SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2020, Vol. 63 190103:1–190103:15
https://doi.org/10.1007/s11432-019-2929-9

Special Focus on Big Data-Driven Intelligent Software Development for Onsite Programming

An analysis of correctness for API recommendation: are the unmatched results useless?

Xianglong KONG, Weina HAN, Li LIAO & Bixin LI*

School of Computer Science and Engineering, Southeast University, Nanjing 211189, China

Received 20 September 2019/Revised 24 January 2020/Accepted 24 April 2020/Published online 13 August 2020

Abstract API recommendation is a promising approach which is widely used during software development. However, the evaluation of API recommendation is not explored with sufficient rigor. The current evaluation of API recommendation mainly focuses on correctness, the measurement is conducted by matching recommended results with ground-truth results. In most cases, there is only one set of ground-truth APIs for each recommendation attempt, but the object code can be implemented in dozens of ways. The neglect of code diversity results in a possible defect in the evaluation. To address the problem, we invite 15 developers to analyze the unmatched results in a user study. The online evaluation confirms that some unmatched APIs can also benefit to programming due to the functional correlation with ground-truth APIs. Then we measure the API functional correlation based on the relationships extracted from API knowledge graph, API method name, and API documentation. Furthermore, we propose an approach to improve the measurement of correctness based on API functional correlation. Our measurement is evaluated on a dataset of 6141 requirements and historical code fragments from related commits. The results show that 28.2% of unmatched APIs can contribute to correctness in our experiments.

Keywords API recommendation, onsite programming, correctness, evaluation of recommendation

Citation Kong X L, Han W N, Liao L, et al. An analysis of correctness for API recommendation: are the unmatched results useless? Sci China Inf Sci, 2020, 63(9): 190103, https://doi.org/10.1007/s11432-019-2929-9

1 Introduction

Application programming interface (API) plays an important role in the promotion of code reuse. During software development, especially for onsite programming, developers need to search for suitable third party libraries or the usage of specific APIs frequently [1]. The previous study found that developers carry out more than 20 search queries related to API everyday [2]. In the past decade, a huge body of research efforts have been dedicated to recommendation systems for software engineering (RSSE) [3–5], which support developers to program effectively and efficiently [6]. Many API recommendation techniques are proposed in recent years, resulting in the problem about selection of suitable API recommendation techniques in a specific scenario. The selection of API recommendation techniques is mainly determined by the results of evaluation [7, 8]. However, the current evaluation of API recommendation is commonly the same as that in traditional recommendation system, regardless of the features of API usage [9].

The current evaluation of API recommendation can be divided into online evaluation and offline evaluation according to the existence of human interaction. The online evaluation is often infeasible due to the high cost of human labor [10]. To overcome these limitations, many researchers turn to offline evaluation [11–16]. However, offline evaluations cannot always reflect the accurate quality of the system due to

* Corresponding author (email: bx.li@seu.edu.cn)

© Science China Press and Springer-Verlag GmbH Germany, part of Springer Nature 2020

the lack of mature evaluation model [17]. The current offline evaluation for API recommendation mainly focuses on the correctness of recommended results. Correctness provides an indicator that measures the consistency of recommended results and expected results. The expected set of recommendation results is also referred to the ground-truth APIs [18], which usually comes from historical data. The ground-truth APIs are commonly constructed by picking out a snippet from a complete code block or extracting committing code from issue tracking system. While evaluating the effectiveness of API recommendation, the current work usually matches the candidate APIs with a set of ground-truth APIs. The value of correctness is commonly calculated by counting the number of matched APIs. The matched APIs denote the APIs that appear in both recommended results and ground-truth APIs. The other APIs in the recommended results are marked as unmatched results, which are usually treated as incorrect answers for the inputs.

In most cases, there is only one set of ground-truth APIs for a specific requirement, and developer may have dozens of ways to implement the requirement in practice. Owing to the diversity of programming, it is impossible to list all the possibly correct solutions for a specific requirement. For example, both String.valueOf() and Integer.to String() can convert data type from int to string, they are used to implement the same feature and they can replace with each other in a recommendation scenario. However, only one of them would be selected while building the ground-truth results for this recommendation attempt. If the other one is recommended by some techniques, the result will be marked as a wrong answer because it is an unmatched result. In practice, the replaceability of APIs is hard to consider while building the ground-truth APIs. The current incomplete matching mechanism brings two disadvantages. First, the inaccurate evaluation makes the users hard to select the required API recommendation techniques. Second, the unique fitness on the specific ground-truth results would neglect the potential of some heuristic methods. Furthermore, the unmatched results may also inspire the developers to choose the correct APIs through the similar API name or description. The possible contributions of unmatched results inspire us to conduct a deep analysis on correctness for API recommendation.

The motivation of this study comes from a hypothesis that the unmatched results can also contribute to the correctness due to the diversity of implementation. So we firstly verify this hypothesis through online evaluation in a user study. Fifteen developers are invited to evaluate the unmatched results. The results confirm that some unmatched APIs can help to implement the requirement, and the useful unmatched APIs are usually correlated with ground-truth APIs in terms of functionality, i.e., their functionalities are equivalent or similar. Then we extract information of API relationships from several sources, i.e., API knowledge graph, API method name and API documentation. The extracted information is combined to measure the API functional correlation. At last, we propose an approach to improve the measurement of correctness, which involves API functional correlation in the traditional calculation of correctness. Traditionally, the matched API contributes 1 and the unmatched API contributes 0 to correctness. We improve this measurement by assigning a value between 0 and 1 to the unmatched API which correlates to the ground-truth API. To evaluate the approach, we reimplement an existing API recommendation technique [19] to extract recommended results on a dataset of 6141 requirements and committed code. For each requirement, there exists a list of ground-truth APIs which are actually used to implement it in history. The results show that our approach can help to identify the contribution of unmatched APIs to correctness in terms of precision and recall. This study provides new insights to the offline evaluation of API recommendation. The improved measurement can help researchers make a more accurate assessment of API recommendation results.

To sum up, this paper makes the following contributions.

