1-2022

"More Than Deep Learning": Post-processing for API sequence recommendation

Chi CHEN

Xin PENG

Bihuan CHEN

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

Zhenchang XING

See next page for additional authors

Citation

CHEN, Chi; PENG, Xin; CHEN, Bihuan; SUN, Jun; XING, Zhenchang; WANG, Xin; and ZHAO, Wenyun. "More Than Deep Learning": Post-processing for API sequence recommendation. (2022). *Empirical Software Engineering*. 27, (1), 1-32.
Available at: https://ink.library.smu.edu.sg/sis_research/6580

Published in Empirical Software Engineering. October 2021, pp. 1-32.
https://10.1007/s10664-021-10040-2

"More Than Deep Learning": post-processing for API sequence recommendation

Chi Chen[1,2] · Xin Peng[1,2] · Bihuan Chen[1,2] · Jun Sun[3] · Zhenchang Xing[4] · Xin Wang[1,2] · Wenyun Zhao[1,2]

Accepted: 16 August 2021Published online: 29 October 2021

Abstract

In the daily development process, developers often need assistance in finding a sequence of APIs to accomplish their development tasks. Existing deep learning models, which have recently been developed for recommending one single API, can be adapted by using encoder-decoder models together with beam search to generate API sequence recommendations. However, the generated API sequence recommendations heavily rely on the probabilities of API suggestions at each decoding step, which do not take into account other domain-specific factors (e.g., whether an API suggestion satisfies the program syntax and how diverse the API sequence recommendations are). Moreover, it is difficult for developers to find similar API sequence recommendations, distinguish different API sequence recommendations, and make a selection when the API sequence recommendations are ordered by probabilities. Thus, what we need is more than deep learning. In this paper, we propose an approach, named COOK, to combine deep learning models with post-processing strategies for API sequence recommendation. Specifically, we enhance beam search with code-specific heuristics to improve the quality of API sequence recommendations. We develop a clustering algorithm to cluster API sequence recommendations so as to make it easier for developers to find similar API sequence recommendations and distinguish different API sequence recommendations. We also propose a method to generate a summary for each cluster to help developers understand the API sequence recommendations. Our evaluation results have shown that (1) three deep learning models with our heuristic-enhanced beam search achieved better performance than with the original beam search in terms of CIDEr-1, CIDEr-5 and CIDEr-10 scores, with an average improvement of 1.8, 2.3 and 2.3, respectively; and (2) our clustering algorithm achieved high performance on six metrics and outperformed two variant clustering algorithms. Moreover, our user study with 24 participants shows that COOK can help developers accomplish programming tasks faster and pass more test cases, and the participants confirm that clusters and summaries indeed help them understand and select the correct API sequence recommendations.

Communicated by: Martin Monperrus

✉ Xin Peng
pengxin@fudan.edu.cn

Keywords API · Recommendation · Deep learning · Encoder-decoder · Post-processing

1 Introduction

Developers often need assistance in finding relevant code snippets or APIs to accomplish their development tasks. They may either describe their task needs in natural language queries or write some partial code fragments, based on which a recommendation system recommends relevant code snippets or APIs. In this work, we focus on the scenario in which *developers want the recommendation of a sequence of APIs for the partial code they have developed.* This scenario can be supported by code search techniques, API usage pattern mining techniques or deep learning-based code recommendation techniques.

Code search techniques (Hill and Rideout 2004; Luan et al. 2019; Ai et al. 2019; Kim et al. 2018; Krugler 2013) encode the provided partial code in some feature space (e.g., variable usage features and token features) and search the code in a large code base that is similar to the provided partial code in the feature space. The assumption is that developers are mostly aware of what to do, but need assistance in some part(s) of the big picture. However, when developers provide little partial code, the returned code snippets vary greatly in the way of implementations (such as the APIs used in the code snippets). Even though the returned code snippets may contain the needed API sequences, it is not an easy task to identify them from the often abundant code snippets.

API usage pattern mining techniques (Fowkes and Sutton 2016; Wang et al. 2013; Zhong et al. 2009; Nguyen et al. 2009, 2012) offer API recommendations at an abstract pattern level, which is relatively intuitive to understand. API usage patterns can be manually constructed or automatically mined. Mining API usage patterns is preferable, because it can better model API usage complexity and diversity than constructing API usage patterns manually. However, these techniques cannot recommend any code outside of the mined patterns and the number of mined patterns are often limited to a few hundreds (Luan et al. 2019).

In recent years, deep learning-based techniques (Chen et al. 2019a; Dam et al. 2016; Yan et al. 2018; Nguyen et al. 2018; Li et al. 2018; Terada and Watanobe 2019; Yang et al. 2019a, b, c) have been proposed for code recommendation and achieve good performance. These approaches encode source code into a program representation (e.g., code tokens, abstract syntax tree, or control flow graphs) and then apply deep neural networks (e.g., Recurrent Neural Network, Pointer Network or Tree-Structured Long Short-Term Memory Network) to learn the code semantics for code recommendation. While these approaches currently only focus on recommending one API but not an API sequence, they can be adapted for API sequence recommendation by using the encoder-decoder framework combined with beam search.

However, are deep learning models sufficient to recommend API sequences of high quality? In fact, some hidden assumptions deserve deeper considerations beyond more advanced models, but they have been largely overlooked in the current accuracy contest. In particular, the space of potential API sequence recommendations is quite open, and can have an exponential growth with the number of recommended APIs, which amplifies the impact of overlooked hidden assumptions.

On the one hand, beam search is a simple greed strategy to select API suggestions at each decoding step based on only the API suggestion probabilities of consecutive decoding steps. Here the assumption is that API suggestion probabilities determined by the encoder-decoder model are sufficient to guide the beam search process. However, when developers

select the API suggestions, they consider other aspects (e.g., program syntax) beyond just probability-based API rankings. The black box nature of encoder-decoder models actually makes it challenging to incorporate these aspects into the neural networks and control what the model outputs beyond probability distributions.

On the other hand, API sequence recommendations are presented in a probability-based ranked list. Again, the assumption here is that the ranking by the probabilities is sufficient for developers to make the selection. However, guided by only API suggestion probabilities, beam search usually outputs many highly overlapping API sequences in the top positions, which ignores the openness and diversity of the potential recommendations. Openness means that the intentions of developers are open. For example, when a developer declares a file object, we cannot accurately determine his intention because he may want to read contents from a file, or delete a file, or write contents into a file. Diversity means that the solutions and implementations are diverse. Since the intentions of developers are open, different intentions need different solutions and implementations. This makes the selection and comparison of diverse solutions very ineffective, as other solutions are pushed down by their low probabilities. Furthermore, to make an informed selection, developers need to understand the commonalities and differences among different solutions. However, probabilities have nothing to do with such commonalities and differences.

Inspired by the above insights, we take a complementary perspective to improve the recommendation of API sequences for the partial code. Since APIs are usually used together with control units (such as the API *java.io.File.exists* is usually used together with the control unit *if*), in our work, we take into consideration the program language keywords for control structures (e.g., for, while, and if). Therefore, the API sequence recommendations of our approach are API sequences with control structures. Our approach does not attempt to further enhance the already very complex encoder-decoder models. Instead, we propose an approach named COOK that post-processes the API sequence recommendations for any encoder-decoder models with three strategies. First, we inject two aspects of heuristics (i.e., program syntax and recommendation diversity) into the beam search process to prune unsatisfied API suggestions. Unsatisfied API suggestions mean that API suggestions at each decoding step that violate the program syntax or API sequence recommendations that have a high overlap. Second, we develop a hierarchical clustering algorithm that combines text similarity and API embedding similarity for grouping similar API sequence recommendations into clusters. Third, to help developer understand the API sequence recommendations, we provide a summary for each cluster based on the common code structure and the functionality description of the key APIs in the API sequence recommendations.

We have instantiated and implemented three different encoder-decoder models (i.e., Seq2Seq, Transformer and GGNN2Seq models) for API sequence recommendation. We have also conducted extensive experiments to evaluate the effectiveness of our approach. Our evaluation results have shown that: (1) all the three encoder-decoder models with our heuristic-enhanced beam search achieved better performance than with the original beam search in terms of CIDEr-1, CIDEr-5 and CIDEr-10 scores, with an average improvement of 1.8, 2.3 and 2.3, respectively; and (2) our clustering algorithm achieved high performance on six metrics and outperformed two other variant clustering algorithms. We also conducted a user study with 24 master students on 6 programming tasks to demonstrate that COOK can help students accomplish the programming tasks faster and pass more test cases. In addition, our interview with students using COOK confirmed that the clusters and summaries are kindly indeed help them understand and select the correct API sequence recommendations.

In summary, this work makes the following contributions:

– We propose an approach, named COOK, to enhance encoder-decoder models for API sequence recommendation with three post-processing strategies, i.e., heuristic-enhanced beam search, clustering API sequence recommendations, and summarizing API sequence recommendations.
– We instantiate and implement three different encoder-decoder models for API sequence recommendation.
– We conduct extensive experiments, including automatic quantitative comparison experiments and a user study, to evaluate the effectiveness of post-processing strategies of COOK.

2 Ingredients from Deep Learning

2.1 Program Representation for Encoder-Decoder Models

We introduce the program representation of the three instances of the encoder-decoder models that our approach is built upon. The three models are a Seq2Seq model, a Transformer model and a GGNN2Seq model, which will be introduced later.

The output of these models are the same, i.e., API sequences. The inputs are however different. The input of the Seq2Seq and Transformer models is API sequences, whereas the input of the GGNN2Seq model is API graphs. The API graph that we define is a directed graph that represents the program structure. Each node in an API graph represents a control unit (e.g., *if* and *while*) or an (labeled) API invocation, and each edge represents a type of flow (e.g., control flow or data flow) between two nodes. The API graph encodes four kinds of flow types, i.e., control flow, data flow, control and data flow, and special flow. A control flow (a.k.a. type c flow) means that there is only control flow between two nodes. A data flow (a.k.a. type d flow) means that there is only data flow between two nodes. A control and data flow (a.k.a. type cd flow) means that there are both control and data flow between two nodes. A special flow (a.k.a. type s flow) means that one of the two nodes is a node representing a "hole" to be filled, i.e., the code to be recommended.

We also introduce special placeholders into an API graph to reflect the structure of control units. Specifically, we use **Condition** node to represent the start of a condition expression, **Then** and **Body** nodes to represent the start of the body of a decision-making (e.g., *if*) and loop (e.g., *while*) control unit respectively, and **Out_control** node to represent the start of the APIs out of the control unit.

