.

The second example is to write the level of an event as well as its class name. As we can see that GraLan, Tree-LSTM and APIRec-CST fail to recommend the correct API in the top 5 recommendations, and APIRec-SO recommends the correct API in the fifth place. APIRec-CST fails to recommend the correct API when introducing the textual information. In this example, the code tokens in the textual information are more relevant to the APIs after the current position. The code token "write" in the method name makes APIRec-CST miss the correct API $java.lang.Class.getName$ in the top-5 recommendations. In fact, the recommended APIs related to object writing are the core APIs but will be used later after the current position.

5.7 Threats to Validity

The threats to the internal validity of our studies lie in four aspects. First, the GraLan implementation may not be exactly consistent with the approach. Second, we use the GloVe vocabulary to determine meaningful tokens. The GloVe vocabulary contains 400K most frequent unique tokens obtained from Wikipedia and Gigaword. Wikipedia contains software engineering-specific tokens, however, Wikipedia maybe does not contain all tokens that are software engineering-specific. Thus, tokens found only in source code or software may be filtered, despite them being meaningful in the software domain, which maybe decreases the accuracy of APIRec-CST. However, we think that this impact will be light, because Wikipedia is large enough and our experiment results show the effectiveness of adding textual information. In the future, we will seek for stronger strategies for text processing. Third, since the control and data dependency analysis is performed statically in APIRec-CST, we acknowledge that it might not be fully accurate. Because we are suffered from the problem of type hierarchy, which is a common problem in static analysis. This is however the standard approach in existing state-of-the-art approaches [8], [9], since obtaining control/data dependency through dynamic analysis has its limitations as well. In our future work, we will try our best to solve this problem. Fourth, the test cases developed for each task may not be complete.

The threats to the external validity of our studies lie in three aspects. First, we only implement our approach for Java and evaluate it with JDK. It is not clear how well the approach can support other languages and API libraries. However, our approach can be easily extended to other libraries. The representations of the API context graph and the code tokens are general. To construct the API context graph containing APIs in other libraries, we need to import the target libraries for using Java reflection mechanism and then use JavaParser to parse the source code for constructing the API context graph and code tokens. For other object-oriented programming languages, we need to use the corresponding parser instead of JavaParser to parse the source code. The cost (such as time and computation power) of retraining APIRec-CST is determined and influenced by the total number of training and validation instances instead of libraries or languages. Second, as adopted in [8], [9], [17], the test cases used in RQ1 are constructed automatically which may not reflect the scenarios in the real world. Different from existing approaches, we additionally conduct a user study to simulate the scenarios in the real world to evaluate the effectiveness of APIRec-CST. Third, we only evaluate the approach with a group of master students and a set of tasks from SO in the user study. It is not clear how effectively the approach can support industrial developers to accomplish more complex programming tasks.

6 Related Work

This work is closely related to various research on code recommendation. In modern IDE (Integrated Development Environment), type information is often used to recommend API method calls when classes or objects are typed. To enhance the performance of the code recommendation in

ID	1

Source Code	`public static List getAllPossibleLevels() {`
	`  ArrayList list = new ArrayList();`
	`  /*hole*/`
	`  return list;`
	`}`

Correct API	java.util.ArrayList.add
GraLan Top-5 Predictions	java.util.List.add, java.util.ArrayList.add, for, if, java.util.List.size
Tree-LSTM Top-5 Predictions	java.util.ArrayList.add, java.util.ArrayList.new, java.util.ArrayList.addAll, for, if
APIRec-SO Top-5 Predictions	java.util.ArrayList.add, java.util.ArrayList.new, for, java.util.ArrayList.addAll, if
APIRec-CST Top-5 Predictions	for, java.util.ArrayList.addAll, java.util.ArrayList.add, if, java.util.ArrayList.new

ID	2

Source Code	`private void writeLevel(ObjectOutputStream oos) throws java.io.IOException {`
	`  oos.writeInt(level.toInt());`
	`  Class clazz = level.getClass();`
	`  if(clazz == Level.class) {`
	`  oos.writeObject(null);`
	`  } else {`
	`  /*hole*/`
	`  }`
	`}`