- We explore the problem about measurement of correctness for API recommendation through a user study. Matching recommended results with the ground-truth APIs through text is not a perfect way to measure correctness.
- We find that the unmatched results which have functional correlation with ground-truth APIs may also contribute to the correctness.
- We propose an approach to improve the measurement of correctness. The evaluation of our approach shows that 28.2% of unmatched APIs in our study can contribute to correctness of recommended results.

The remainder of the paper is organized as follows. In Section 2, we make an overview of background information. In Section 3, we conduct a user study to analyze the effects of unmatched results and Section 4 shows how to measure the contributions of unmatched results. Section 5 presents the evaluation of our approach . In Section 6, we analyze the threats to validity. Finally, Section 7 concludes the work with some final remarks and future work.

2 Background

In this section, we present the background of the analysis on correctness of API recommendation. We mainly focus on the offline evaluation of correctness. The calculation of correctness can be different for different kinds of API recommendation techniques. We roughly divide the API recommendation techniques into two types, i.e., specification-based API recommendation and context-based API recommendation according to the different source of information.

2.1 API recommendation

Specification-based API recommendation. This kind of techniques are conducted based on the relationship between the requirement and the code that can implement it [9,20]. To automate the recommendation, the requirement is usually represented as some specifications. The most famous specification-based API recommendation technique is proposed by Thung et al. [19]. They analyze the relationship between description of feature requests and the used APIs in the committing code to achieve the recommendation. In the recommendation of specification-based techniques, the input information is presented as a functional description, and the recommended results are sets of possible APIs which can be used together to implement the input feature. The range of candidate APIs is determined by the selection of third party libraries, all the available APIs from the selected libraries have chance to be recommended. The ground-truth result is a set of APIs which is extracted from the historical code that is committed during the accomplishment of specifications. The specification-based API recommendation techniques can generate several results for a specific functional description, and each result may contain more than one candidate APIs. The ground-truth results may not exist in the sets of candidate results due to the limitation of selected libraries.

Context-based API recommendation. This kind of techniques use the relationship between APIs and the surrounding code to train a model that can recommend candidate API with an input of new context [3,5,21]. The recommendation of context-based techniques are usually triggered while typing a dot after a class or instance variable within an IDE. The range of candidate APIs is limited by the attributes of triggered elements, i.e., only the callable APIs are considered for a specific editing location. The ground-truth result is conducted by picking out some APIs from existing code snippets. The remaining code is treated as the input of context and the removed API is the ground-truth result. Each candidate result can only contain one API and the ground-truth API must exist in the sets of candidate APIs.

For context-based API recommendation techniques, each result only contains one API and the correct result certainly exists in the candidate APIs. There rarely exist replaceable APIs within a specific pool of candidate APIs for an editing location, so the unmatched APIs are hard to contribute to the correctness of context-based API recommendation in theory. In this paper, we mainly focus on the specification-based API recommendation techniques which can generate several sets of candidate APIs in a recommendation attempt. Each set of candidate APIs refers to a result, and it may contain matched APIs and unmatched APIs meanwhile. The APIs contained in a set are expected to achieve the feature together. In the traditional evaluation, the unmatched results are marked as incorrect answers, resulting in a possible lost of information [22]. Actually, the unmatched results have potential to achieve similar functionality with the APIs in the ground-truth results due to the diversity of code implementation. So we conduct a deep analysis on the contribution of unmatched APIs in this paper.

	Results of recommendation #1	Ground-truth APIs	Results of recommendation #2	
	Iterables.getOnlyElement Lists.newArrayList	Iterables.getOnlyElement Lists.newArrayList	Iterables.getOnlyElement Lists.newArrayList	Matched APIs
Unmatched-but-useful APIs	Iterables.add LogFactory.getLog Map.HashMap	Iterables.addAll LoggerFactory.getLogger Maps.newHashMap	Mockito.mock LogFactory.getLog InvocationOnMock.get	
		Iterables.removef	Arguments	

Figure 1 An example on the contributions of unmatched APIs.

2.2 Evaluation for API recommendation

The specification-based API recommendation technique usually generates several sets of candidate APIs, and each set may contain more than one APIs [9, 19]. In most cases, the correctness is measured in terms of precision and recall. Some existing work also adopt mean average precision (MAP) and mean reciprocal rank (MRR) metrics in their study [23–25]. These metrics are explained as below.

- Precision: ratio of correct recommended results to all the recommended results.
- Recall: ratio of correct recommended results to all the ground-truth results.
- MAP: the average accuracy of all correct APIs in recommended results, the closer to the top of the recommendation results for the related methods, the higher the MAP score will be.
- MRR: the average accuracy of the first feature related method, the closer to the top of the recommendation results for the first related method is, the higher the MRR score will be.

The context-based API recommendation technique generally produces a list of several results on en editing location, and each result only contain one candidate API. There is only one correct API in the recommended list. Top-k accuracy is commonly used to measure the correctness [7]. If a ground-truth API m locates the top k position in the list of recommended results, it is treated as a hit for top-k accuracy. The overall top-k accuracy is calculated as the ratio of the number of hits to the total number recommendation attempts.

There are also some other metrics for API recommendation. For example, when the order of candidate API in a recommended result can impact the effectiveness, BLEU [26], NDGG [6], and DCG [27] are used to evaluate the correctness. Since these metrics are usually considered in the recommendation of API usage sequence and example code, we do not discuss them in this paper.

3 Analysis on the unmatched results

Owing to the diversity of code implementation and features of specification-based API recommendation, we raise a hypothesis that the unmatched results can also contribute to the correctness. In the evaluation, the unmatched results denote the recommended APIs which do not appear in the ground-truth APIs. However, some unmatched APIs may be used to implement the same or similar functionality as the specific ground-truth APIs. We will verify the hypothesis through a user study in this section.