The API graph we adopt in our work is borrowed and referenced from the API context graph in Chen et al. (2021). Thus, as in Chen et al. (2021), each edge is given an obvious and unique type in our API graph. However, the type of an edge is not labeled in the API usage graph in Nguyen et al. (2009). In addition, when a program contains a hole, our API graph is still a connected graph containing a hole node, but the API usage graph in Nguyen et al. (2009) is not a connected graph.

Take the code in Fig. 1 as an example. This code is to read contents from a file line by line and compute the hash code of each line. The code at lines 3/4/5/6/10 are used to read contents from a file line by line. The code at line 2 is used to create a list to store the hash code of each line. The code at line 7 is used to declare a variable to store the hash code. The *$hole$* at line 8 is the position that the developer wants to get API sequence recommendations to help him compute the hash code of the line and then add it into the list. To recommend API sequences with deep learning models, we should convert the code into some kind of code representation as an input. For the Seq2Seq and Transformer models,

```
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:          }
10:         br.close();
11:         return result;
12: }
```

Fig. 1 Code Example

the input code representation is the input API sequence in the code which is obtained by traversing the API graph shown in Fig. 2 according to depth-first order and control flow before the hole. For the GGNN2Seq model, the input is the API graph itself.

2.2 Seq2Seq Model

In the Seq2Seq model (Bahdanau et al. 2015), given an API sequence $x_1, x_2, ..., x_n$ as input (where x_i (i = 1, 2, ..., n) is an API call or a control unit), the encoder first sequentially embeds (a.k.a. maps) x_i into a vector through the embedding layer and then each vector is

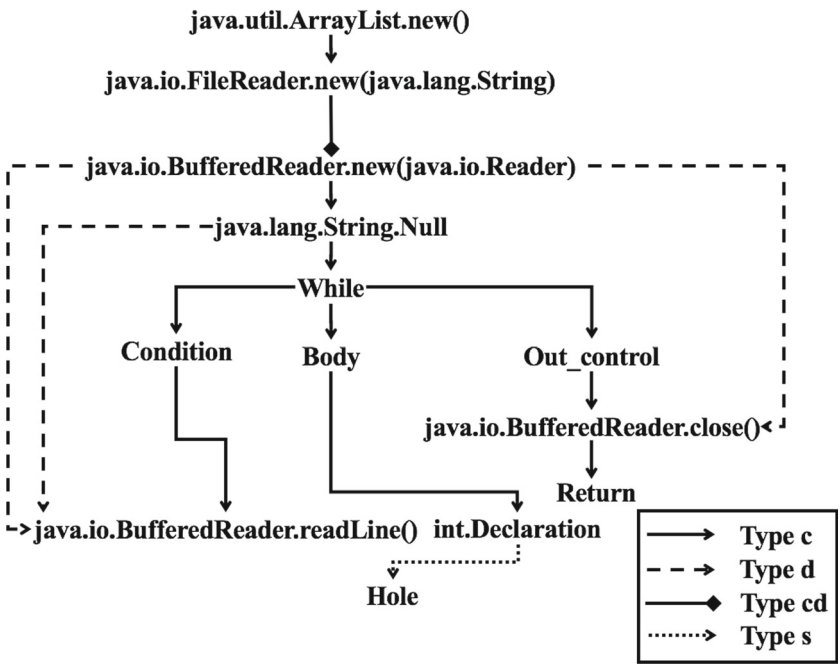

Fig. 2 API Graph

fed into the gated recurrent units (GRUs) one by one to generate the hidden states. Based on the hidden states and the attention mechanism, the input API sequence is represented as a context vector c. The decoder then generates the API sequence recommendations step by step. At each step t, the decoder generates API suggestions based on the current context vector c_t, the last hidden state h_{t-1} of the GRUs in the decoder as well as the vector representation of the last chosen API suggestion y_{t-1} through the embedding layer.

2.3 Transformer Model

In the Transformer model (Vaswani et al. 2017), given an API sequence $x_1, x_2, ..., x_n$ as input, the encoder first embeds x_i into a vector through the input embedding layer and then passes these vectors through the following layers (i.e., the positional encoding and a block including multi-head attention, add&norm layer, and feed forward layer) to generate a vector representation. The vector representation is then fed into the decoder, whose structure is similar to the encoder, to generate API sequence recommendations step by step.

2.4 GGNN2Seq Model

In the GGNN2Seq model (Li et al. 2016), given an API graph as input, the API graph is processed as a set of nodes and edges. The encoder first embeds the label of each node into a vector, which is then used as the initial vector of the node annotation of GGNNs, through the embedding layer. Then the nodes and edges are fed into the GGNNs to generate the vector representation of the API graph. The vector representation of the API graph is set as the initial hidden state of the GRU in the decoder for decoding. At each decoding step t, the decoder generates API suggestions based on the last hidden state h_{t-1} of the GRU and the vector representation of the last chosen API suggestion y_{t-1} through the embedding layer.

2.5 Beam Search

Encoder-decoder models usually apply the beam search to generate sequences at the decoding step, which is a natural choice to generate API sequences. Instead of choosing one API suggestion y_t with the highest probability, K suggestions are chosen at each decoding step t, typically the K most probable suggestions. After each step, the K API suggestions are combined with K API sequences from the previous step to form a set of K^2 candidates. Out of the candidates, K API sequences with the highest probabilities are generated, which are then used for the next step. A common way to compute the probability of a generated API sequence is as follows.

$$P(Y) = \frac{\sum_{t=1}^{T} log\, P(y_t|y_{1:t-1}, X)}{T} \tag{1}$$

where T is the total number of steps and $log\, P(y_t|y_{1:t-1}, X)$ is the log-likelihood of generating y_t at the t-th step.

For example, for the Seq2Seq model, when generating K API sequences with the beam search at the decoding step t, we choose K API suggestions with the highest probabilities at the decoding step t instead of just choosing one API suggestion with the highest probability for each candidate API sequence at the decoding step $t-1$. Each API suggestion that we choose at the decoding step t is then appended at the end of its candidate API sequence at the decoding step $t-1$. Since there are K candidate API sequences at the decoding step $t-1$, we can obtain K^2 candidate API sequences at the decoding step t. Out of the K^2 candidate API sequences, we preserve the K API sequences with the highest probabilities

according to (1). At the decoding step $t + 1$, the last hidden state at the decoding step t of each candidate API sequence, the current context vector at the decoding step $t + 1$ of each candidate API sequence, and the API suggestion that we choose at the decoding step t are used accordingly to generate the next K API suggestions of each candidate API sequence at the decoding step $t + 1$. The decoding process of an API sequence stops when the length of the API sequence reaches the max length or the end of the API sequence is EOS (denoting end of an API sequence). The overall processes of the Transformer and GGNN2Seq models to generate K API sequences with the beam search are similar to the process of the Seq2Seq model. However, the ways of computing the next states in each model are different.

3 Cooking for API Sequence Recommendation

In this section, we present the details of our approach, which is referred to as COOK. COOK is designed based on the idea that API sequence recommendation requires more than deep learning. That is to say, post-processing on the raw ingredients provided by deep learning is critical to generating high-quality API sequence recommendations.

There are two phases in COOK. One is the model training phase, applying one of those three encoder-decoder models so as to generate raw API suggestions at the decoding step. The other is the post-processing phase, which includes multiple post-processing strategies designed to improve the quality of the API sequence recommendations and to help developers to distinguish and understand the API sequence recommendations. There are three strategies, i.e., heuristic-enhanced beam search, clustering API sequence recommendations, and summarizing API sequence recommendations. We present the details of each strategy in the following sections.

3.1 Heuristic-Enhanced Beam Search

We propose several code-specific heuristics to enhance the original beam search (as introduced in Section 2.5) so that unsatisfied API suggestions at decoding steps are pruned. The underlying idea of these heuristics is to take program syntax (i.e., syntax of control units and variable accessibility) and recommendation diversity into account; i.e., API suggestions at each decoding step that violate program syntax or API sequence recommendations that have a high overlap are unsatisfied API suggestions that should be pruned.

3.1.1 Syntax-Oriented Heuristics

Program syntax is critical for API sequence recommendation. If an API sequence violates the program syntax, it will bring difficulty for the developers to understand the API sequence and the developer will waste time locating and fixing the program syntax error of the API sequence. Since the API sequence that we recommend contains both control units and APIs (e.g., special placeholders, special APIs, API method calls and API field accesses), we need to guarantee the correctness of the program syntax of the control units and APIs. For each control unit, we should make sure that its syntactical structure is correct. For example, *Condition* must come after a control unit to represents the condition branch of a control unit. For another example, the API suggestion *ElseIf* among API suggestions at a decoding step is invalid when there is no *If* structure in the program, because *ElseIf* without an *If* structure as the root is not syntactically correct. For special placeholders (i.e., *Condition*, *Then*, *Body* and *Out_control*) that we introduce into the structure of the control

units, we must make sure that the special placeholders match their corresponding control units. For example, a control unit must appear before *Condition*. For another example, the API suggestion *Body* among API suggestions at a decoding step is invalid when there is no *While* structure in the program. For special APIs (i.e., *Break*, *Continue* and *Return*), to guarantee the correctness of the program syntax when they are used together with control units, we must make sure that the special APIs satisfy the syntactical structure of control units. For example, *Continue* should be used at the end of the body of a loop, which means that *Out_control* should come after *Continue* when the control unit is a loop. For common APIs (e.g., API method calls and API field accesses), to avoid the program syntax error of the variable (object) accessibility, we should make sure that there exist variables or objects that meet the type of the receiver and parameters of an API. For example, the API suggestion *java.io.OutputStream.write(byte[])* among API suggestions at a decoding step is invalid and will cause an program syntax error when the program contains no object of the type *java.io.OutputStream*. Note that we focus on post-processing, thus new heuristics can be easily extended into our heuristic-enhanced beam search. However, in this work, we focus on the proposed heuristics based on the above mentioned analysis.

Therefore, we first propose a set of heuristic rules to ensure that an API suggestion is always syntactically correct. An API suggestion at a decoding step is preserved only if it satisfies the rules. The detailed syntax checking rules are as follows. Specifically, for control units, we define the following rules.

- If the last chosen API suggestion is a control unit (e.g., *If* and *While*), the current API suggestion must be *Condition* and other API suggestions should be pruned.
- If the current API suggestion is *ElseIf* or *Else*, it must match a previous *If* control unit. Otherwise, it is pruned.

For special placeholders (i.e., *Condition*, *Then*, *Body* and *Out_control*), we define the following rules.