Correct API	java.lang.Class.getName
GraLan Top-5 Predictions	java.io.ObjectOutputStream.new, java.io.ObjectOutputStream.close, if, java.io.ObjectOutputStream.defaultWriteObject, java.io.ObjectOutputStream.flush
Tree-LSTM Top-5 Predictions	java.io.ObjectOutputStream.writeObject, java.io.ObjectOutput.writeUTF, java.io.ObjectOutputStream.writeBoolean, java.io.ObjectOutputStream.close, java.io.ObjectOutputStream.writeInt
APIRec-SO Top-5 Predictions	java.io.ObjectOutputStream.defaultWriteObject, java.io.ObjectOutputStream.writeObject, java.io.ObjectOutputStream.writeUTF, java.io.ObjectOutputStream.writeBoolean, java.lang.Class.getName
APIRec-CST Top-5 Predictions	java.io.ObjectOutputStream.writeObject, java.io.ObjectOutputStream.writeBoolean, java.io.ObjectOutputStream.writeUTF, java.io.ObjectOutputStream.writeInt, java.io.ObjectOutputStream.writeLong

Fig. 11. Negative Examples for Qualitative Analysis

IDEs, several approaches have been proposed to sort, filter and group API methods for better recommendation [40], [41], [42]. In comparison, APIRec-CST does not require a developer to write a receiver expression. Heinemann *et al.* [43] propose an API method recommendation algorithm based on the extracted identifiers (such as variable and type names) in the development context. In comparison, in addition to the textual information (including identifiers), APIRec-CST also takes structural information into consideration. Besides, APIRec-CST uses a deep neural network to learn the semantics of the textual information instead of simply using Jaccard similarity. Several approaches [44], [45] compute the similarity between the current code context and previous code examples based on a set of API calls or other additional information (such as method names, Java keywords, class or interface names). In comparison, APIRec-CST considers the complete API usage modeled in an API context graph, which contains API calls, Java keywords, and control and data flow among them. Furthermore, APIRec-CST combines textual information which includes method names, parameter names, and variable names.

This work is also related to work on mining usage patterns from source code, such as [46], [47], [48], [49], [50]. These approaches often apply deterministic mining algorithms to mine usage patterns for code recommendation. Zhong *et al.* [48] propose MAPO to cluster code snippets and mine usage patterns by frequent subsequence mining. Nguyen *et al.* [49] propose GrouMiner to mine usage patterns by representing source code as groums. Nguyen *et al.* propose Grapacc [50], which first mines usage patterns based on graphs and then matches these patterns with the code fragment under editing based on graph-based features and token-based features. Wang *et al.* [47] apply a two-step clustering strategy to cluster call sequences and mine usage patterns for each cluster using a frequent closed sequence mining algorithm. Fowkes *et al.* [46] propose a near parameter-free probabilistic algorithm to infer the most interesting usage patterns. In comparison, APIRec-CST learn regularity of the API usage based on deep learning techniques instead of mining usage patterns explicitly.

Based on the conjecture that source code is naturally repetitive and predictable [1], many approaches have been proposed to learn statistical language models from source code for code recommendation. Hindle *et al.* [1] train an n-gram model based on the tokens of the source code to recommend the next token. Allamanis *et al.* [2] use a large corpus of source code from various domains to train an n-gram model. Nguyen *et al.* [3] enhance the n-gram model with roles and data types of code tokens and global technical concerns/functionality. Tu *et al.* [4] enhance the n-gram model with a cache to capture the localized regularities in the source code to improve the accuracy. Nguyen *et al.* [8] propose a graph-based statistical language model by using Bayesian statistical inference to compute the probabilities of API recommendations based on graphs. Liu *et al.* [9] propose a re-ranking approach based on the top-10 recommendations of GraLan to improve the top-1 accuracy using API usage paths as features. Nguyen *et al.* [30] propose APIRec that learns from fine-grained code changes by developing an association-based change inference model to recommend API calls. Xie *et al.* [51] propose HiRec which leverages hierarchical context with a statistical model for API recommendation. HiRec identifies third-party APIs in the project-specific methods and use them as hierarchical context. However, HiRec does not take control and data flow into consideration and ignores the impact of textual code information. In addition, HiRec focuses on recommending API methods of a given object, which means that a developer needs to write the object. However, APIRec-CST does not require a developer to write an object. In comparison, APIRec-CST learns from the control and data flow in the source code instead of treating the source code as tokens as in the above-mentioned proposals. Furthermore, APIRec-CST takes a holistic approach to learn from both structural and textual code information.