3.1 Motivating example

API recommendation techniques are design to improve the effectiveness and efficiency of programming, and correctness is the primary factor that can represent the quality of recommended results. The traditional measurement of correctness is calculated by counting matched APIs, the neglect of unmatched APIs may result in information loss during recommendation. We show an example on the contributions of unmatched results in Figure 1. The ground-truth APIs in this example come from HBase project in JIRA[1]. The results of recommendation #1 and #2 are generated by an existing technique [19] with

1) https://issues.apache.org/jira/projects/HBASE/issues.

different training data. In terms of traditional measurement, the results of recommendation #1 and #2 obtain the same precision (i.e., $2/5 = 0.4$) and recall (i.e., $2/6 = 0.33$). However, the two sets of candidate APIs have different efforts on the implementation of the input feature in practice. In the example, the candidate API LogFactory.getLog has the same functionality with the ground-truth API LoggerFactory.getLogger, and they can replace with each other. These two APIs have the same functionality and perform differently in terms of efficiency. The candidate API Iterables.add in the results of recommendation #1 is correlated to ground-truth API Iterables.addAll, developers can also implement the feature with the recommended answer. The last API Map.HashMap is also similar with Maps.newHashMap in terms of functionality. These unmatched APIs in the results of recommendation #1 can contribute to correctness, and the contribution is neglected by the traditional measurement of correctness.

3.2 User study

We conduct a user study to investigate the real efforts of unmatched APIs on the implementation of input features. Through the user study, we want to answer the following research question.

- RQ1: Are the unmatched results useless?

To answer the above research question fairly, we need to generate results of API recommendation and conduct the online evaluation. Therefore, we reimplement a widely used API recommendation technique and invite developers with different level of programming experience to evaluate its results.

Selection of API recommendation techniques. According to the discussion in Section 2, we mainly focus on specification-based API recommendation techniques. The most famous technique in this area is proposed by Thung et al. [19]. There are also some extending work from this technique [9, 20], but the innovations of the following work focus on either the recommendation of API usage example or the APIs in a specific domain. So the work proposed by Thung et al. [19] is still state-of-the-art method in the area of specification-based API recommendation. We reimplement their technique in this study.

Selection of studied projects. To obtain the specifications which are used as input information, we follow the existing work [19] to select feature requests from JIRA[2)], i.e., an issue management system. A feature request in JIRA is an issue during software development. JIRA contains explicit links between issues reports and repository commits. Using these links, we extract APIs that developers used to accomplish the issue in historical code. In total, we obtained 6141 feature requests, and there is one set of ground-truth APIs for each feature request. These feature requests come from 32 projects, which are shown in Table 1.

Selection of developers. We invite 5 senior developers from Huawei and AliBaba company, who have 3–5 years of working experience, and another 10 primary developers who are postgraduate students in our team. There are no other conditions in our selection and the domain knowledge of different developers is not considered in the user study.

To conduct the user study, we build two sets of training data and validation data based on the selected 6141 specifications to analyze the different results. For each set, we randomly select 90% of the available APIs as training data, and the other 10% of the available APIs as validation data. The sets of training data and validation data are randomly selected so that the results for a specific feature request may be different. For each feature request, we collect top-5 recommended APIs in our study. Finally, we obtain 4455 recommended results and ground-truth results by filtering the unavailable recommendation attempts.

For each selected developer, we assign 100 specifications and the corresponding 200 results from the two different sets of API recommendation attempts. To investigate the efforts of unmatched results clearly, we mark the matched APIs and unmatched APIs with different labels in each result. We set only one question in the study, i.e., are the unmatched results useless to the implementation of the feature? There are three options in the answers, useless, a little helpful, and very helpful. The developer should choose one option for each recommendation separately. For the 10 primary developers, the study is implemented

2) https://www.atlassian.com/software/jira.

Table 1 The statistics of projects in our study

Project	# Feature requests	Project	# Feature requests
Ambari	176	Avro	155
Axis2	162	Cassandra	195
CXF	192	Derby	199
Drill	180	Falcon	155
Felix	121	Flume	154
Groovy	162	HBase	286
Hive	397	Jackrabbit	201
LuceneCore	358	Mahout	285
MyFaces	188	Nutch	237
OFBiz	164	Oozie	111
PDFBox	173	Phoenix	166
Pig	318	Qpid	175
Solr	393	Sqoop	121
Tajo	138	Tez	92
Thrift	158	Tika	119
Wicket	79	ZooKeeper	131

Table 2 The feedback on unmatched results from the user study

Developers	Useless (%)	A little helpful (%)	Very helpful (%)
Senior developers	60	25	15
Primary developers	54	30	16

in our laboratory, and the developers spend around 4 hours to finish the work. For the 5 senior developers, the study is implemented remotely, and the developers spend around 3 hours to finish the survey.

3.3 Analysis and discussion

According to the feedback from the invited developers, we present the results in Table 2. From the table, we have the following findings. First, there are more than a half unmatched candidate APIs which are actually useless. The major part of results is coincident with traditional measurement of correctness. Second, 25% unmatched results are a little helpful for senior developers and 30% unmatched results are a little helpful for primary developers. In our study, opinion “a little helpful” means the unmatched APIs can contribute to correctness, but cannot replace the ground-truth APIs, i.e., they have similar features with ground-truth APIs. Third, there are 15% and 16% unmatched APIs which are marked as “very useful” by senior and primary developers in the study. The opinion “very useful” denotes that the unmatched APIs can replace ground-truth APIs with each other, i.e., they are equivalent in terms of functionality.

According to the results in Table 2, we find that more than 40% unmatched results can contribute to the correctness, i.e., they are useful for the implementation of the given specification. Through the analysis, the contribution comes from the functional correlation between recommended APIs and ground-truth APIs. The functional correlation can be presented as equivalence or similarity of functionalities. This kind of correlation is usually ignored in the traditional evaluation of correctness. This finding inspires us to improve the evaluation of correctness based on the functional correlation, and the improved correctness is expected to be close to the criterion of online evaluation.

Finding 1. The unmatched APIs that correlate with ground-truth APIs in terms of functionality can also benefit to programming to some extent, even they are marked as incorrect results.

valueOf
public static String valueOf (int *i*) Returns the string representation of the int argument. The representation is exactly the one returned by the Integer.toString method of one argument. Parameters: *i* - an int. Returns: A string representation of the int argument. See also: Integer.toString (int, int)

Figure 2 An example of "see also" relationship in API documentation.

