IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 18, 2019, accepted April 5, 2019, date of publication April 11, 2019, date of current version April 25, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910732

Enabling Feature Location for API Method Recommendation and Usage Location

XIAOBING SUN[1,2], (Member, IEEE), CONGYING XU[1], BIN LI[1], YUCONG DUAN[3], AND XINTONG LU[1]

[1]School of Information Engineering, Yangzhou University, Yangzhou 225009, China
[2]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
[3]College of Information Science and Technology, Hainan University, Hainan 570228, China

Corresponding author: Bin Li (lb@yzu.edu.cn)

This work was supported in part by the Natural Science Foundation of China under Grant 61872312, Grant 61472344, and Grant 61402396, in part by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant KFKT2018B12, in part by the Jiangsu Qin Lan Project, in part by the Jiangsu 333 Project, and in part by the Jiangsu Overseas Visiting Scholar Program.

ABSTRACT Given a new feature request during software evolution, developers are used to employing existing third-party libraries and APIs for implementation. However, it is usually non-trivial to find suitable APIs and to decide where to use these APIs in the original software. In this paper, we develop an approach for recommending API methods and usage locations through mining various software repositories. First, we analyze software repositories and use the feature location technique to localize feature-related files as API usage locations. Then, we utilize the feature-related files and API libraries to identify potential API methods for the implementation of the incoming feature request. We evaluate our approach on 5000 feature requests selected from five Java projects, and the results demonstrate that our approach can achieve 29.7% and 9.6% improvement in terms of Hit@5 and Hit@10 in recommending API methods, compared with the existing approach. For API usage localization, our approach can get MAP, MRR, Hit@1, HIt@5, and Hit@10 values with 0.293, 0.434, 0.303, 0.602, and 0.685, respectively.

INDEX TERMS Feature implementation, API recommendation, API usage location, feature location, mining software repositories.

I. INTRODUCTION

Developers always face the challenge of implementing a huge amount of feature requests to meet various users' requirements during software evolution. To implement a feature request, developers are used to employing third-party libraries or Application Programming Interfaces (APIs) to save the implementation time [1], [2].

However, in order to effectively use third-party libraries or APIs, developers need to spend a lot of time in understanding the methods and classes in the original software. At present, more than 2.6 million Java third-party libraries are stored in Maven Central Repository.[1] It is non-trivial for developers to find suitable methods from such a huge amount of libraries for usage. Moreover, when using APIs for maintenance, developers have to go through a tedious process to identify the locations of related API methods or classes in the original software. After these tedious steps, developers can utilize these libraries or APIs in the original software to finish the incoming feature requests implementation.

To alleviate above challenges, lots of work have been devoted to API recommendation [3]–[9]. Thung *et al.* proposed an approach to recommend API methods by mining historical feature request repositories and API libraries [3]. They measured the similarity between the descriptions of feature requests and API methods to recommend potential API methods for implementing the incoming feature request. Our previous work extends Thung et al.'s work, and proposed an approach called *MULAPI* [10], to improve the accuracy of API method recommendation based on API usage location. In addition to recommend API methods, *MULAPI* also recommends API usage locations to guide developers for feature implementation. Specifically, *MULAPI* uses the feature location technique [11] to identify the API usage locations. However, when using *MULAPI*, there are nearly 20 parameters to set to recommend API locations, which is not well for its generalizability.

The associate editor coordinating the review of this manuscript and approving it for publication was Hailong Sun.

[1]http://mvnrepository.com/repos/central

In this paper, we attempt to alleviate the complexity of using *MULAPI*, and design a simplified technique based on *MULAPI*. Specifically, we simplify the *MULAPI* by using fewer parameters (from 17 to 2) during the API location recommendation process. In this way, the complexity of the technology is reduced and the generalizability is improved. In addition, with respect to information retrieval technology for computing semantic similarity, we employ *CNN_NPL*[2] technology to improve the accuracy.

We evaluate our approach on 5000 feature requests stored in JIRA[3] issue tracking system from five Java software projects (''Axis2/ Java'', ''CXF'', ''Hadoop Common'', ''Hbase'' and ''Struts2''), and use Hit@N, MAP and MRR to evaluate our approach. Our study shows that our approach achieves 29.7% and 9.6% improvement in terms of Hit@5 and Hit@10, respectively in recommending API methods, compared with the state-of-the-art approach [3]. For recommending API usage locations, our approach can get MAP, MRR, Hit@1, HIt@5 and Hit@10 values of 0.293, 0.434, 0.303, 0.602 and 0.685, respectively.