- If the current API suggestion is *Condition* and the last chosen API suggestion is not a control unit, it is pruned. Otherwise, it is preserved.
- If the current API suggestion is *Then*, *Body* or *Out_control*, it must match a previous control unit. Otherwise, it is pruned.

For special APIs (i.e., *Break*, *Continue* and *Return*), we define the following rules.

- If the last chosen API suggestion is *Break* or *Continue*, the current API suggestion should be *Out_control*, *ElseIf* or *Else*. If the current API suggestion is *Out_control* and matches a previous control unit, it is preserved. If the current API suggestion is *ElseIf* or *Else* and matches a previous *If* control unit, it is preserved. Otherwise, it is pruned.
- If the last chosen API suggestion is *Return*, the current API suggestion should be *Out_control*, *ElseIf*, *Else* or *EOS* (denoting end of an API sequence). If the current API suggestion is *EOS* and *Return* is not in a control unit, it is preserved. If the current API suggestion is *ElseIf* or *Else* and matches a previous *If* control unit, it is preserved. If the current API suggestion is *Out_control* and matches a previous control unit, it is preserved. Otherwise, it is pruned.

Then, we propose a heuristic rule to check whether the current code context contains the variables or objects that meet the type of the receiver and parameters of the API suggestion at a decoding step. It ensures variable accessibility; i.e., all variables and objects for the receivers and parameters of API suggestions at each decoding step are accessible in the

code context. The variables and objects in the code context include the variables and objects declared or created before the recommendation position, and the variables and objects generated in the already generated API sequence till the last decoding step. If the return type of an API suggestion in the already generated API sequence till the last decoding step is not $void$, we consider it as an object of that return type. At the current decoding step, if an API suggestion is a static API method or field access, we do not check the type of its receiver, because the class can be directly used. Otherwise, we check whether there exists a variable or object that can match the type of the receiver. If there is no match for the receiver, this API suggestion is pruned. For each parameter in an API suggestion at the current decoding step, if the type of the parameter is a basic type (e.g., int and $double$) or $java.lang.String$, we do not check because this parameter can be concretized with a constant. Otherwise, we check whether there exists a variable or object that can match the type of the parameter. If there is no match for the parameter, this API suggestion is pruned. Note that we also consider the scope of the variables and objects when making a check. For example, when we check an API suggestion out of an *If* control unit, the variables and objects within the *If* control unit are not included in the code context.

The reasons why we use syntactical rules as heuristics are as follows. First, since the deep learning model is an implicit model that runs in a black-box way, it is difficult to enforce syntactical rules into the deep learning model explicitly. Thus, it is better to use syntactical rules as heuristics to enforce them in the post-processing process. Second, using syntactical rules as heuristics is more lightweight and flexible, which allows us to extend new heuristics in the future.

3.1.2 Diversity-Oriented Heuristics

Recommendation diversity is another important factor for API sequence recommendation. Take the two API sequence recommendations in Fig. 3 as an example. We can see that the second API sequence contains the first API sequence and these two API sequences reflect the same core semantics that invoke a method using the reflection mechanism. However, due to the uncertainty of the intention of developers, whether a developer needs to set the accessibility of a method depends on the method (s)he invokes. In addition, a developer may want to invoke a field or create an instance instead of invoking method using the reflection mechanism. Thus, it is better to just preserve one of the two API sequences, which can allow us to add another API sequence (which reflects the semantics of invoking a field or creating an instance) into the top-k API sequence recommendations to increase the diversity

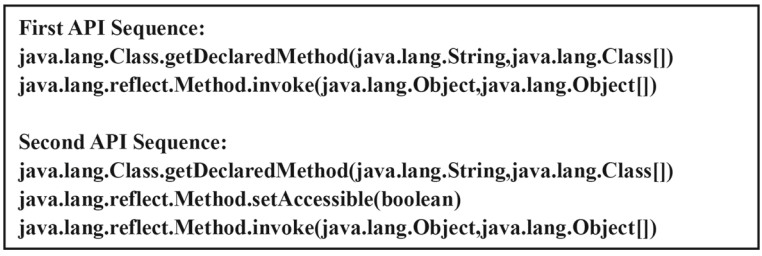

Fig. 3 Example for Recommendation Diversity

of recommendations. In this example, the second API sequence is preserved because it has a higher probability according to (1).

Therefore, we design a heuristic rule to increase the diversity of the final API sequence recommendations. If one API sequence recommendation contains another API sequence recommendation, we define that these two API sequences have an inclusion relationship. Before appending an API sequence into the final API sequence recommendations, we compare this API sequence with API sequences in the API sequence recommendations one by one. If two API sequences have inclusion relationship, we just preserve one of them. Currently, we preserve the one with the higher probability computed by (1) because the one with higher probability is considered more satisfiable with the current code context by the encoder-decoder model.

We modify the longest common subsequence (LCS) algorithm to judge whether two API sequences have inclusion relationship. Since we take into consideration the control units in an API sequence, we need to make sure that the syntax of the control units in the longest common subsequence is correct. If we cannot guarantee the correctness of the syntax of control units in the longest common subsequence, the program logic may be destroyed, which means that APIs maybe appear in the wrong control unit. Take the two API sequences in Fig. 4 as an example. The modification is that we add a judgment on the mapping of *Condition*, *Then*, *Body* and *Out_control* to its corresponding control unit. If we just apply LCS, the longest common subsequence is marked italic in Fig. 4. However, it is obvious that the italic *Out_control* does not belong to the italic *If* in the first API sequence, which causes the problem that the italic APIs $java.lang.String.length()$ and $java.lang.String.substring(int, int)$ appear in the first *If* control structure. Indeed, these two APIs belong to the second *If* control structure. Thus, we should add a judgment on the mapping of *Condition*, *Then*, *Body* and *Out_control* to its corresponding control unit in LCS. If *Condition*, *Then*, *Body* and *Out_control* are all mapped with their corresponding control units correctly, we can guarantee that the APIs in the longest common subsequence appear in the correct control units. Therefore, the two API sequences in Fig. 4 do not have inclusion relationship.

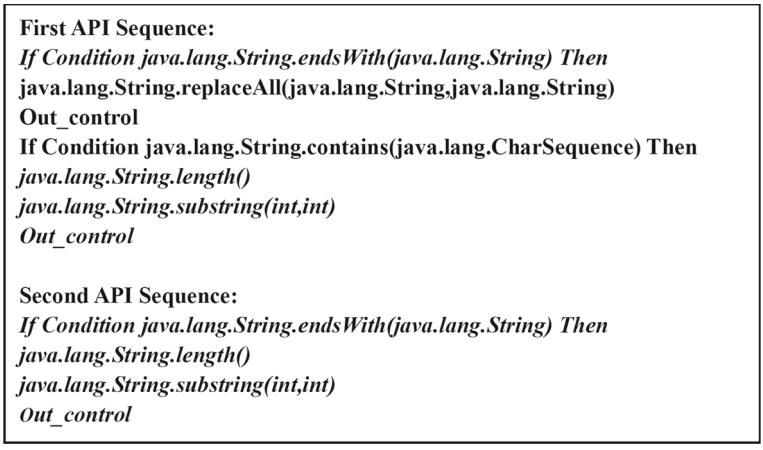

Fig. 4 Example for Modified LCS

3.2 Clustering API Sequence Recommendations

Given the API sequence recommendations from our heuristic-enhanced beam search, we cluster them into several clusters. The goal is to make it easier for developers to find similar API sequence recommendations, distinguish different API sequence recommendations and facilitate their selection. For example, there are two API sequence recommendations that use *FileOutputStream* to write contents into a file, one API sequence recommendation that uses *ZipInputStream* to process a zip file, and three API sequence recommendations that use *BufferedReader* to read contents from a file. These API sequence recommendations should be ideally grouped into three clusters. Without clustering, these API sequence recommendations are ordered by their probabilities, which makes it difficult for developers to find similar API sequence recommendations and distinguish different API sequence recommendations.

To this end, we apply a hierarchical clustering algorithm to cluster API sequence recommendations. Clustering requires a way of measuring the similarity between different API sequences. We combine text similarity and API embedding similarity to compute the distance of two API sequences.

Text similarity reflects how textually-similar two API sequences are. If the text similarity of two API sequences is zero, meaning that there is no common API in these two API sequences except for control units, these two API sequences should be clustered into different clusters, because we consider that these two API sequences reflect different semantics or solutions. If the text similarity of two API sequences is not zero, which means that there exist common APIs in these two API sequences and they are more or less semantically similar, we further take into consideration the API embedding similarity between these two API sequences. The underlying idea is that two API sequences should be put in the same cluster if their API embedding similarity is relatively high even if their text similarity is relatively low and two API sequences should be clustered into different clusters if their API embedding similarity is relatively low even if their text similarity is relatively high.

Take the three API sequences in Fig. 5 as an example. The text similarity of the first and second API sequences is 0.29, which is lower than the text similarity (0.33) of the second and third API sequences. Thus, if just use the text similarity, the second and third API sequences are clustered into the same cluster. However, the core semantics of the first and second API sequences are more similar, which are to process a string and return a substring. So, even though the text similarity of the first and second API sequences is lower, they should be clustered into the same cluster instead of clustering the second and third

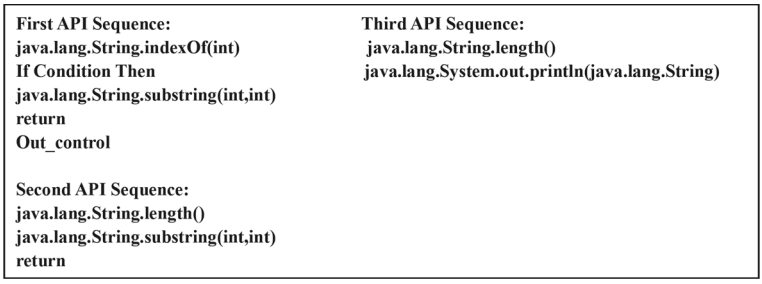

Fig. 5 Example for Combining Text Similarity with API Embedding Similarity

API sequences into the same cluster. Combining the text similarity and API embedding similarity can solve the above problem.

Based on the above observations, we define the distance of two API sequences Y and Y' used in the hierarchical clustering algorithm, written as $D(Y, Y')$, as follows.

$$D(Y, Y') = \begin{cases} 1 & text(Y, Y') = 0 \\ \frac{(1-ae(Y,Y'))+(1-(text(Y,Y'))}{2} & text(Y, Y') \neq 0 \end{cases} \quad (2)$$

where $ae(Y, Y')$ is the API embedding similarity of Y and Y'; and $text(Y, Y')$ is the text similarity of Y and Y'.