4 Measurement of correctness based on API functional correlations

Our hypothesis on the contribution of unmatched results to correctness has been proved correct in the above section. Then we want to further investigate that how to measure the contribution of unmatched results. This kind of contribution comes from functional correlation between recommended APIs and ground-truth APIs. The API functional correlation can provide additional contribution to correctness. So we raise a new research question in this section.

• RQ2: What are the features of unmatched API that can impact functional correlation?

When the features of APIs that can impact functional correlation are clear, we can investigate an approach to measure the correlation to some extent. Then the evaluation of correctness can be improved based on API functional correlation.

4.1 Analysis on API functional correlation

According to the discussion in previous user study, if a candidate API has equivalent or similar functionality with specific ground-truth API, the API has potential to contribute to correctness of API recommendation, even it is not matched with ground-truth API. The API functional equivalence and functional similarity are treated as two types of API functional correlation in this paper. We will introduce a heuristic approach to instantiate the functional equivalence and functional similarity in this subsection.

API functional equivalence is a specific relationship between APIs, it means the corresponding APIs are used to implement the same functionality and they can replace with each other. Since general program equivalence is undecidable [28], we turn to find out API relationships from API knowledge graph. In this study, we obtain the API knowledge graph from CodeWisdom[3], which is built by Peng et al. from Fudan University. An API knowledge graph is an API centric knowledge base that links API elements (e.g., libraries, classes, methods, and parameters), their descriptive knowledge (e.g., functionalities and directives), and related background knowledge (e.g., concepts about computer and programming). The relationships in CodeWisdom are extracted from the API documents, StackOverflow, Github, and other sources. There are 7 primary types of relationships for Java APIs in CodeWisdom, i.e., link, return value, source from, throw exception, has exception, has return value and has parameter. Among these relationships, the strongest association is link, which denotes that the two APIs have correlated functionality, or they are usually used together. In code implementation, the link relationship is usually represented as "see also" reference[4]. The "see also" reference is an annotation that can be added to the head of a package, class or member which means the entity is strongly related with another entity. Figure 2 shows an example of "see also" reference. The API String.valueOf is linked to Integer.toString through "see also" reference. Both the two APIs can convert data type from int to string. Although the "see also" reference has another aspect of interpretation that means the corresponding APIs are commonly used together, this meaning cannot impact the usage of "see also" reference in our analysis. This is because

3) http://bigcode.fudan.edu.cn/.
4) https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

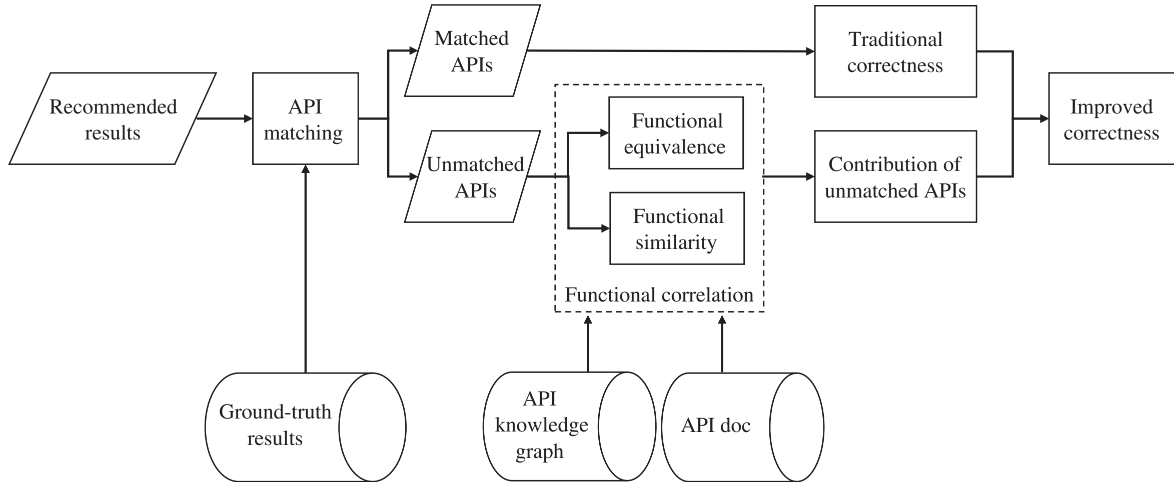

Figure 3 The overall framework of the evaluation of correctness based on API functional correlation.

that if two APIs are commonly used together, they either appear or miss together in the results due to the training model of API recommendation. The co-occurrence of "see also" reference do not exist in recommended results and ground-truth APIs respectively in our study. So we directly extract "see also" reference from the API knowledge graph to represent API functional equivalence.

The measurement of API functional similarity is inspired by the useful unmatched APIs in previous user study. We find that the useful unmatched APIs usually have similar name with the ground-truth APIs and their documents also have some similar contents. API name is the most important symbol of a certain API, and it usually indicates the functionality. The name of an API is a string that can be spliced by multiple words. We use the Levenshtein distance [29] to measure the name similarity. The Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. For example, LogFactory.getLog and LoggerFactory.getLogger have a high name similarity, which is 6, and the number of characters changed is about 30% of the total length. API documentation contains textual descriptions that explain the functionality and usage of the API. Similar API documentation can imply similar functionality to some extent. For example, the description of LogFactory.getLog in official document is "Convenience method to return a named logger, without the application having to care about factories", while the description of LoggerFactory.getLogger is "Return a logger named according to the name parameter using the statically bound ILoggerFactory instance". These two sentences are different expressions of the same meaning. According to the findings in existing work [19], we can measure the similarity of two descriptions with cosine similarity. From the above, we can use the similarity on API names and descriptions to measure the API functional similarity.

Finding 2. There are various features that can result in the correlation between unmatched results and ground-truth APIs. According to our analysis, we collect "see also" relationship in API knowledge graph, similarity on API names, and descriptions as indicators of the API functional correlation.