There are also approaches which apply deep learning

techniques for code recommendation. Raychev *et al.* [5] treat the source code as sentences and combine the n-gram model with RNN for recommending sentences. Dam *et al.* [6] train an LSTM (Long Short-Term Memory) neural network based on code tokens. Nguyen *et al.* [7] train a deep neural network called Dnn4C, which not only leverages the local context of lexical code elements, but also syntactic and type contexts. Svyatkovskiy *et al.* [52] reformulate code completion from a generation task to a task of learning to rank the valid completion suggestions computed from a candidate provider. They propose a framework to design and combine different deep learning components to retrieve a range of trade-offs beyond reusing existing neural architectures. However, they do not focus on improving the representation of source code (such as considering control and data flow) and designing new deep learning models. Liu *et al.* [53] propose an MTL-based self-attentional neural architecture, which leverages the paths in ASTs and combines the tasks of predicting next value and predicting next type in an AST in the same learning process. In comparison, APIRec-CST combines a graph-based deep neural network and a token-based deep neural network to capture both structural and textual code information.

This work is broadly related to other applications of deep learning techniques on source code for various objectives including code summarization [54], code generation [55], [56], [57], [58], [59], code search [60], [61], comment generation [62], [63], [64], [65], [66] or defect prediction [67], [68]. For example, Allamanis *et al.* [54] propose an attentional neural network to give an extreme summary of a sequence of code tokens. Mou *et al.* [59] apply a sequence-to-sequence recurrent neural network to generate code when given a user intention. Gu *et al.* [61] propose a deep neural network called CODEnn to jointly embed code snippets and natural language descriptions. Hu *et al.* [62] propose DeepCom which takes AST sequences of source code as input and generates the corresponding comments based on an attentional Seq2Seq model. Wang *et al.* [68] apply Deep Belief Network to learn features of tokens extracted from source code for defect prediction. These approaches apply different deep learning models to learn program semantics for different objectives. In comparison, APIRec-CST represents program as an API context graph and a bag of code tokens and designs a novel deep neural network for API recommendation.

There are also approaches focusing on recommending API code snippets or API usage examples, such as [69], [70], [71]. For example, Moreno *et al.* [72] propose an approach named MUSE to mine and rank actual code examples which can show the usage of a method. MUSE extracts method usage examples by using static code slicing and clone detection. Gu *et al.* [73] propose a graph kernel based approach named CodeKernel to recommend API usage examples for developers. CodeKernel models source code as graphs and clusters the graphs by a graph kernel method. The task of recommending API code snippets or API usage examples and the task of recommending the next API are two different scenarios. In comparison, though recommending API code snippets or API usage examples can provide a complete code snippet or usage example to developers, the accuracy is not as high as recommending the next API. In addition, developers may spend a lot of time and effort to identify and modify the code they need and delete the code they do not need in the recommended API code snippets or API usage examples. However, when recommending the next API, developers just need to choose the API they need and modify the parameters. Furthermore, when recommending the next API one by one, it is flexible for developers to find more API usages.

7 Conclusion

In this paper we propose a deep learning based API recommendation approach that combines the API usage with the text information in the source code to simultaneously learn structural and textual features. Our evaluation shows that our approach significantly outperforms an existing graph-based statistical model and a tree-based deep learning model for API recommendation and can effectively help students to finish programming tasks faster and more accurately. Our future work will improve the approach from several aspects. First, we will improve the utilization of textual code information, for example by using better data preprocessing methods and model architectures or introducing user interactions. Second, we will incorporate argument recommendation and API explanation into the approach. Third, we will apply our approach for other API libraries and try to extend the approach to support API recommendation of multiple libraries.

References

[1] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 837–847.

[2] M. Allamanis and C. A. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 207–216.

[3] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 532–542.

[4] Z. Tu, Z. Su, and P. T. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 269–280.

[5] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 419–428.

[6] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *CoRR*, vol. abs/1608.02715, 2016.

[7] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 323–334.

[8] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 858–868.

[9] X. Liu, L. Huang, and V. Ng, "Effective API recommendation without historical software repositories," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 282–292.

[10] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," *CoRR*, vol. abs/1511.05493, 2015.

[11] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[12] L. B. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment." in *Proceedings, 1st First International Conference on Neural Networks*, vol. 2, 1987, pp. 609–618.

[13] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Physical review letters*, vol. 59, no. 19, p. 2229, 1987.

[14] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014.

[15] "Camel case," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Camel_case

[16] "Glove," 2020. [Online]. Available: https://nlp.stanford.edu/projects/glove

[17] C. Chen, X. Peng, J. Sun, Z. Xing, X. Wang, Y. Zhao, H. Zhang, and W. Zhao, "Generative API usage code recommendation with parameter concretization," *SCIENCE CHINA Information Sciences*, vol. 62, no. 9, pp. 192 103:1–192 103:22, 2019.