The rest of this paper is organized as follows. In Section II, we present the technique of feature location and natural language processing. In Section III, we introduce our approach. We describe the empirical study in Section IV, and followed by discussing the related work in Section V. Finally, We conclude our work in Section VI.

II. PRELIMINARIES

A. FEATURE LOCATION

In our work, we use the feature location technique to help recommend API usage locations. Feature location takes a feature request as input, and maps the feature request to the source code, i.e., files, classes, methods, code units, etc. [11]. In the JIRA system, there are several fields for a feature request,[4] i.e., description, components, reporter, etc. Then, based on the software repository mining technique, a set of feature related files are generated as the locations to implement the feature request [12].

In this paper, we use the information retrieval technique to mine the software repository to implement the feature location technique and identify the feature related code files. First, we parse source code files into abstract syntax trees to extract the static source code, i.e., classes, methods, comments, etc. Then, we make use of the natural language processing technique to compute relevance between feature requests and source code files. In addition, we mine historical request repository to search for similar requests. Based on the similar historical requests, we can identify their corresponding revised files as relevant to the incoming feature request.

B. NATURAL LANGUAGE PROCESSING

The information of a feature request and those stored in software repositories are mostly expressed with the natural language format. So we employ the natural language processing (NLP) technique to deal with these information.

In our approach, we employ the *CNN_NLP* released by *Baidu AI*[5] for natural language processing. The *CNN_NLP* is shown to be effective to compute the similarity with a high accuracy for short texts, which is suitable for the data in the software repositories. The interface of *CNN_NLP* is implemented based on the neural network semantic matching model with supervised massive data training, which is called *SimNet* framework. The *SimNet* framework uses deep neural network structure, including input layer, presentation layer, matching layer, and output layer. The input layer is based on the sequence of words, and the word sequence is then converted to a word embedding sequence. The presentation layer is to construct the word-to-sentence representation and transform it into one or more low-dimensional dense semantic vectors with global web information. The matching layer uses the expression vector of the text to perform the interactive calculation. Based on different application scenarios, two matching algorithms (representation-based matching and interaction-based matching) are developed and finally the matching score of two inputs is generated.

We mainly employ the *CNN_NLP* interface to compute semantic similarity between two messages in the software repositories. Given two messages $M1$ and $M2$, we input $M1$ and $M2$, and then the interface returns their semantic similarity result. More details can refer to their web page.[6] In the following parts of this paper, we compute the similarity of two messages as follows:

$$SimilarityScore(M1, M2) = CNN\_NLP(M1, M2) \quad (1)$$

In this paper, our approach employs this *CNN_NLP* interface to process natural language information and computes the similarity result between two messages.

III. OUR APPROACH

Figure 1 shows the process of our approach. The input includes an incoming feature request, source code repository, historical feature request repository, and API libraries. First, we mine the source code repository and historical feature request repository to extract feature related files and historical similar requests which are related to the new feature request. Then, these feature related files and similar feature requests are processed by employing the feature location technique to identify more accurate feature related files used as API usage locations recommended to developers. Finally, historical similar requests, API usage locations and API libraries are respectively analyzed to recommend API methods. By doing this, three different lists of API methods are recommended by mining three different data sources. In this way, a ranked list

[2]http://ai.baidu.com/tech/nlp/simnet

[3]https://www.atlassian.com/software/jira

[4]https://issues.apache.org/jira/secure/ShowConstantsHelp.jspa?decorator=popup#IssueTypes

[5]http://ai.baidu.com

[6]http://ai.baidu.com/tech/nlp

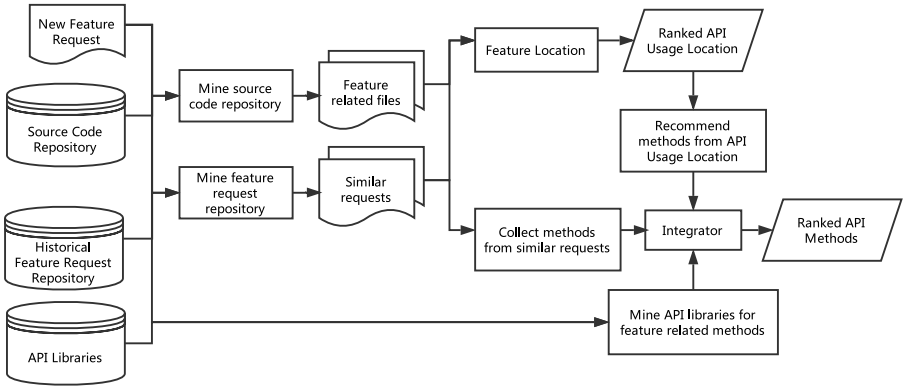

FIGURE 1. The process of our approach.