4.2 Improved evaluation of correctness

According to the above findings, we propose an improved evaluation of correctness based on API functional correlation. Figure 3 outlines the overall framework of our approach to improve the evaluation of correctness. For the given recommended results, our evaluation requires the ground-truth results, API knowledge graph and API documentation of the related APIs to obtain the improved correctness.

To achieve the evaluation, we firstly match the recommended results with the ground-truth results to select the matched APIs as the correct results. Then we calculate the value of correctness by applying the traditional measurement. Second, we measure the API functional correlation according to the evaluation

of functional equivalence and similarity, i.e., the dotted box in Figure 3. Finally, the improved evaluation of correctness is obtained through combining the traditional correctness and contribution of unmatched APIs. We will explain the details of our key methods (i.e., measurement of functional equivalence and similarity, the combination method) as follows.

4.2.1 *Measurement of API functional correlation*

In our approach, the API functional correlation can be decomposed into functional equivalence and functional similarity. These metrics are evaluated by comparing the unmatched APIs in recommended results with the ground-truth APIs one by one. For the API functional equivalence, we use "see also" relationship in API knowledge graph to represent it. We extract all the "see also" relationships from CodeWisdom to build a primary benchmark of equivalent APIs on functionality. The metric of the API functional equivalence is represented as SimEQ. We set SimEQ to 1 if there is a "see also" reference between the two APIs, otherwise it is set to 0.

The API functional similarity can be further decomposed into similarity on API name and similarity on API description. We use Levenshtein distance to measure the similarity on API name. The Levenshtein distance of unmatched API name and ground-truth API name denotes the minimum actions which are needed while transforming one name to the other name. Let A_1 denote the name of a recommended API, and A_2 denote the name of a ground-truth API. The Levenshtein distance $\mathrm{lev}_{(A_{1i},A_{2j})}$ between A_1 and A_2 is calculated as follows:

$$\mathrm{lev}_{A_1,A_2}(i,j)=\begin{cases}\max(i,j), & \text{if } \min(i,j)=0,\\ \min\begin{cases}\mathrm{lev}_{A_1,A_2}(i-1,j)+1,\\ \mathrm{lev}_{A_1,A_2}(i,j-1)+1,\\ \mathrm{lev}_{A_1,A_2}(i-1,j-1)+1_{(A_{1i}\neq A_{2j})},\end{cases} & \text{otherwise,}\end{cases} \tag{1}$$

where $\mathrm{lev}_{(A_{1i},A_{2j})}$ is equal to 0 when $A_{1i}=A_{2j}$, and $\mathrm{lev}_{A_1,A_2}(i,j)$ is the distance between the first character i of A_1 and the first character j of A_2. Note that the first expression in the minimum corresponds to the deletion from A_1 to A_2, the second expression corresponds to insertion and the third expression corresponds to match or mismatch, depending on whether the respective symbols are the same with each other. Based on Levenshtein distance, the similarity on API names can be calculated according to the following formula:

$$\mathrm{SimName}_{(A_1,A_2)}=\max\left\{1-\frac{\mathrm{lev}_{A_1,A_2}}{\max(\mathrm{len}_{A_1},\mathrm{len}_{A_2})}-\delta,0\right\}, \tag{2}$$

where $\mathrm{SimName}_{(A_1,A_2)}$ represents the similarity of two strings, len_{A_1} and len_{A_2} denote the length of A_1 and A_2 respectively. To our knowledge, two different API names should be treated as uncorrelated if the value of SimName is too low. So we design a threshold value δ in the formula. The similarity of API names could be considered while the value of SimName is higher than the threshold value. If the value of SimName is too low, the similarity of API names would be recorded as 0.

To obtain the similarity on API description, we apply standard natural language preprocessing (i.e., tokenization and stemming) to convert the API documents into a bag of words and convert these words to their corresponding term vectors by calculating the weight of each word. The weight of the words is analyzed based on TF-IDF. Then the similarity between two descriptions is calculated through cosine similarity. The formula is shown as follows:

$$\mathrm{SimDesc}(A_1,A_2)=\max\left\{\mathrm{Cosine}(\mathrm{Desc}(A_1),\mathrm{Desc}(A2))-\theta,0\right\}, \tag{3}$$

where Desc denotes the description of an API. If the similarity of API descriptions is smaller than θ, the two APIs are marked as irrelevant APIs and we will set the value to 0. Since the value of similarity is always positive, the low similarity may cause threats to the effectiveness of our approach. So we directly set the low similarity to 0 to eliminate the threats. The calculation of threshold values δ and θ will be discussed in Subsection 4.2.3.

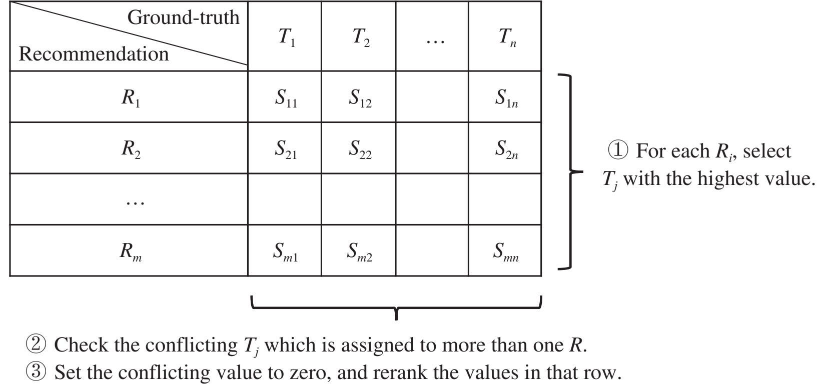

Figure 4 Combination of API functional correlations in a matrix.