[18] "Javaparser," 2020. [Online]. Available: https://github.com/javaparser/javaparser/

[19] "Stanford corenlp," 2020. [Online]. Available: https://stanfordnlp.github.io/CoreNLP/

[20] "Tensorflow," 2020. [Online]. Available: https://github.com/tensorflow/tensorflow

[21] "gg-nns reference implementation," 2020. [Online]. Available: https://github.com/Microsoft/gated-graph-neural-network-samples

[22] "Replication package," 2020. [Online]. Available: https://apireccst.wixsite.com/apirec-cst

[23] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "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, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 383–392.

[24] "Galaxy," 2020. [Online]. Available: https://github.com/puniverse/galaxy

[25] "Log4j," 2020. [Online]. Available: https://github.com/apache/log4j

[26] "Jgit," 2020. [Online]. Available: https://github.com/eclipse/jgit

[27] "Froyo-email," 2020. [Online]. Available: https://github.com/Dustinmj/Froyo_Email

[28] "Grid-sphere," 2020. [Online]. Available: https://github.com/brandt/GridSphere

[29] "Itext," 2020. [Online]. Available: https://github.com/itext/itextpdf

[30] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 511–522.

[31] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: a case study on real-world completions," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 960–970.

[32] B. Schwartz, "The paradox of choice: Why more is less." Ecco New York, 2004.

[33] E. Reutskaja, A. Lindner, R. Nagel, R. A. Andersen, and C. F. Camerer, "Choice overload reduces neural signatures of choice set value in dorsal striatum and anterior cingulate cortex," *Nature Human Behaviour*, vol. 2, no. 12, pp. 925–935, 2018.

[34] "Stack overflow question," 2020. [Online]. Available: https://stackoverflow.com/questions/326390/

[35] "Stack overflow question," 2020. [Online]. Available: https://stackoverflow.com/questions/223918/

[36] "Stack overflow question," 2020. [Online]. Available: https://stackoverflow.com/questions/415953/

[37] "Stack overflow question," 2020. [Online]. Available: https://stackoverflow.com/questions/160970/

[38] "Stack overflow question," 2020. [Online]. Available: https://stackoverflow.com/questions/1844688/

[39] "Stack overflow question," 2020. [Online]. Available: https://stackoverflow.com/questions/428918/

[40] D. M. Pletcher and D. Hou, "BCC: enhancing code completion for better API usability," in *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, 2009, pp. 393–394.

[41] D. Hou and D. M. Pletcher, "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, RSSE 2010, Cape Town, South Africa, May 4, 2010*, 2010, pp. 26–30.

[42] ——, "An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion," in *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, 2011, pp. 233–242.

[43] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel, "Identifier-based context-dependent API method recommendation," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, 2012, pp. 31–40.

[44] M. Bruch, M. Monperrus, and M. Mezini, "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, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 213–222.

[45] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "A simple, efficient, context-sensitive approach for code completion," *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 512–541, 2016.

[46] J. M. Fowkes and C. A. Sutton, "Parameter-free probabilistic API mining at github scale," *CoRR*, vol. abs/1512.05558, 2015.

[47] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 319–328.

[48] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, 2009, pp. 318–343.

[49] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "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, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 383–392.

[50] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 69–79.

[51] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "Hirec: API recommendation using hierarchical context," in *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, 2019, pp. 369–379.

[52] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," *CoRR*, vol. abs/2004.13651, 2020.

[53] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, 2020, pp. 37–47.

[54] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 2091–2100.

[55] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 440–450.

[56] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," *CoRR*, vol. abs/1811.06837, 2018.

[57] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang, and A. W. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

[58] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 1139–1149.

[59] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin, "On end-to-end program generation from user intention by deep neural networks," *CoRR*, vol. abs/1510.07211, 2015.

[60] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 631–642.

[61] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 933–944.

[62] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, 2018, pp. 200–210.

[63] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, pp. 2269–2275.

[64] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 2018, pp. 5229–5236.

[65] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 397–407.

[66] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

[67] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, 2015, pp. 17–26.

[68] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 297–308.

[69] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: code recommendation via structural code search," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 152:1–152:28, 2019.

[70] L. Ai, Z. Huang, W. Li, Y. Zhou, and Y. Yu, "SENSORY: leveraging code statement sequence information for code snippets recommendation," in *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, 2019, pp. 27–36.

[71] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 946–957.

[72] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus, "How can I use this method?" in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 880–890.

[73] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of API usage examples," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 590–601.