The text similarity is designed such that the higher the score, the more textually-similar they are. In this work, it is defined based on the longest common sequence of Y and Y'. That is, given two API sequences, to compute the text similarity, we apply the modified LCS algorithm in Section 3.1 to get the longest common sequence that contains at least one common API. The text similarity $text(Y, Y')$ is computed as follows.

$$text(Y, Y') = \frac{len(C)}{max(len(Y), len(Y'))} \quad (3)$$

where C denotes the longest common sequence that contains at least one common API, function $len()$ is to get the length of a sequence, and function $max()$ is to get the larger length.

Given two API sequences, to compute the API embedding similarity, we leverage the embeddings from the encoder-decoder model to represent an API sequence as a vector. Given an API sequence $Y = y_1, y_2, ..., y_n$, its vector representation $V(Y)$ is defined as follows.

$$V(Y) = \frac{\sum_{t=1}^{n} embed(y_t)}{n} \quad (4)$$

where $embed(y_t)$ is the vector representation of the API y_t that can be obtained from the embeddings of the encoder-decoder model, and n is the length of the API sequence. The API embedding similarity is computed based on the cosine similarity of these two vectors.

Given the similarity measure, we use the Silhouette Coefficient (Rousseeuw 1987) to select the clusters with the highest score as the final cluster results of the hierarchical clustering algorithm.

3.3 Summarizing API Sequence Recommendations

Given the API sequence recommendations in each cluster, we summarize them to help developers understand the "consensus" of each cluster.

As reported in Chen et al. (2019a), developers are often not familiar with recommended APIs, they just rely on the names of the recommended APIs to decide whether an API is what they expect. They hope that easy-to-understand explanations of recommended APIs can be provided to allow them to better choose the right API. We can see that developers already call for explanations of APIs in the single API recommendation task. Hence, it is more important to provide summaries in the API sequence recommendation task, which contains multiple APIs.

The main idea of the solution on summarizing API sequences is to first extract the longest common subsequence of API sequences (which reflects the common semantics (intention) of API sequences) in a cluster, then recognize the key APIs reflecting the key intention of the longest common subsequence, and finally generate the summary of the key APIs.

To generate a summary for each cluster, we first use the algorithm introduced in Section 3.1 to extract the longest common sequence of any pair of API sequence recommendations. Note that our approach is based on the assumption that the longest common sequence summarizes the "consensus" of the sequences. The API sequence recommendations in the same cluster contain common APIs and have similar semantics, so the longest common sequence can be used as the "consensus" to represent the similar semantics.

Then, we recognize the key APIs in the longest common sequence for summarization. Specifically, we construct a graph to reflect the data and control flows among APIs in the longest common sequence, which is then used in the PageRank algorithm (Brin and Page 1998) to recognize the key APIs. The control flow among APIs in the longest common sequence is exact, however, we cannot obtain the exact data flow among APIs in the longest common sequence because all APIs are represented in an abstract way. Thus, we analyze all possible data flows among APIs in the longest common sequence combined with the partial code based on type matching. If the type of a declared variable (or object) or the return type of an API matches the type of a receiver or a parameter in another API, we add a data flow between them. For example, a longest common sequence of a partial code is shown in Fig. 6, and the constructed graph is also shown in Fig. 6. The return type of *java.io.BufferedReader.readLine()* is *java.lang.String*, which matches the type of the parameter in the API *java.lang.StringBuilder.append(java.lang.String)*.

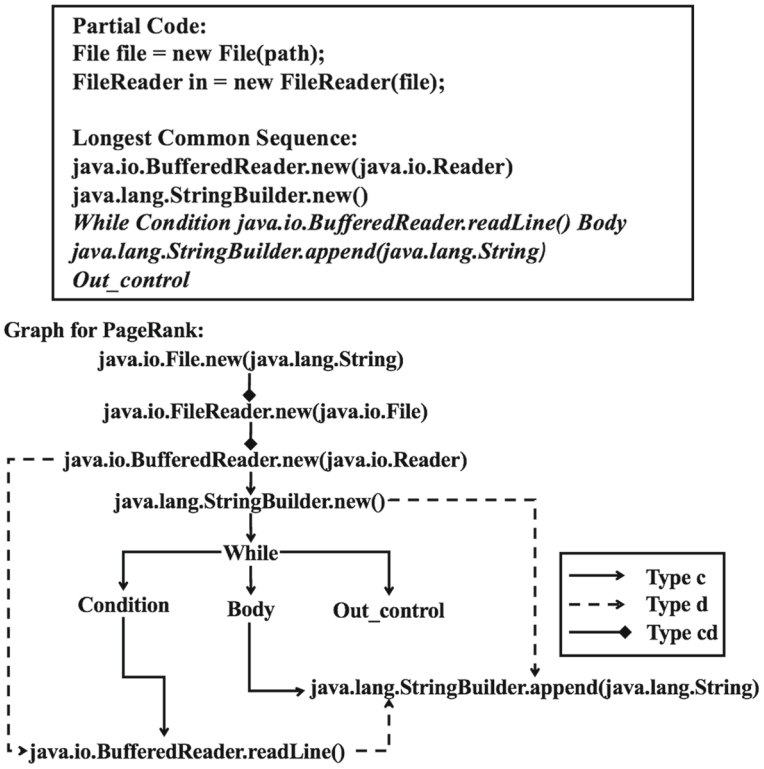

Fig. 6 Example for Recognizing Key APIs

Thus, there is a possible data flow between these two APIs. After applying the PageRank algorithm on the graph in Fig. 6, we obtain *java.io.BufferedRead er.readLine()* and *java.lang.StringBuilder.append(java.lang.String)* as two key APIs. As the two key APIs are in a control unit (i.e., a loop unit *While*), we also mark the control unit as a key API (is italic Fig. 6) to maintain the structural completeness.

Finally, for each identified key API, we extract the first sentence in its documentation as its summary. For a control unit, we define a summary template. For example, for a *While* control unit, we define its summary as "Depending on the result of *some condition*, do the following things iteratively: *do something*". "*some condition*" and "*do something*" will be replaced with the summary of the corresponding API in the condition and body of *While*. For example, the summary of the key APIs in Fig. 6 is "Depending on the result of reading a line of text, do the following things iteratively: append the specified string to this character sequence".

4 Evaluation

To evaluate the effectiveness of COOK, we conducted experiments to answer the following three research questions.

- **RQ1 (Effectiveness of Heuristic-Enhanced Beam Search)**: How is the quality of the API sequences recommended by our heuristic-enhanced beam search?
- **RQ2 (Effectiveness of Clustering Algorithm)**: How effectively can our clustering algorithm cluster API sequence recommendations?
- **RQ3 (Effectiveness of COOK in Real Tasks)**: How effectively can COOK help developers accomplish real programming tasks?

4.1 Implementation and Data Collection

We have developed an implementation of COOK for JDK 1.8, i.e., recommend API sequences that involve JDK APIs. We used JavaParser[1] to parse source code into ASTs (Abstract Syntax Trees) and leveraged Java reflection mechanism to identify JDK API invocations in source code for constructing API sequences and graphs. All three encoder-decoder models were implemented using TensorFlow 1.14. https://github.com/tensorflow/tensorflow.

We collected a large code corpus from GitHub for training. We crawled Java projects that had more than 1000 stars, and finally we obtained 1,914 Java projects. We randomly selected 90% of the Java projects as the training set and the remaining 10% of the Java projects as the validation set. Following prior works (Chen et al. 2019a; Nguyen and Nguyen 2015; Liu et al. 2018), for each method in a Java project, we constructed training and validation instances at each position in the method. A position is the position of an API in the target library (e.g., JDK library) or a control unit in the method. At a position, we removed some APIs. The removed APIs were used as ground truth and the remaining code was used as the incomplete code. In total, our training and validation sets had 9,133,429 and 1,456,829 instances, respectively.

For all the three encoder-decoder models, the embedding size was set to 300; the hidden size was set to 300 in the encoder and 600 in the decoder; the dropout was set to 0.75; the

[1] https://github.com/javaparser/javaparser/

learning rate was set to 0.005; the batch size was set to 256. During training, if the loss does not decrease in five successive epochs on validation set, the training process ends and the model with the lowest loss is used as the final model.

4.2 Effectiveness of Heuristic-Enhanced Beam Search (RQ1)

To evaluate the effectiveness of our heuristic-enhanced beam search, we compared the quality of the API sequences recommended by the three encoder-decoder models with and without our heuristic-enhanced beam search on six test projects. The API sequence recommendations of the models equipped with the original beam search are generated in the way we introduced in Section 2.5, in which we set k to 10. The API sequence recommendations of the models equipped with our heuristic-enhanced beam search are generated by introducing the syntax-oriented heuristics and diversity-oriented heuristics into the original beam search at each decoding step to prune unsatisfied API suggestions.

Effectiveness Metrics We employed the CIDEr[2] score (Vedantam et al. 2015) as the indicator of the quality of API sequence recommendations. CIDEr was originally proposed to compute the similarity between two sentences based on a weighted combination of n-grams while taking the frequency of words into consideration.

Zohar and Wolf (2018) focused on program synthesis task and employed the CIDEr score to compute the similarity between the synthesized program and the target program. Following this work, we employed the CIDEr score to compute the similarity between an API sequence recommendation and the API sequence ground truth. The higher the CIDEr score, the more similar the two API sequences. Thus, a 10% improvement of CIDEr means that the API sequence recommendation is 10% more similar to the API sequence ground truth, which indicates that the API sequence recommendation may contain more ordered APIs that appear in the API sequence ground truth.

As we want to recommend the top-k API sequence recommendations for each test instance, we defined the CIDEr-k score to compute the highest CIDEr score. For each of the top-k API sequence recommendations of each test instance, we computed its CIDEr score, and the highest CIDEr score was used as the final CIDEr score of these top-k API sequence recommendations.