4.2.2 *Combination of the measurements*

For each unmatched API in recommended results, we need to calculate its functional correlation with all the APIs in ground-truth results. In this way, the measurement of API functional correlation can produces a matrix which presents the similarity of the two sets of APIs. To describe the measurement clearly, let R and T denote recommended results and ground-truth results respectively, m denote the size of R and n denote the size of T $(m, n \geqslant 1)$. R_i refers to the i-th element in R and T_j refers to the j-th element in T. For each metric, we can get an $m \times n$ matrix as the output. The data in the matrix denotes the value of functional correlation on the pair of APIs. The API functional correlation is represented as Sim, it is calculated as follows:

$$\mathrm{Sim}(A_1, A_2) = \alpha \cdot \mathrm{SimName}(A_1, A_2) + \beta \cdot \mathrm{SimDesc}(A_1, A_2) + \gamma \cdot \mathrm{SimEQ}(A_1, A_2), \tag{4}$$

where α, β, and γ are the weights for SimName, SimDesc, and SimEQ, respectively. The values of these three parameters will be discussed in Subsection 4.2.3.

Since each unmatched API has a degree of API functional correlation with every ground-truth API, these paired data can be organized to generate a $m \times n$ matrix. Figure 4 shows an example of API functional correlation in a $m \times n$ matrix. The first column in the matrix denotes the unmatched APIs, the APIs are represented as $R_1, \ldots, R_m$. The first row represents the ground-truth APIs as $T_1, \ldots, T_n$. S_{ij} refers to the value of functional correlation between R_i and T_j. For each R_i in the recommended results, it has n degrees of functional correlation with the n APIs in ground-truth results. In our approach, we select the highest value of functional correlation for each unmatched API, i.e., the biggest number in each row. The matched ground-truth APIs are removed during the improvement of correctness. To avoid the situation that two or more unmatched APIs in the same result are all marked as correlated with the same ground-truth API, the selected highest functional correlation for each row should not lie in the same column. If two unmatched APIs both have strong functional correlation with a specific ground-truth API, we will select the one with higher degree of functional correlation. The lower degree of functional correlation will be set to 0 while reranking in that row. This checking method will be triggered when a new value of functional correlation is selected. Finally, each unmatched API has a degree of functional correlation with an unique ground-truth API. The degree of functional correlation ranges from 0 to 1, it can be treated as a intermediate state between matched results and unmatched results.

For the selected specification-based API recommendation techniques in this paper, Precision and Recall are the most widely used indicators of correctness [9, 19, 20]. The traditional Precision and Recall are measured as follows:

$$\mathrm{Precision} = \frac{\#\text{Correct results}}{\#\text{Recommended results}}, \tag{5}$$

$$\mathrm{Recall} = \frac{\#\text{Correct results}}{\#\text{Ground-truth results}}, \tag{6}$$

where Correct results represent the matched APIs in recommended results, Recommended results denote the sets of candidate APIs generated by API recommendation techniques, and Ground-truth results denote the sets of ground-truth APIs.

Our improved approach on the measurement of correctness is designed to make use of the contribution of unmatched APIs. The improved measurement of $\text{Precision}_{\text{corr}}$ and $\text{Recall}_{\text{corr}}$ are calculated according to the following formulas:

$$\text{Precision}_{\text{corr}} = \frac{\#\text{Correct results} + \sum \text{Functional correlation}}{\#\text{Recommended results}}, \tag{7}$$

$$\text{Recall}_{\text{corr}} = \frac{\#\text{Correct results} + \sum \text{Functional correlation}}{\#\text{Ground-truth results}}, \tag{8}$$

where $\text{Precision}_{\text{corr}}$ and $\text{Recall}_{\text{corr}}$ are measured based on the combination of matched APIs and contribution of unmatched APIs. In the traditional measurement, the matched API contributes 1 to the value and the unmatched API contributes 0 to the value. In our improved measurement, the matched API also contributes 1 to the value, but the contribution of unmatched API is calculated according to the API functional correlation. The functional correlation comes from API functional equivalence and similarity which we have discussed above. Since the value of functional correlation is never less than 0, the values of our metrics are greater than the traditional values in most cases. The improved measurements are expected to perform closer to the online evaluation.

4.2.3 *Calculation of the weights and threshold values*

In our approach, there are two threshold values in (2) and (3), and three parameters of weights in (4). To calculate the five parameters, we apply a greedy weight tuning algorithm to adjust the values iteratively. The detailed steps are shown in Algorithm 1. The evaluation of API recommendation in the user study from Subsection 3.2 is collected as the target of the fitting. The contribution of a little useful but unmatched APIs is quantized as 0.5 and the contribution of very useful APIs is recorded as 1.0.

Algorithm 1 Greedy weight tuning algorithm

```
Require: numIter =number of iterations.
 1: weights = {0.0, 0.0, 1.0, 1.0, 1.0};
 2: maxEvalScore = 0;
 3: valForMax = 0;
 4: for i = 0 to numIter do
 5:    for j = 0 to weights.Length do
 6:       for k = 1.0 to 0.0 by 0.01 do
 7:          weights[j] = k;
 8:          evalScore = eval();
 9:          if maxEvalScore > evalScore then
10:             maxEvalScore = evalScore;
11:             valForMax= k;
12:          end if
13:       end for
14:       weights[j] =valForMax;
15:    end for
16: end for
17: return weights.
```

The numIter (number of iterations) is subjectively set as 10 in our greedy weight tuning algorithm. The initial values of parameters α, β, γ, δ, and θ are set as 0.0, 0.0, 1.0, 1.0, and 1.0. In each iteration, we go through the five parameters and check the 101 possible values (Lines 6–14). The function eval() is used to check the effectiveness of a specific parameter with respect to the criteria from online evaluation. After the 10 iterations, the weights and threshold values are set as follows: $\alpha = 0.24$, $\beta = 0.31$, $\gamma = 0.45$, $\delta = 0.67$, and $\theta = 0.58$. These five parameters are used in the measurements of improved Precision and Recall.