of API methods is recommended by combining these three different lists of API methods.

A. MINING SOFTWARE REPOSITORIES

1) MINING SOURCE CODE REPOSITORY

For API usage location, we have to analyze the source code repository to identify feature related files. We extract the key information from the source code files by parsing the source code files into AST (abstract syntax tree) [13], [14], and then extract their comments, class names, method names and variable names. To compute the similarity between a source code file and a feature request, we extract the summary and description in the incoming feature request and use the NLP (natural language processing) technique described in Section II-B to process them. We further analyze the natural language text and programming language text in the source code files, respectively [15]. Natural language text contains comments from the source code files, while programming language text mainly includes class names, method names and variable names. Given a new feature request FR, the feature relevant value of a source code file f is computed as follows:

$$SCFScore\,(FR, f) = a_1 \times CNN\_NLP\,(f_{cmvn}, FR_{SD}) + a_2 \times CNN\_NLP\,(f_{co}, FR_{SD}) \quad (2)$$

f_{cmvn} refers to the content of programming language text in the source code file f; f_{co} refers to the comments in the source code file f. FR_{SD} is the summary and description of a feature request FR; $CNN\_NLP\,(f_{cmvn}, FR_{SD})$ is the similarity value between f_{cmvn} and FR_{sd} computed by the CNN natural language processing technique. We set weights a_1 and a_2 for $CNN\_NLP\,(f_{cmvn}, FR_{SD})$ and $CNN\_NLP\,(f_{co}, FR_{SD})$. The values of them are set as 0.5 by default, and the sum of a_1 and a_2 is 1. It aims to limit the maximum value of $SCFScore$ to be no more than 1.

At this step, we can estimate the correlation between a source code file and a feature request via the $SCFScore$ value of the corresponding file based on mining the source code repository.

2) MINING HISTORICAL FEATURE REQUEST REPOSITORY

During the implementation process of a feature request, some files must be co-changed and the API methods will be applied to implement it [3], [16]. In addition, there may be some similar historical feature requests that have been implemented. At this time, we can refer to revisions of these similar historical feature requests to use similar API methods in relevant or the same code files. So identifying these similar historical requests is helpful to recommend API methods and their usage locations.

In the process of identifying similar historical requests, we refer to Thung et al.' approach, which obtains the similarity value between two feature requests by combining different fields in the feature request, i.e., summary, description, reporter, components and priority. Our approach further considers another field, i.e., 'linked-issues', which is necessary and useful to identify similar historical feature requests.

Then, we compute the similarity value between two feature requests based on their respective fields, and combine them into an aggregated value. For each field in the feature request, their similarity values are computed as follows.

Summary and Description: The information of these two fields are mostly natural language text. We employ the CNN natural language processing technique described in Section II-B to compute the similarity values:

$$SDScore(FR1, FR2) = CNN\_NLP(FR1_{SD}, FR2_{SD}) \quad (3)$$

$FR1_{SD}$ and $FR2_{SD}$ represent the content in the summary and description field for $FR1$ and $FR2$, respectively.

Components: When a feature request is implemented, some components in the original software may be revised correspondingly. The similarity value of this field is computed as follows:

$$CompScore(FR1, FR2) = \frac{S_{FR1_C} \bigcap S_{FR2_C}}{\sqrt{NS_{FR1_C}} \times \sqrt{NS_{FR2_C}}} \quad (4)$$

S_{FR1_C} and S_{FR2_C} indicate the components of the incoming feature request $FR1$ and the $FR2$'s, and the numerator is the size of the intersection between them. NS_{FR1_C} and NS_{FR2_C}

represent the number of components related to $FR1$ and $FR2$, respectively.