Test Projects We chose six open-source Java projects: Galaxy https://github.com/puniverse/galaxy, JGit https://github.com/eclipse/jgit, Froyo-Email https://github.com/Dustinmj/Froyo_Email, Grid-Sphere https://github.com/brandt/GridSphere, Itext https://github.com/itext/itextpdf and Log4j https://github.com/apache/log4j to construct the test instances. These projects have been widely used as test projects in previous research on API recommendation (e.g., Chen et al. 2019a; Nguyen et al. 2015, 2016; Liu et al. 2018). Note that they were not included in the training set or validation set. The test instances were constructed in the same way as the training and validation instances. Furthermore, we filtered duplicated test instances. If the input API sequences, API graphs and the ground truth of two test instances were the same, the two test instances were considered to be duplicated. In addition, we filtered test instances whose length of the API sequence ground truth are less than 4 for two reasons. First, we expect to help developers finish tasks with long API sequences. When the length of an API sequence is short, developers may tend to write

[2] https://github.com/vrama91/cider

code by themselves or use tools that recommend the next API iteratively. Second, the computation of the CIDEr score involves 1-ngram, 2-ngram, 3-ngram and 4-ngram. Thus, as in Zohar and Wolf (2018), we focus on long API sequences.

Results The results of CIDEr-*k* score on the constructed test instances are reported in Table 1. The first column lists the project name as well as the number of test instances

Table 1 CIDEr-k scores of encoder-decoder models with and without heuristic-enhanced beam search

Project	Model	CIDEr-1	CIDEr-5	CIDEr-10
	Seq2Seq+OBS	12.6	20.5	23.6
	Seq2Seq+HBS	**16.0**	**24.9**	**28.3**
Galaxy	Transformer+OBS	9.8	16.6	18.1
(167)	Transformer+HBS	**12.1**	**18.2**	**19.9**
	GGNN2Seq+OBS	13.5	20.3	23.2
	GGNN2Seq+HBS	**14.9**	**25.1**	**28.8**
	Seq2Seq+OBS	12.6	21.8	25.0
	Seq2Seq+HBS	**13.5**	**23.6**	**26.5**
JGit	Transformer+OBS	12.1	21.7	25.4
(1,298)	Transformer+HBS	**14.0**	**23.2**	**25.7**
	GGNN2Seq+OBS	12.9	22.5	25.5
	GGNN2Seq+HBS	**15.4**	**25.5**	**28.3**
	Seq2Seq+OBS	13.8	25.1	28.1
	Seq2Seq+HBS	**16.6**	**27.2**	**30.4**
Froyo-Email	Transformer+OBS	13.8	24.5	28.2
(504)	Transformer+HBS	**16.0**	**28.7**	**31.6**
	GGNN2Seq+OBS	15.8	27.7	29.7
	GGNN2Seq+HBS	**18.4**	**32.7**	**35.0**
	Seq2Seq+OBS	17.4	27.1	29.3
	Seq2Seq+HBS	**17.9**	**27.8**	**30.3**
Grid-Sphere	Transformer+OBS	18.4	27.7	29.9
(574)	Transformer+HBS	**18.8**	**28.3**	**31.8**
	GGNN2Seq+OBS	17.3	28.0	29.7
	GGNN2Seq+HBS	**20.3**	**31.5**	**33.5**
	Seq2Seq+OBS	18.6	27.4	29.9
	Seq2Seq+HBS	**20.9**	**29.3**	**31.6**
Itext	Transformer+OBS	19.7	27.2	29.3
(1,546)	Transformer+HBS	**20.4**	**28.5**	**30.3**
	GGNN2Seq+OBS	18.8	28.0	30.1
	GGNN2Seq+HBS	**22.8**	**32.4**	**34.6**
	Seq2Seq+OBS	16.5	**22.8**	**25.6**
	Seq2Seq+HBS	**17.2**	22.7	24.7
Log4j	Transformer+OBS	16.3	23.4	25.0
(696)	Transformer+HBS	**17.5**	23.4	**25.5**
	GGNN2Seq+OBS	18.2	24.9	27.3
	GGNN2Seq+HBS	**18.4**	**25.2**	**28.2**

in each project. The second column shows the encoder-decoder models equipped with the original beam search (OBS) and our heuristic-enhanced beam search (HBS). The remaining columns report the CIDEr-1, CIDEr-5 and CIDEr-10 score.

We can see that the GGNN2Seq model with the original beam search achieved higher CIDEr-*k* scores than the Seq2Seq and Transformer models with the original beam search in most cases, and the improvement ranged from 0.5 to 3.7. The Transformer model achieved better performance than the Seq2Seq model in NLP tasks, but the Transformer model with the original beam search achieved lower CIDEr-*k* score than the Seq2Seq model in most cases. The improvement of the GGNN2Seq model is credited to the consideration of program structure. However, the improvement of the Transformer model over the Seq2Seq model is not as we expected. These results indicate that the advances in encoder-decoder models are not applicable to all cases every time, unless the advances take into consideration of the features of a specific problem (e.g., GGNN2Seq model takes the program structure into consideration). Therefore, we take a complementary perspective to add post-processing strategies for API sequence recommendation.

In almost all the cases, all encoder-decoder models with our heuristic-enhanced beam search achieved higher CIDEr-*k* score than with the original beam search, and the improvement ranged from 0.2 to 5.6. In some cases, the improvement of our heuristic-enhanced beam search was even higher than using more advanced encode-decoder models. For example, the CIDEr-1 scores of the Seq2Seq model and the Transformer model with our heuristic-enhanced beam search on Froyo-Email were higher than that of the GGNN2Seq model with the original beam search. It shows that our heuristic-enhanced beam search is effective to all deep learning models and almost all the cases. Besides, our heuristic-enhanced beam search can be extended with more heuristic rules and combine domain-specific knowledge. As the GGNN2Seq model achieves the best performance in most cases, we adopt the GGNN2Seq model as the representative model in RQ2 and RQ3.

Since all encoder-decoder models with our heuristic-enhanced beam search achieved higher CIDEr-*k* score than with the original beam search, we further conducted an experiment to evaluate the false positive rate of HBS as opposed to OBS. We define that API sequence recommendations of OBS that violate the heuristic rules (i.e., syntax-oriented heuristics and diversity-oriented heuristics), are not included in the API sequence recommendation of HBS, and the CIDEr scores are lower than the highest CIDEr score in the API sequence recommendation of HBS as false positives. The false positive rate is computed as the ratio of the number of false positives to the total number of API sequence recommendations of OBS. The results are reported in Table 2.

We can see that the false positive rate is high for each testing project of each encoder-decoder model. This indicates that our heuristic-enhanced beam search can effectively reduce the false positives in the decoding step. We can also see that even with better code representation (as in GGNN2Seq), our heuristic-enhanced beam search can still make sense though the false positive rate is lower than those of the other two models. This does not mean that there is no need to apply heuristic-enhanced beam search when the code representation becomes better. Conversely, when the code representation becomes better, we should exploit other suitable heuristic rules and try to introduce knowledge into to the heuristic-enhanced beam search to enhance the reasoning ability.

Furthermore, we conducted an experiment to evaluate whether the length of an API sequence to be recommended will affect the effectiveness of our heuristic-enhanced beam search. We divided the test instances in all the six test projects into 18 subsets

Table 2 False positive rate (%)

Project	Seq2Seq	Transformer	GGNN2Seq
Galaxy	72.5	73.8	62.4
JGit	64.8	68.0	60.3
Froyo-Email	64.0	66.6	58.3
Grid-Sphere	63.9	68.2	57.5
Itext	64.3	69.1	61.1
Log4j	66.9	69.5	56.4

according to the length of an API sequence (i.e., 4 to 20, and over 20). For each subset, we evaluated the CIDEr-10 score for each model with OBS and HBS separately. The results are reported in Table 3.

We can see that our heuristic-enhanced beam search is insensitive to the sequence length. In most cases, the model with our heuristic-enhanced beam search achieved higher CIDEr-10 score than with original beam search. This is because that the heuristic rules defined in our heuristic-enhanced beam search pay attention to characteristics of the source code itself. The sequence length has no relation with the characteristics of the source code.

Table 3 CIDEr-10 scores for different sequence lengths (%)

Length	Seq2Seq+OBS	Seq2Seq+HBS	Transformer+OBS	Transformer+HBS	GGNN2Seq+OBS	GGNN2Seq+HBS
4	**26.5**	25.5	**25.8**	25.3	28.5	**30.0**
5	35.4	**35.8**	36.0	**36.2**	35.4	**41.5**
6	25.5	**27.4**	25.0	**26.9**	26.3	**32.0**
7	**27.1**	26.7	24.2	**26.0**	27.8	**29.5**
8	23.2	**25.4**	22.7	**24.6**	24.1	**26.8**
9	23.3	**25.8**	24.0	**24.5**	24.7	**26.2**
10	27.9	**32.3**	28.7	**31.4**	30.0	**33.4**
11	25.0	**26.6**	24.8	**28.2**	24.6	**28.8**
12	24.2	**27.8**	22.5	**25.4**	25.6	**27.8**
13	23.4	**27.4**	24.6	**25.5**	24.7	**28.6**
14	24.3	**27.2**	24.0	**25.5**	24.5	**28.0**
15	26.9	**29.3**	28.0	**29.5**	27.4	**29.1**
16	25.4	**28.6**	26.4	**26.6**	24.8	**29.3**
17	28.1	**28.6**	28.6	**29.1**	27.0	**29.8**
18	25.2	**29.5**	26.3	**28.4**	24.1	**28.0**
19	26.0	**28.5**	24.5	**26.0**	27.0	**28.7**
20	29.7	**33.8**	**32.0**	31.9	28.8	**31.2**
20+	24.6	**26.1**	23.5	**25.2**	23.8	**26.5**

4.3 Effectiveness of the Clustering Algorithm (RQ2)

To evaluate the effectiveness of our clustering algorithm, we compared our clustering results with manual clustering results.

Evaluation Metrics We employed six metrics, i.e., Adjusted Rand Index (ARI) (Hubert and Arabie 1985), Normalized Mutual Information (NMI) (Nguyen et al. 2010), Homogeneity (HOM), Completeness (COM), V-measure (V-M) (Rosenberg and Hirschberg 2007), and Fowlkes-Mallows Index (FMI) (Fowlkes and Mallows 1983), as used in prior work (Huang et al. 2018), as the indicators to measure the performance of our clustering algorithm. Since our clustering algorithm combines the text distance and API embedding distance (TD+AED) for hierarchical clustering, we also compared it with two variants, i.e., the first variant clustering algorithm only uses the API embedding distance (AED) and the second variant clustering algorithm only uses the text distance (TD).

ARI measures the similarity between two clustering labels in a statistical way. NMI measures the mutual information between two clustering labels. HOM is the proportion of clusters containing only members of a single class. COM is the proportion of all members of a given class that are assigned to the same cluster. V-M is the harmonic mean of HOM and COM. FMI is the geometric mean of the pairwise precision and recall. For all above six metrics, the higher the value, the better the clustering performance (Huang et al. 2018).