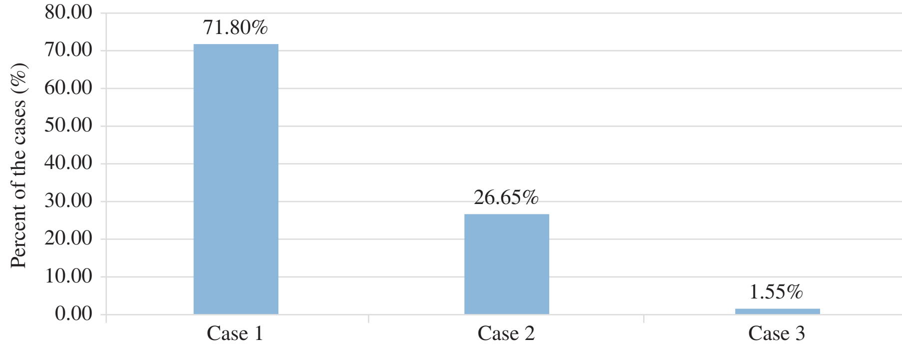

Figure 5 (Color online) Statistics of the difference between traditional and improved measurements.

5 Experiments and results

For the specification-based API recommendation techniques, correctness of results is commonly represented as Precision and Recall [9, 19, 20]. Our improved measurements of Precision and Recall aim to reflect the substantial quality of the recommended results on the basis of API functional correlation. To evaluate the effectiveness of $\text{Precision}_{\text{corr}}$ and $\text{Recall}_{\text{corr}}$, we apply the measurements on the subjects selected in our previous user study. The studied subjects contain 4455 valid recommended results. Based on these results, we want to answer the following research question.

• RQ3: How does the improved measurements of Precision and Recall perform in the evaluation of API recommendation?

For a given specification which is extracted from feature requests, we can obtain several recommended results generated by the existing work [19]. We calculate Precision and $\text{Precision}_{\text{corr}}$, Recall and $\text{Recall}_{\text{corr}}$ for each result. The candidate APIs are sorted according to the values of traditional measurements and improved measurements. We can obtain two sequences of results for the measurements of Precision and Recall separately. Since the denominators in the traditional measurements and improved measurements are the same, the variation trends of Precision and Recall are actually the same for the traditional and improved measurements. Our analysis is built on the comparison of the two sequences of results ordered by traditional and improved measurements. Obviously, there are 3 types of cases for the difference between two sequences:

• Case 1. The orders of the two sequences are the same;

• Case 2. The orders of the two sequences are different, and the traditional measurements of the sub-sequences are the same;

• Case 3. The orders of the two sequences are different, and the traditional measurements of the sub-sequences are also different.

The sub-sequence we mentioned above means a pair of APIs in a specific result. To analyze the improvements in detail, we focus on each pair of APIs in the results whose orders are different between sequences by traditional and improved measurements. Case 1 denotes that the two measurements are actually the same. Case 2 denotes our approach can identify the different degrees of effectiveness of recommended results and the traditional measurements can not achieve it. Case 3 denotes that the two measurements have the opposite results, one of the studied measurement goes wrong. In the experiments, we record the detailed recommendation data for each case, and the results are shown in Figure 5. In the figure, the three boxes represent the percentages of the three cases in our analysis. From the figure, we have the following observations.

First, the two measurements produce the same evaluation for the majority of recommended results, i.e., 71.80% in out experiments. This finding means that our approach cannot improve the evaluation in most cases. It is acceptable for us, because the traditional Precision and Recall are used in decades, they have the basic ability of evaluation. The calculation of API functional correlation is heuristic. These factors result in the finding that the improvement of our approach on correctness measurement is not widespread.

Table 3 The recommendation results on HBase-18646

Feature request	HBase-18646
Recommended results	[‘LoggerFactory.getLogger’, ‘Arrays.asList’, ‘Path.toString’, ‘Configuration.setBoolean’, ‘Configuration.set’]
Ground-truth results	[‘LogFactory.getLog’]
Precision and Recall	0.0 and 0.0
$\text{Precision}_{\text{corr}}$ and $\text{Recall}_{\text{corr}}$	0.154 and 0.77

Second, our approach has greater degree of discrimination than the traditional measurement for 26.65% recommended results. This finding confirms that our approach can identify the contribution of unmatched APIs. To make the finding convinced, we analyze 118 out of 1187 recommended results manually to check whether our approach improves the evaluation. The selection of 118 results is executed randomly. According to our analysis, the improvements are miscalculated on 6 out of 118 results. The miscalculation comes from the functionally independent sentences in the API documentation. For the other results, our approach can identify the contribution of unmatched APIs. Table 3 presents the recommendation results and measurements for HBase-18646, i.e., a feature request in our experiment. In this case, the value of traditional precision is 0 and our approach find that LoggerFactory.getLogger and logFactory.getLog can be used to implement the same task of logging. The value of functional correlation is 0.154, which mainly comes from the similarity of API name and description. According to the traditional measurement, this recommended result is marked equivalent to other unmatched results. Our improved measurement can identify its effects, resulting in a different sequence of recommended results. Actually, all the subsequences of recommended results in case 2 obtain 0 in terms of traditional measurement. Some 0-valued unmatched results can obtain a new value by applying our approach.

Third, our approach may generate opposite conclusion of evaluation in rare cases, i.e., 1.55% in our experiments. We analyze all the 69 results and find that there exists a trade-off question in these kinds of cases, i.e., how many unmatched-but-useful APIs can contribute equivalently to correctness comparing with a matched API. To our experience, there is no general answer for this question. In the 69 results, the contribution of several unmatched APIs is greater than 1, and the contribution of matched API is set to 1. In practice, we will report these cases to developers for further online evaluation to obtain a accurate result. Since case 3 is relative rare in the study, the further online evaluation will not cost too much. The evaluation on these results are subjective, one result may receive different feedbacks from different developers.

Finding 3. Our approach can help to identify the contribution of unmatched APIs to correctness in terms of precision and recall. There exist rare cases that the contribution of several unmatched APIs is considered greater than a matched API, the rare cases need further online evaluation to confirm.