Reporter: Reporters are users or developers who reported these requests. The similarity is computed as follows:

$$RepScore(FR1, FR2) = \begin{cases} 1 & (the\ same\ reporter) \\ 0 & (different\ reporters) \end{cases} \quad (5)$$

Priority: Each issue in the tracking system has a priority value (e.g. "Block", "Critical", "Major", "Minor" and "Trivial").[7] We assign value 1 for "Blocker", 2 for "Critical", 3 for "Major", 4 for "Minor" and 5 for "Trivial", respectively. The formula used to compute the 'priority' similarity value is as follows:

$$PrioScore(FR1, FR2) = \frac{1}{1 + |Prio_{FR1} - Prio_{FR2}|} \quad (6)$$

$Prio_{FR1}$ and $Prio_{FR2}$ indicate the values of feature $FR1$'s priority field and $FR2$'s, respectively.

Linked-issues: A new feature request is sometimes linked to previous issues. Given a new feature request $FR1$ and a historical feature request $FR2$, if the linked-issues field of $FR1$ contains $FR2$'s issuekey(ID), the value of $LinkScore(FR1, FR2)$ is 1, otherwise it is 0.

Finally, we combine the similarity values of the above fields to obtain an integrated similarity result as follows:

$$\begin{aligned} SimScore(FR1, FR2) = {} & b_1 \times SDScore(FR1, FR2) \\ & + b_2 \times CompScore(FR1, FR2) \\ & + b_3 \times RepScore(FR1, FR2) \\ & + b_4 \times PrioScore(FR1, FR2) \\ & + b_5 \times LinkScore(FR1, FR2) \end{aligned} \quad (7)$$

$SDScore(FR1, FR2)$, $CompScore(FR1, FR2)$, $RepScore(FR1, FR2)$, $PrioScore(FR1,FR2)$ and $LinkScore(FR1,FR2)$ represent the similarity values between $FR1$'s and $FR2$'s summary and description, components, reporter, priority and linked-issues, respectively. b_1-b_5 are weights of each field contributing to the $SimScore(FR1, FR2)$.

3) MINING API LIBRARIES

A direct and convenient way of recommending API methods is searching API libraries to identify relevant API methods. In the API libraries, API documents contain textual description for explaining each API in the library. Developers often investigate into API documents to identify suitable API methods to implement a feature request. So we mine API libraries through identifying the relevance between API description and feature request text. The details will be described in Section III-C.2-3.

B. FEATURE LOCATION

We employ the feature location technique to localize feature related files to obtain the API usage locations. For implementing the new feature request, developers need to modify some feature related files. In our approach, we make use of the feature location technique to get these related files as API usage location. Feature location is a process to identify the potential locations in the software to implement an incoming feature request. In our approach, we localize feature related files from two aspects. First, we mine source code repository through computing the semantic similarity of text in feature requests and source code files. Then, we search for similar *closed* or *resolved* feature requests in historical feature request repository, and extract the files modified for implementing these similar feature requests. Finally, we combine the results from the above two steps to get the list of potential API usage locations.

Mining source code repository has been described in Section III-A, and other steps of feature location are shown as follows:

1) COLLECTING FEATURE RELATED FILES FROM SIMILAR HISTORICAL REQUESTS

During the process of implementing a feature request, some feature related files are modified. Similar historical feature requests may involve modification to the same or relevant files. Thus, we can collect those files which are involved in implementing those similar feature requests as the feature location result.

According to the step described in Section III-A, we can identify the top-k most similar feature requests from historical feature request repository. After that, we compute the feature relevant value for each candidate file based on the top-k similar feature requests. Given a file f, we compute the feature relevant value for a new feature request FR as follows:

$$SFRScore\,(FR, f) = \frac{Count_f}{k} \quad (8)$$

$Count_f$ indicates the number of similar feature requests that the file f was modified for their implementation.

2) INTEGRATING FEATURE LOCATION RESULTS

After the above steps, we get two sets of values for feature related files, i.e., $SCFScore\ (FR, f)$ by mining source code repository and $SFRScore\ (FR, f)$ by mining historical feature repository. To generate the final API usage locations, we combine the two sets of values as follows:

$$\begin{aligned} FLScore\,(FR, f) = {} & a \times SCFScore\,(FR, f) \\ & + b \times SFRScore\,(FR, f) \end{aligned} \quad (9)$$

a and b are the weights for source code repository based value and similar historical request repository based value, respectively. The source code file with the highest $FLScore\ (FR, f)$ value is viewed as the most suitable API usage location recommended to developers.