Test Instances and Ground Truth Construction we randomly selected 30 test instances (5 test instances from each project) from all the test instances in RQ1. As we need to manually label clusters of each test instance to construct the ground truth and it is time-consuming to label a large amount of test instances, we randomly selected 30 test instances.

For each test instance, we obtained its top 10 API sequence recommendations through GGNN2Seq model with our heuristic-enhanced beam search. Two of the authors individually manually clustered the top 10 API sequence recommendations of each test instance. For any two of the top 10 API sequence recommendations for each test instance, we checked whether the two authors clustered these two API sequences into the same cluster, and used Cohen's Kappa coefficient (McHugh 2012) to compute the score of the agreement between the two authors. The original agreement between the two authors was substantial, achieving a Cohen's Kappa coefficient of 0.76. Further, based on the original clusters of the top 10 API sequence recommendations clustered by the two authors, if the clusters were not the same, a third author would be involved to have group discussions to reach a total agreement.

Results The results are presented in Table 4. Our clustering algorithm using TD+AED achieved the best performance on all the six metrics and achieved a significantly higher score (from 0.82 to 0.92) on the six metrics than the two variants. These results demonstrate that our clustering is effective in clustering API sequence recommendations.

4.4 Effectiveness of COOK in Real Tasks (RQ3)

To evaluate how effectively COOK helps developers in practice, we developed IntelliJ IDEA plugins for the GGNN2Seq model with the original beam search and COOK respectively, and conducted a user study. We asked two groups of participants to complete a set of programming tasks with these two plugins. In addition, we asked another group of participants

Table 4 Clustering performance

Method	ARI	NMI	HOM	COM	V-M	FMI
AED	0.37	0.63	0.58	0.86	0.63	0.59
TD	0.66	0.81	0.88	0.82	0.81	0.78
TD+AED	**0.82**	**0.84**	**0.88**	**0.92**	**0.87**	**0.90**

to complete the same set of programming tasks with the standard IntelliJ IDEA. Note that the standard IntelliJ IDEA itself provides standard code-completion feature.

Tasks in the User Study Following a prior work (Chen et al. 2019a), we obtained programming tasks from Stack Overflow. First, we obtained the top 750 posts with tag "Java", ordered by votes. Second, we excluded posts about concept explanation, bug fixes, performance comparisons and difference explanation as they were not about API usages. Third, we checked whether the answer with most votes of each post was JDK API sensitive and had at least four API invocations. If yes, the corresponding post was preserved. Finally, we obtained 31 posts. We randomly selected six posts as programming tasks from the 31 posts. For each task, we designed several test cases (i.e., 5 test cases on average). Following previous works (Chen et al. 2019a, 2021) that include a user study, we randomly selected 6 tasks to avoid bias. We do not include more tasks because participants will become slack when there are too many tasks to finish, which will affect the results of the experiment. In addition, the number of tasks used in Chen et al. (2019a, 2021) is also no more than 6. The six tasks are as follows:

- T1: Read the contents of a file. https://stackoverflow.com/questions/4716503/.
- T2: Extract a substring using regex. https://stackoverflow.com/questions/4662215/.
- T3: Get the filenames of all files in a folder. https://stackoverflow.com/questions/5694385/.
- T4: Convert a serializable object to a byte array. https://stackoverflow.com/questions/2836646/.
- T5: Create an instance by class name and constructor. https://stackoverflow.com/questions/6094575/.
- T6: Generate an MD5 hash of a given string. https://stackoverflow.com/questions/415953/.

Though the tasks were chosen randomly as in previous work, we find the length of API sequence to be recommended and the cyclomatic complexity of the chosen tasks are different, which can simulate the scenarios of different situations.

Participants in the User Study We recruited 24 master students, majoring in software engineering, from our school. We asked the 24 participants to complete a pre-study survey to evaluate their experience and capability in Java programming. Based on the results of the survey, we divided the 24 participants into three groups G1, G2, and G3, so that the overall capabilities of G1, G2 and G3 were at an equivalent level.

We assigned G1 to use the plugin of the GGNN2Seq model with the original beam search, and assigned G2 to use the plugin of COOK. We assigned G3 to use the standard IntelliJ IDEA which provides the standard code-completion feature. For each task, we provided the first line of code of the answer to make sure that even if a participant had no solution to solve a task, participants in G1 and G2 could use plugins for help based on

the first line of code. If the participants had their own solutions to solve a task, they were allowed to ignore and delete the first line of code. Participants were asked to accomplish the six tasks and they were not allowed to search code and solutions from the Internet, and they could use plugins we provided to obtain top 10 API sequence recommendations. Each participant was given 20 minutes to accomplish each task, and they were required to submit their current implementation of each task once the time was up.

Evaluation Metrics We defined task completion time and test pass rate as the indicators to measure the performance of the two plugins. The task completion time is the time that a participant takes to accomplish a task. The test pass rate is the percentage of test cases that are passed.

Results The results are reported in Tables 5 and 6 respectively. We can see that participants in G3 accomplished some tasks faster (i.e., T1, T3 and T4) and passed more test cases (i.e., T1, T3, T5 and T6) than participants in G1, and participants in G1 accomplished some tasks faster (i.e., T2, T5 and T6) and passed more test cases (i.e., T2 and T4) than participants in G3. The reasons why participants in G1 accomplished some tasks slower and passed less test cases than participants in G3 are as follows. First, when the participants rely on the API sequence recommendations to accomplish a task, the more accurately (the higher quality) the API sequence recommendations the more test cases will be passed. However, the quality of the API sequence recommendations provided by the plugin of G1 is lower than that of G2. Thus, when the participants mainly rely on API sequence recommendations of relatively low quality in G1, which may need extra efforts to make changes, their thinking may be limited to the API sequence recommendations and ignore to carefully check and change the API sequence recommendations accordingly. In this situation, it is better for

Table 5 Task completion time of the tasks (seconds)

Task	Group	avg	min	max	median	stan. dev.	p-value (G1 & G2)
T1	G1	749.5	454	1130	693.5	227.57	0.0156
	G2	**402.6**	140	1052	308.5	310.98	
	G3	523.4	132	1028	468	339.08	
T2	G1	643.3	335	960	629.5	273.47	0.0518
	G2	**393.8**	110	720	387.5	262.78	
	G3	741.3	395	1200	672.5	322.29	
T3	G1	429.8	180	660	454	182.93	0.1860
	G2	**336.6**	124	648	245.5	192.11	
	G3	398.0	205	649	344	188.19	
T4	G1	1037.8	736	1200	1150	204.61	0.0013
	G2	**332.4**	87	959	189	326.74	
	G3	959.0	246	1200	1200	374.23	
T5	G1	844.9	281	1200	990	396.30	0.0633
	G2	**524.0**	185	1200	313.5	420.74	
	G3	911.4	274	1200	1197.5	417.84	
T6	G1	407.9	122	1080	291	327.72	0.2474
	G2	**258.3**	97	500	214.5	143.45	
	G3	435.8	134	1200	282	370.30	

Table 6 Test pass rate of the tasks

Task	Group	avg	min	max	median	stan. dev.	p-value (G1 & G2)
T1	G1	0.67	0.00	1.00	1.00	0.47	0.1199
	G2	**0.92**	0.33	1.00	1.00	0.24	
	G3	0.88	0.33	1.00	1.00	0.25	
T2	G1	0.78	0.00	1.00	1.00	0.41	0.2441
	G2	**0.97**	0.75	1.00	1.00	0.09	
	G3	0.44	0.00	1.00	0.25	0.48	
T3	G1	0.66	0.13	1.00	0.69	0.37	0.0417
	G2	**0.94**	0.50	1.00	1.00	0.18	
	G3	0.72	0.00	1.00	1.00	0.41	
T4	G1	0.50	0.00	1.00	0.50	0.53	0.0151
	G2	**1.00**	1.00	1.00	1.00	0.00	
	G3	0.38	0.00	1.00	0.00	0.52	
T5	G1	0.25	0.00	1.00	0.00	0.46	0.0303
	G2	**0.75**	0.00	1.00	1.00	0.46	
	G3	0.38	0.00	1.00	0.00	0.52	
T6	G1	0.77	0.00	1.00	1.00	0.43	0.0855
	G2	**1.00**	1.00	1.00	1.00	0.00	
	G3	0.88	0.00	1.00	1.00	0.35	

the participants to write the code by themselves if they know how to write. Second, the API sequence recommendations provided by the plugin of G1 have an overlap. This may make the participants get lost in the recommendation list, which may waste their time for selection. However, when using the post-processing strategies that we proposed, participants in G2 accomplished each task faster and passed more test cases than participants in G3. This indicates that our proposed post-processing strategies can improve the quality of the API sequence recommendations and help participants to select the expected API sequence faster. These results show that it is necessary to improve the quality of the API sequence recommendations to avoid side effects.

In addition, we further analyzed the improvement achieved by the plugin of G2 when compared with the plugin of G1 to evaluate the effectiveness of our heuristic-enhanced beam search compared with the original beam search, clustering algorithm and summarizing method. We can see that participants in G2 accomplished each task faster and passed more test cases than participants in G1. On average, participants in G1 took 407.9-1037.8 seconds to accomplish a task, while participants in G2 spent 258.3-524.0 seconds. Participants in G1 passed 25-78% test cases, while participants in G2 passed 75-100% test cases. Especially, the test pass rate of T4 and T6 in G2 were 100%. The length of the API sequence and the cyclomatic complexity of each task is different. The length of the API sequence for each task is 16, 11, 19, 7, 5 and 5 respectively. The cyclomatic complexity of each task is 2, 2, 4, 1, 1 and 1 respectively. From the results, we can see that participants in G2 achieved better performance for each task regardless of the length and cyclomatic complexity of each task. It indicates that our heuristic-enhanced beam search can be applied to tasks with different lengths and cyclomatic complexity. This is because that the heuristic rules defined in our heuristic-enhanced beam search pay attention to characteristics of the source code itself.

The sequence length has no relation with the characteristics of the source code (which is consistent with the finding in RQ1). In addition, since we take into consideration the control units, we can successfully recommend API sequences with different cyclomatic complexity. We used Mann-Whitney U test to evaluate whether the difference of the task completion time and test pass rate between G1 and G2 was significant. A difference is considered to be significant if the p-value is less than 0.05. We can see that participants in G2 significantly outperformed participants in G1 in terms of task completion time for two tasks (i.e., T1 and T4) and in terms of test pass rate for three tasks (i.e., T3, T4 and T5).