6 Threats to validity

Threats to construct validity. The main threat to construct validity is the features which used to measure API functional correlation, i.e., “see also” relationship in API knowledge graph, similarity of API name and description. Although the features are examined in the experiments, we cannot infer that they are effective for all the subjects. To reduce this threat, we will further investigate the relationship between other features and API functional correlation.

Threats to internal validity. The main threat to internal validity is the potential faults in the process of exploratory study and reimplementation of an existing API recommendation technique [19]. To reduce this threat, the first two authors carefully reviewed all the related code, configurations, scripts and subject projects.

Threats to external validity. The feature requests and the corresponding API methods could pose threats to external validity. To reduce this threat, we try to collect the data according to the constraints list in original work. We will conduct the study with more kinds of requirements and API recommendation techniques in the future.

7 Conclusion

In this paper, we conduct an exploratory study to analyze the contribution of unmatched results to correctness. We find some unmatched results can also benefit to programming through a user study, and these APIs are commonly correlated with ground-truth APIs in terms of functionality. The API functional correlation can be measured by extracting information of API relationships from API knowledge graph, API method name and API documentation. Furthermore, we propose improved measurement of correctness by considering API functional correlation in the traditional calculation of correctness. We evaluate the approach on the API recommendation attempts based on 6141 specifications. The results show that our approach can help to identify the contribution of 28.2% unmatched APIs to correctness in our study. The current measurement of API functional correlation is heuristic, resulting in the possible missing of useful unmatched APIs. In the future, we will try to extract information of API relationships from more sources and apply our approach on more API recommendation techniques and more subjects.

Acknowledgements This work was supported in part by National Key R&D Program of China (Grant No. 2018YFB100-3900), in part by National Natural Science Foundation of China (Grant Nos. 61402103, 61572126, 61872078), in part by Open Research Fund of Key Laboratory of Safety-Critical Software Fund (Nanjing University of Aeronautics and Astronautics) (Grant No. NJ2019006), and in part by Key Laboratory of Computer Network and Information Integration of the Ministry of Education of China (Grant No. 93K-9).

References

1 Xia X, Bao L F, Lo D, et al. What do developers search for on the web? Empir Softw Eng, 2017, 22: 3149–3185
2 Bao L F, Xing Z C, Wang X Y, et al. Tracking and analyzing cross-cutting activities in developers' daily work. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, 2015. 277–282
3 Lv C, Jiang W, Liu Y, et al. APISynth: a new graph-based API recommender system. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 596–597
4 Thung F. API recommendation system for software development. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 896–899
5 Yu H B, Song W H, Mine T. APIBook: an effective approach for finding APIs. In: Proceedings of the 8th Asia-Pacific Symposium on Internetware, 2016. 45–53
6 Robillard M, Walker R, Zimmermann T. Recommendation systems for software engineering. IEEE Softw, 2010, 27: 80–86
7 Rahman M M, Roy C K, Lo D. Rack: automatic API recommendation using crowdsourced knowledge. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, 2016. 349–359
8 Raghothaman M, Wei Y, Hamadi Y. Swim: synthesizing what I mean-code search and idiomatic snippet synthesis. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 357–367
9 Xu C Y, Sun X B, Li B, et al. MULAPI: improving API method recommendation with API usage location. J Syst Softw, 2018, 142: 195–205
10 Proksch S, Amann S, Nadi S, et al. Evaluating the evaluations of code recommender systems: a reality check. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 111–121
11 Nguyen A T, Nguyen T N. Graph-based statistical language model for code. In: Proceedings of the 37th IEEE International Conference on Software Engineering, 2015. 858–868
12 Asaduzzaman M, Roy C K, Schneider K A, et al. A simple, efficient, context-sensitive approach for code completion. J Softw Evol Proc, 2016, 28: 512–541
13 Thung F, Lo D, Lawall J. Automated library recommendation. In: Proceedings of the 20th Working Conference on Reverse Engineering, 2013. 182–191
14 Li J, Wang Y, Lyu M R, et al. Code completion with neural attention and pointer networks. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, 2018. 4159–4165
15 Yuan W Z, Nguyen H H, Jiang L X, et al. LibraryGuru: API recommendation for Android developers. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 364–365
16 Cao B Q, Liu X Q, Rahman M, et al. Integrated content and network-based service clustering and web APIs recommendation for mashup development. IEEE Trans Serv Comput, 2020, 13: 99–113
17 Beel J, Genzmehr M, Langer S, et al. A comparative analysis of offline and online evaluations and discussion of research paper recommender system evaluation. In: Proceedings of International Workshop on Reproducibility and Replication in Recommender Systems Evaluation, 2013. 7–14
18 Manning C, Raghavan P, Schütze H. Introduction to information retrieval. Nat Lang Eng, 2010, 16: 100–103
19 Thung F, Wang S W, Lo D, et al. Automatic recommendation of API methods from feature requests. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, 2013. 290–300
20 Thung F, Oentaryo R J, Lo D, et al. WebAPIRec: recommending web APIs to software projects via personalized ranking. IEEE Trans Emerg Top Comput Intell, 2017, 1: 145–156

21 Nguyen A T, Hilton M, Codoban M, et al. API code recommendation using statistical learning from fine-grained changes. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 511–522

22 Shani G, Gunawardana A. Evaluating recommendation systems. In: Recommender systems handbook. Berlin: Springer, 2011. 257–297

23 Huang Q, Xia X, Xing Z C, et al. API method recommendation without worrying about the task-API knowledge gap. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 293–304

24 Nguyen A T, Hilton M, Codoban M, et al. API code recommendation using statistical learning from fine-grained changes. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 511–522

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

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

27 Weimer M, Karatzoglou A, Le V, et al. COFIRANK: maximum margin matrix factorization for collaborative ranking. In: Proceedings of Annual Conference on Neural Information Processing Systems, 2007. 222–230

28 Weimer W, Fry Z P, Forrest S. Leveraging program equivalence for adaptive program repair: models and first results. In: Proceedings of the 28th International Conference on Automated Software Engineering, 2013. 356–366

29 Levenshtein V. Binary codes capable of correcting insertions and reversals. Sov Phys Dokl, 1966, 10: 707–710