C. API METHOD RECOMMENDATION

To identify potential API methods for implementing the incoming feature request, we mine three different data

[7] https://issues.apache.org/jira/secure/ShowConstantsHelp.jspa?decorator=popup#IssueTypes

sources, and extract three different sets of feature related API methods. Then, we integrate the three sets of methods into a list of potential API methods, which will be recommended to developers for implementing features.

1) RECOMMENDING API METHODS FROM API USAGE LOCATIONS

In this step, we recommend API methods based on the API usage location recommendation as shown in Section III-B. Given a new feature request FR, its relevant files f and an API method m, we compute API's feature relevant value as follows:

$$RecScore^{L}(FR, m) = \alpha_1 \times CNN\_NLP(f_{nl}, m_d) + \alpha_2 \times CNN\_NLP(f_{pl}, m_d) \quad (10)$$

f_{nl} and m_d are the natural language text in the file f and method m, respectively; f_{pl} is the programming language code in f. The weights for similarity of natural language part and programming language part are α_1 and α_2, respectively. The values of them are set as 0.5 by default, and the sum of a_1 and a_2 is 1, which aims to limit the maximum value of $RecScore^{L}(FR, m)$ to be no more than 1.

2) COLLECTING API METHODS FROM SIMILAR HISTORICAL FEATURE REQUESTS FOR RECOMMENDATION

In this step, we extract API methods that were involved to implement these feature requests.

We compute feature relevant values for each API method based on similar historical feature requests by using the following equation.

$$RecScore^{H}(FR, m) = \frac{Count_m}{k} \quad (11)$$

$Count_m$ represents the number of similar feature requests that the method m was used for these features' implementation, and k shows the number of similar historical feature requests.

3) MINING API LIBRARIES FOR FEATURE RELATED METHODS

To use API methods for implementing a feature request, the corresponding API descriptions in the API libraries should be relevant to this feature request. So we also need to mine API libraries to get the relevance of the API method to the incoming feature request.

For a new feature request FR and API documents, we integrate the summary and description fields of the new feature request into a whole textual format. Then, we compute the similarity between the feature request and API documents with the CNN natural language processing technique. The feature relevant value between FR and method m is computed in the following way:

$$RecScore^{D}(FR, m) = CNN\_NLP(FR_d, m_d) \quad (12)$$

FR_d is the description text in the feature request FR, and m_d is the description text of method m.

4) INTEGRATOR

Finally, we combine three sets of results generated above to get the final values for all potential API methods, as follows:

$$RecScore^{LHD}(FR, m) = \alpha \times RecScore^{L}(FR, m) + \beta \times RecScore^{H}(FR, m) + \gamma \times RecScore^{D}(FR, m) \quad (13)$$

α, β and γ represent the weights of $RecScore^{L}(FR, m)$, $RecScore^{H}(FR, m)$ and $RecScore^{D}(FR, m)$ contributing to the API recommendation by mining different software repositories, respectively. To set appropriate values of these weights, we employ a greedy approach based on Gibbs sampling [17], which is shown in Section IV-D. Finally, API methods are ranked and recommended based on the $RecScore^{LHD}(FR, m)$ values.

IV. EMPIRICAL STUDY

In this section, we show the evaluation of our approach. First, we introduce the used dataset and evaluation metrics. Next, we propose our research questions, followed by describing our experimental methodology and environment. Finally, we present and discuss the empirical results.

A. DATASET

To evaluate our approach effectively and compare with the state-of-the-art approach fairly, we choose the same projects in JIRA[8] that are used by Thung et al., including "*Axis2/Java*", "*CXF*", "*Hadoop Common*", "*Hbase*" and "*Struts2*". *Axis2/Java* [9] is a Web Services/SOAP/WSDL engine, the successor to the widely used Apache Axis SOAP stack. It has integrated support for the widely popular REST style of Web services. *CXF*[10] is an open source services framework, which helps developers build and develop services using frontend programming APIs, like JAX-WS and JAX-RS. These services can speak a variety of protocols such as SOAP, XML/HTTP, or RESTful HTTP and can work over a variety of transports such as HTTP, JMS or JBI. *HadoopCommon*[11] is one of the modules in the Apache Hadoop project, which includes the common utilities to support distributed processing of large data sets across clusters of computers using simple programming models for reliable, scalable, distributed computing. *Hbase*[12] is an open-source, distributed, versioned, non-relational database modeled for the Hadoop database, a distributed, scalable, big data store. *Struts2*[13] is a free, open-source, MVC framework for creating elegant, modern Java web applications, which supports convention over configuration. Moreover, it is extensible by using the plugin architecture, and ships with plugins to support REST, AJAX and JSON.