Moreover, participants in G2 were asked to evaluate the effectiveness of the clusters and summaries of each task by giving scores ranging from 1 to 5 (the higher the score, the better the effectiveness) with the following three questions.

- **UQ1**: How is the quality (e.g., recommendations in the same cluster are similar) of the clusters?
- **UQ2**: Are the clusters helpful for you to distinguish recommendations?
- **UQ3**: Are the summaries useful to help you understand the recommendations?

The average score of **UQ1**, **UQ2** and **UQ3** of each task ranged from 3.5 to 4.4, which indicates that the feedback of clusters and summaries are quite positive, and our clusters and summarizations do help developers better distinguish and understand the recommendations.

Interview with Participants We also conducted an interview with each participant to obtain the feedback of using two plugins. We asked the following two interview questions.

- **IQ1**: When are you willing to get help from plugins?
- **IQ2**: In which tasks do you think the plugins help you?

For **IQ1**, all participants in G1 and G2 said that they preferred to use plugins when they had no idea on which API(s) to use to accomplish a task, they forgot the API(s), or the code to be written was long (i.e., a long sequence of function calls) though they knew the API(s). In the first and second situations, the API sequence recommendations can help them correctly accomplish a task. In the third situation, the API sequence recommendations can help them accomplish a task faster. If they were familiar with the API(s) related to a task and the code to be written was not long, they preferred to write the code by themselves.

For **IQ2**, participants in G2 said that the plugin they used could help them accomplish each task, especially for T4 and T5. They said that they had no idea which API(s) to use to accomplish tasks T4 and T5, but the API sequence recommendations provided by the plugin gave them solutions and the clusters and summaries helped them distinguish and understand the recommendations and facilitate their selection. For example, participants in G2 said that the clusters in T5 successfully distinguished API sequence recommendations (e.g., a cluster for invoking a method using the reflection mechanism, a cluster for invoking a field using the reflection mechanism, a cluster for creating an object instance using the reflection mechanism, and a cluster for connecting to a database). In addition, the summaries for each cluster can help them understand the API sequence recommendations and successfully select the one to create an object instance using the reflection mechanism.

Then, we investigated the API sequence recommendations provided by plugins of G1 and G2 for T4 and T5 respectively, and we found that the plugin of G2 successfully provided the correct API sequence for T4 and T5, but the plugin of G1 did not. The successful API sequence recommendation of the plugin in G2 for T4 and T5 is credited to our heuristic-enhanced beam search. Two of the API sequence recommendations for T5 without our

heuristic-enhanced beam search were shown in Fig. 3. These two API sequences have an inclusion relationship, and thus one of these two API sequence recommendations should be pruned. With our heuristic-enhanced beam search, the second API sequence in Fig. 3 with a higher probability is preserved, which leaves a position in the top 10 recommendations for the API sequence related to creating an object instance using the reflection mechanism. We also asked the reason that some participants in G1 successfully accomplished T4 and T5. It turns out that they knew the key API(s) for T4 and T5. Thus, when they wrote the key API(s) in the code context, the plugin in G1 can also provide the correct API sequence recommendation. However, most participants in G1 and G2 did not know the key API(s) and the advantages of the plugin in G2 can help them complete the task.

We also analyzed the performance of T1, T2, T3 and T6 for two groups. For T1, even though the plugin in G1 failed to provide the API sequence recommendations that participants expected, the participants in G1 still accomplished the task by themselves because they were familiar with the APIs for reading a file. However, the plugin in G2 provided the expected API sequence for participants to select. As a result, the improvement of the task completion time for T1 is significant.

For T3, the plugin in G2 provided an API sequence recommendation to judge whether a file is a directory, which required that the participants recursively process a directory. However, some participants in G1 forgot to recursively process a directory as the above recommendation was not provided. As a result, the improvement of the test pass rate is significant.

For T2 and T6, their API usages were relatively common and rigid, which are to use regex to extract a string and digest a string with MD5. Thus, both plugins can provide the correct API sequence recommendations. As a result, the improvement of the task completion time and test pass rate for T2 and T6 are not significant.

4.5 Threats to Validity

The threats to the internal validity lie in two aspects. First, the ground truth of clustering labels in RQ2 were manually labeled by two authors. However, the original agreement of the two authors was substantial and we organized group discussions to reach an agreement when they had conflicts. Second, the test cases in RQ3 for each task may not be complete. However, the test cases were designed by taking into consideration of all situations as many as possible of each task.

The threats to the external validity lie in two aspects. First, our approach focuses on JDK APIs. Thus, it is not clear how well the approach can support APIs in other libraries. Second, the test instances in RQ1 were constructed automatically which may not reflect the scenarios in real world. Thus, we also conducted a user study to evaluate the scenarios in real world.

5 Related Work

A large body of works on code recommendation are related to this work. These code recommendation methods recommend the next API or a code snippet under a given context.

Most of the existing code recommendation methods focus on recommending the next API under a given context. Traditional methods (Pletcher and Hou 2009; Hou and Pletcher 2010, 2011) heavily rely on the type information to infer the next API. Some methods (Bruch et al. 2009; Asaduzzaman et al. 2016) focus on computing the similarity between

the given context and previously seen code context to provide recommendations. Many methods (Hindle et al. 2012; Allamanis and Sutton 2013; Nguyen et al. 2013, 2015, 2018; Tu et al. 2014; Raychev et al. 2014; Liu et al. 2018; White et al. 2015; Dam et al. 2016; Yan et al. 2018; Svyatkovskiy et al. 2019; Li et al. 2018; Terada and Watanobe 2019; Yang et al. 2017, a, b, c; Chen et al. 2019a) have been proposed based on learning statistical language models from source code for recommending the next API. The statistical language models can be as simple as n-gram model or as complex as deep learning models. For example, Hindle et al. (2012) train an n-gram model based on the tokens of the source code to recommend the next token, which can be used to recommend the next API. Tu et al. (2014) leverage a cache to capture the localized regularities in the source code to enhance the n-gram model. Nguyen and Nguyen (2015) propose a graph-based statistical language model, which treats source code as graphs and leverages Bayesian statistical inference to compute the probabilities between a given graph and the graphs in the code corpus, to recommend the next API. Deep learning models are also widely used for code recommendation and achieve good performance. For example, Dam et al. (2016) train an LSTM neural network based on code tokens. Yan et al. (2018) enhance the LSTM neural network with deterministic negative sampling to filter out the APIs that do not belong to current class. Chen et al. (2019a) treat source code as trees and apply Tree-LSTM to train a deep learning model to recommend the next API. Compared with the above methods, our approach focuses on recommending API sequences under a given context instead of just the next single API.

There also exist works focusing on recommending a code snippet under a given context. One category of these approaches, such as Fowkes and Sutton (2016), Wang et al. (2013), Zhong et al. (2009), Nguyen et al. (2009), and Nguyen et al. (2012), focus on mining API usage patterns and then these mined patterns can be used to recommend a code snippet by matching the mined API patterns with the given context. For example, Zhong et al. (2009) apply frequent subsequence mining to mine API usage patterns based on clustered API sequences. Nguyen et al. (2009) mine API usage patterns based on API usage graphs instead of API sequences. Nguyen et al. (2012) propose an approach to try to match mined patterns as in Nguyen et al. (2009) with the given code context based on graph-based features and token-based features. Methods based on mined API usage patterns are limited to the number of mined API usage patterns. In comparison, benefit from the deep learning models, there is no need for our approach to mine API usage patterns in advance.

The other category of current methods, such as Hill and Rideout (2004), Luan et al. (2019), Ai et al. (2019), Kim et al. (2018), and Krugler (2013), define and extract features from source code and then search for the most similar code snippets as recommendations based on these features. For example, Hill and Rideout (2004) represent a method as a vector that contains different features, such as the number of lines of code, the number of arguments and the cyclomatic complexity, and then apply the k-nearest neighbor algorithm to detect similar code snippets. Luan et al. (2019) define a set of structural features (e.g., token features, variable usage features and parent features) of source code represented as a feature matrix, and then search similar code snippets based on these features. These methods rely heavily on the quality of the defined and extracted features. In comparison, above mentioned methods focus on recommending code snippets in a concretized way. However, our approach focuses on recommending API sequences in an abstract way. Therefore, the scenario is different. In addition, it is inappropriate to evaluate above mentioned methods with the CIDEr score. Because the code snippets maybe contain native methods that do not exist in the ground truth and the variable names in the code snippets and the ground truth may be quite different, which may affect the CIDEr score. Thus, we do not compare our approach with above mentioned methods. Furthermore, our approach provides clusters

and summaries to help developers understand and select the correct recommendation more conveniently and easily.

There are also other approaches aiming at recommending code snippets, but the input is not the source code. Mandelin et al. (2005) and Alnusair et al. (2010) focus on recommending code snippets based on a given input type of an object and an output type of an object. Many methods focus on recommending code snippets based on a given natural language query, such as Guet al. (2016, 2018), Sachdev et al. (2018), Cambronero et al. (2019). Compared with the above methods approaches, our approach takes incomplete source code directly as input.

There exist approaches that leverage encoder-decoder models for program repair, such as Tufano et al. (2019a, 2019b), Chen et al. (2019b) and Chakraborty et al. (2020). These approaches use a variety of syntactic and semantic representations to improve the performance of program repair. Though these approaches achieve different improvements, there still leaves room for improvement by using our idea of applying post-processing for encoder-decoder models. Take CODIT (Chakraborty et al. 2020) as an example. It is valuable to use tree-based neural networks, which take into consideration the syntactic structure, to model source code in CODIT. This indicates that using better code representations with corresponding neural networks makes sense, which is consistent with the results in our experiment (RQ1). However, there also leaves room for improvement in CODIT. In our work, we emphasize the idea of using post-processing for encoder-decoder models to reduce false positives instead of making the encoder-decoder models complicated. For example, we can define a heuristic rule to check whether a method call is available in the previous generated object in the token generation model in CODIT. For another example, we can define a heuristic rule to preserve the correct tokens of a token type instead of using mask operation in the encoder-decoder models to reduce the complexity of the encoder-decoder models. To sum up, the idea of applying post-processing for encoder-decoder models we propose is more flexible and can reduce the complexity of the encoder-decoder models. Essentially, encoder-decoder models are probabilistic models. Thus, even with better code representations, encoder-decoder models cannot guarantee that there is no false positive only based on probabilities. In addition, knowledge can be introduced into post-processing in the future to enhance the reasoning capability. This means that even if we use better abstract representations of source code, we still have to do necessary post-processing to further improve its capability.