[8] https://www.atlassian.com/software/jira
[9] http://axis.apache.org/axis2/java/core/
[10] http://cxf.apache.org
[11] http://hadoop.apache.org
[12] http://hbase.apache.org
[13] http://struts.apache.org

TABLE 1. Five apache subject projects for evaluation.

Subjects	Period
Axis2/Java	2005/9/5 - 2017/1/5
CXF	2011/7/18 - 2017/7/20
Hadoop Common	2012/11/21 - 2017/7/24
Hbase	2014/8/14 - 2017/7/22
Struts2	2006/1/28 -2017/7/19

To evaluate the effectiveness of API method and usage location recommendation in our study, we chose the latest 1000 *closed* or *resolved* feature requests of each project, as Table 1 shows. For collecting methods from historical similar requests, we also collected commit logs from Git repositories of five projects. In addition, to collect the used API methods in these projects, we refer to the work of Thung et.al., and employ API methods from ten widely used third-party libraries with the latest version in Apache projects, i.e., commons-codec,[14] commons-io,[15] commons-lang,[16] commons-logging,[17] easymock,[18] junit,[19] log4j,[20] servlet-api,[21] slf4j-api,[22] and slf4j-log4j12.[23]

B. EVALUATION METRICS

To evaluate the effectiveness of our approach for API recommendation, we use the following metrics:

1) MEAN AVERAGE PRECISION (MAP)

MAP is the mean value of AP (average precision) results over the evaluated feature requests, which takes all feature related methods into account to show the overall recommendation, defined as the following formula:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ correctly\ recommended\ elements} \tag{14}$$

M indicates the number of all the recommended results; k represents the position in the recommendation list; $pos(k)$ indicates whether the result in this position (k) is truly recommended (1 if the result is truly recommended, otherwise 0.), and $P(k)$ is the ratio of correctly recommended results over top-k recommended results.

2) MEAN RECIPROCAL RANK (MRR)

The MRR indicates the reciprocal of the position of the first feature relevant method, which is defined as the following formula:

$$MRR = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{rank_i} \tag{15}$$

Q refers to the number of recommendation results; $rank_i$ is the position of the first feature related method.

3) TOP-K RANK (HIT@N)

Hit@N is used to calculate the number of feature requests where one of the related methods appears in the top N (i.e., 1, 5, 10) results. For a feature request, if at least one of its related methods occurs in the top N results, we consider that the recommendation is effective, and the value of top N is 1, otherwise it is 0.

4) GAIN

Gain is used to show the improvement between two results generated by two approaches:

$$Gain = \frac{R_2 - R_1}{R_1} \times 100\% \tag{16}$$

R_1 and R_2 refer to the computation results of two approaches, respectively.

C. RESEARCH QUESTIONS

To show the effectiveness of our approach, we propose the following three research questions.

1) RQ1: DOES OUR APPROACH OUTPERFORM THE STATE-OF-THE-ART APPROACH IN API METHOD RECOMMENDATION?

Our approach extends Thung et al.'s approach by mining more software repositories and employing the feature location technique to recommend API methods. So we first investigate whether our approach can improve the effectiveness of API method recommendation over Thung et al.'s approach.

2) RQ2: IS OUR APPROACH EFFECTIVE IN RECOMMENDING API USAGE LOCATION?

Our approach recommends not only API methods as traditional API recommendation techniques, but also their probable usage locations. So we would like to show whether our approach can recommend accurate API usage locations for developers to use.

3) RQ3: HOW EFFECTIVE IS TO USE THE SOURCE CODE REPOSITORY FOR API RECOMMENDATION?

Our approach extends Thung et al.'s approach by mining more software repositories, i.e., source code repository. So we examine whether mining source code repository is useful for API recommendation.