Nguyen et al. (2019) apply program syntax rule and accessibility rule to recommend a statement. Though the main idea of Nguyen et.al is similar to us, which is to improve the performance through post-processing, our approach is still different from theirs in the following aspects. First, we focus on recommending an API sequence, while they focus on recommending a statement. Thus, the scenario is different. Second, since the scenario is different, the details of syntax-oriented rules are different. In fact, the detailed rules are designed based on the code representation of the corresponding scenario. Third, we design a diversity-oriented rule to deal with the overlap problem of the original beam search, which Nguyen et.al do not consider. Fourth, although the experiments in Nguyen et al. (2019) show the effectiveness of post-processing for the n-gram model, the effectiveness of post-processing for deep learning models is not clear. Thus, there is a need to evaluate the effectiveness of post-processing for deep learning models. Since our heuristic-enhanced beam search can ease the overlap problem and improve the diversity of the API sequence recommendations, there is a need to help developers distinguish API sequences of different semantics. This is why we propose a clustering algorithm. Since there may be several clusters of API sequences, there is a need to generate a summary for each cluster to help developers more

quickly and easily capture and understand the core semantics of API sequences. This is why we propose a method to generate a summary for each cluster.

6 Conclusion

In this work, instead of pursuing more advanced deep learning models, we propose an approach, named COOK, to enhance deep learning-based API sequence recommendation with three post-processing strategies, i.e., heuristic-enhanced beam search, and clustering and summarizing API sequence recommendations. Our strategies can improve the quality of recommended API sequences, and facilitate developers to understand and select recommended API sequences. Our extensive experiments on real-world projects and a user study with 24 participants have demonstrated the promising results of our approach.

Acknowledgements This work is supported by National Natural Science Foundation of China under Grant No. 61972098.

Data Availability We have released the source code of COOK with all the data at our website https://cook2020.wixsite.com/cook.

References

Ai L, Huang Z, Li W, Zhou Y, Yu Y (2019) SENSORY: leveraging code statement sequence information for code snippets recommendation. In: 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, vol 1, Milwaukee, pp 27–36

Allamanis M, Sutton CA (2013) 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, pp 207–216

Alnusair A, Zhao T, Bodden E (2010) Effective API navigation and reuse

Asaduzzaman M, Roy CK, Schneider KA, Hou D (2016) A simple, efficient, context-sensitive approach for code completion. J Softw Evol Process 28(7):512–541

Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings

Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. Comput Netw 30(1-7):107–117

Bruch M, Monperrus M, Mezini M (2009) 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, pp 213–222

Cambronero J, Li H, Kim S, Sen K, Chandra S (2019) When deep learning met code search. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, pp 964–974

Chakraborty S, Ding Y, Allamanis M, Ray B (2020) Codit: Code editing with tree-based neural models. IEEE Trans Softw Eng:1–1

Chen C, Peng X, Sun J, Xing Z, Wang X, Zhao Y, Zhang H, Zhao W (2019a) Generative API usage code recommendation with parameter concretization. Sci China Inf Sci 62(9):192103:1–192103:22

Chen C, Peng X, Xing Z, Sun J, Wang X, Zhao Y, Zhao W (2021) Holistic combination of structural and textual code information for context based api recommendation. IEEE Trans Softw Eng:1–1

Chen Z, Kommrusch SJ, Tufano M, Pouchet LN, Poshyvanyk D, Monperrus M (2019b) Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Trans Softw Eng:1–1

Dam HK, Tran T, Pham T (2016) A deep language model for software code. CoRR arXiv:1608.02715

Fowkes JM, Sutton CA (2016) Parameter-free probabilistic API mining across github. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, pp 254–265

Fowlkes EB, Mallows CL (1983) A method for comparing two hierarchical clusterings. J Amer Stat Assoc 78(383):553–569

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

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

Hill R, Rideout J (2004) Automatic method completion. In: 19Th IEEE international conference on automated software engineering (ASE 2004), Linz, pp 228–235

Hindle A, Barr ET, Su Z, Gabel M, Devanbu PT (2012) On the naturalness of software. In: 34Th international conference on software engineering, ICSE 2012, Zurich, pp 837–847

Hou D, Pletcher DM (2010) 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, pp 26–30

Hou D, Pletcher DM (2011) 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, pp 233–242

Huang Y, Chen C, Xing Z, Lin T, Liu Y (2018) Tell them apart: distilling technology differences from crowd-scale comparison discussions. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, pp 214–224

Hubert L, Arabie P (1985) Comparing partitions. J Class 2(1):193–218

Kim K, Kim D, Bissyandé TF, Choi E, Li L, Klein J, Traon YL (2018) Facoy: a code-to-code search engine. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, pp 946–957

Krugler K (2013) Krugle code search architecture. In: Finding Source Code on the Web for Remix and Reuse, pp 103–120

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

Li Y, Tarlow D, Brockschmidt M, Zemel RS (2016) Gated graph sequence neural networks. In: 4th International Conference on Learning Representations, ICLR 2016. Conference Track Proceedings, San Juan

Liu X, Huang L, Ng V (2018) Effective API recommendation without historical software repositories. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, pp 282–292

Luan S, Yang D, Barnaby C, Sen K, Chandra S (2019) Aroma: code recommendation via structural code search. Proc ACM Program Lang 3(OOPSLA):152:1–152:28

Mandelin D, Xu L, Bodík R, Kimelman D (2005) Jungloid mining: helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, pp 48–61

McHugh ML (2012) Interrater reliability: the kappa statistic. Biochem Med Biochem Med 22(3):276–282

Nguyen AT, Nguyen TN (2015) Graph-based statistical language model for code. In: 37Th IEEE/ACM international conference on software engineering, ICSE 2015, vol 1, Florence, pp 858–868

Nguyen AT, Nguyen TT, Nguyen HA, Tamrawi A, Nguyen HV, JM Al-Kofahi, Nguyen TN (2012) Graph-based pattern-oriented, context-sensitive source code completion. In: 34Th international conference on software engineering, ICSE 2012, Zurich, pp 69–79

Nguyen AT, Hilton M, Codoban M, Nguyen HA, Mast L, Rademacher E, Nguyen TN, Dig D (2016) 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, pp 511–522

Nguyen AT, Nguyen TD, Phan HD, Nguyen TN (2018) A deep neural network language model with contexts for source code. In: 25Th international conference on software analysis, evolution and reengineering, SANER 2018, Campobasso, pp 323–334

Nguyen SV, Nguyen TN, Li Y, Wang S (2019) Combining program analysis and statistical language model for code statement completion. In: 34Th IEEE/ACM international conference on automated software engineering, ASE 2019, San diego, pp 710–721

Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi JM, Nguyen TN (2009) 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, pp 383–392
Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN (2013) 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, pp 532–542
Nguyen XV, Epps J, Bailey J (2010) Information theoretic measures for clusterings comparison: variants, properties, normalization and correction for chance. J Mach Learn Res 11:2837–2854
Pletcher DM, Hou D (2009) BCC: enhancing code completion for better API usability. In: 25Th IEEE international conference on software maintenance (ICSM 2009), Edmonton, pp 393–394
Raychev V, Vechev MT, Yahav E (2014) Code completion with statistical language models. In: ACM SIGPLAN Conference on programming language design and implementation, PLDI '14, Edinburgh, pp 419–428
Rosenberg A, Hirschberg J (2007) V-measure: A conditional entropy-based external cluster evaluation measure. In: EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, Prague, pp 410–420
Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math 20(1):53–65
Sachdev S, Li H, Luan S, Kim S, Sen K, Chandra S (2018) Retrieval on source code: a neural code search. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, pp 31–41
Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N (2019) Pythia: Ai-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, pp 2727–2735
Terada K, Watanobe Y (2019) Code completion for programming education based on recurrent neural network. In: 11Th IEEE international workshop on computational intelligence and applications, IWCIA 2019, Hiroshima, pp 109–114
Tu Z, Su Z, Devanbu PT (2014) On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, pp 269–280
Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D (2019a) On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, pp 25–36
Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2019b) An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans Softw Eng Methodol 28(4):19:1–19:29
Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems 30: Annual conference on neural information processing systems 2017, Long Beach, pp 5998–6008
Vedantam R, Zitnick CL, Parikh D (2015) Cider: Consensus-based image description evaluation. In: IEEE Conference on computer vision and pattern recognition, CVPR 2015, Boston, pp 4566–4575
Wang J, Dang Y, Zhang H, Chen K, Xie T, Zhang D (2013) 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, pp 319–328
White M, Vendome C, Vásquez ML, Poshyvanyk D (2015) Toward deep learning software repositories. In: 12Th IEEE/ACM working conference on mining software repositories, MSR 2015, Florence, pp 334–345
Yan J, Qi Y, Rao Q, He H (2018) Learning API suggestion via single LSTM network with deterministic negative sampling. In: The 30th international conference on software engineering and knowledge engineering, hotel pullman, Redwood City, pp 137–136
Yang Y, Jiang Y, Gu M, Sun J, Gao J, Liu H (2017) A language model for statements of software code. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, pp 682–687
Yang Y, Xiang C (2019a) Improve language modelling for code completion by tree language model with tree encoding of context (S). In: The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, pp 675–777
Yang Y, Xiang C (2019b) Improve language modelling for code completion through learning general token repetition of source code. In: The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, pp 667–777
Yang Y, Chen X, Sun J (2019c) Improve language modeling for code completion through learning general token repetition of source code with optimized memory. Int J Softw Eng Knowl Eng 29(11&12):1801–1818

Zhong H, Xie T, Zhang L, Pei J, Mei H (2009) MAPO: mining and recommending API usage patterns. In: ECOOP 2009 - Object-Oriented Programming, 23rd European Conference. Proceedings, Genoa, pp 318–343

Zohar A, Wolf L (2018) Automatic program synthesis of long programs with a learned garbage collector. In: Advances in neural information processing systems 31: Annual conference on neural information processing systems 2018, neurIPS 2018, Montrėal, pp 2098–2107

Chi Chen

Xin Peng

Bihuan Chen

Jun Sun

Zhenchang Xing

Xin Wang

Wenyun Zhao

Affiliations

Chi Chen[1,2] · Xin Peng[1,2] · Bihuan Chen[1,2] · Jun Sun[3] · Zhenchang Xing[4] · Xin Wang[1,2] · Wenyun Zhao[1,2]

1 School of Computer Science, Fudan University, Shanghai, China

2 Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China

3 Singapore Management University, Singapore, Singapore

4 Australian National University, Canberra, Australia