D. METHODOLOGY AND ENVIRONMENT

We evaluated our approach by performing stratified ten-fold cross validations for all these five projects. Specifically,

[14] http://commons.apache.org /proper/commons-codec/
[15] http://commons.apache.org/proper/commons-io/
[16] http://commons.apache.org/proper/commons-lang/
[17] http://commons.apache.org/proper/commons-logging/
[18] http://easymock.org/api/
[19] http://junit.org/junit4/
[20] https://logging.apache.org/log4j/1.2/
[21] http://tomcat.apache.org/tomcat−7.0-doc/servletapi/
[22] http://www.programmingforfuture.com/2010/05/servlet-api.html
[23] https://www.versioneye.com/java/org.slf4j:slf4j-log4j12/1.7.25

we first selected 1000 feature requests for each project from their issue tracking systems, and divided them into ten groups. Then, nine groups were combined as the historical feature request repository, and one group was used as the set of new feature requests. The historical feature request repository was used to train parameters and the set of new feature requests was used to evaluate the effectiveness. Our study was conducted on a workstation with 24-core Intel Xeon processor (2.4 GHz) and 32GB RAM

To obtain appropriate values for different weights in our approach: α, β and γ in Equation (13), $b_1 - b_5$ in Equation (7), a and b in Equation (9), we employed a greedy approach based on Gibbs sampling [17] to iteratively execute for relatively optimal values of these weights. During each iteration, each weight is optimized independently (with 0.1 increment at each iteration from 0 to 1) and other weights are set as constants. Then, in the similar way, several other iterations were performed to further optimize the other weights. During the iteration process, we employed *MAP* as the goal for iterations. When the value of *MAP* is the highest and does not change any more, the iteration is finished. Finally, we obtain the values of different weights as shown in Table 2.

TABLE 2. Values of various weights based on Gibbs sampling.

Subjects	a	b	b_1	b_2	b_3	b_4	b_5
Axis2/Java	0.36	0.84	0.68	0.52	0.66	0.01	0.40
CXF	0.47	0.66	0.84	0.40	0.60	0.20	0.40
Hadoop Common	0.26	0.62	0.59	0.86	0.14	0.08	0.76
Hbase	0.27	0.68	0.91	0.1	0.32	0.1	0.94
Struts2	0.11	0.26	0.90	0.40	0.60	0.00	0.40
Average	**0.29**	**0.61**	**0.78**	**0.45**	**0.46**	**0.08**	**0.58**

To answer RQ1, we re-ran Thung et al.'s approach in the same environment with the same dataset and compared the two approaches with the same evaluation metrics defined above, i.e., *Hit*@5 and *Hit*@10 values. We also referred to Thung et al.'s evaluation approach to employ a 'gold standard set' criterion to process feature requests [3]. To answer RQ2, we validated our approach with 5000 feature requests from these five projects. We employed three evaluation metrics (*MAP*, *MRR* and *Hit*@*N*) to measure the effectiveness of our approach for API usage location recommendation, and these metrics were commonly used in existing works [18]–[21]. To answer RQ3, we tuned suitable weights for recommendation of API usage locations, which can reflect the contribution of the source code repository in recommending API methods.

E. EMPIRICAL RESULTS

1) RQ1: EFFECTIVENESS IN RECOMMENDING API METHODS

For API method recommendation, the comparative results of our approach and Thung et al.'s are shown in Table 3, which shows that, on average, the values of *Hit*@5 and *Hit*@10 are improved to 0.772 and 0.832, respectively. This means that our approach can recommend at least one feature related method with the accuracy of 77.2% and 83.2% in the top five and ten recommended API methods, respectively.

TABLE 3. Results of our approach VS thung et al.'s approach.

Subjects	Technique	Hit@5	Gain	Hit@10	Gain
Axis2/Java	Our approach Thung's approach	0.952 0.818	16.38%	0.960 0.909	5.61%
CXF	Our approach Thung's approach	0.614 0.563	9.05%	0.757 0.750	0.93%
Hadoop Common	Our approach Thung's approach	0.699 0.482	45.02%	0.778 0.660	17.88%
Hbase	Our approach Thung's approach	0.870 0.649	34.05%	0.937 0.863	8.57%
Struts2	Our approach Thung's approach	0.726 0.462	57.14%	0.727 0.615	18.21%
Average	Our approach Thung's approach	**0.772** 0.595	**29.7%**	**0.832** 0.759	**9.6%**

TABLE 4. Results of API usage location recommendation.

Subjects	MAP	MRR	Hit@1	Hit@5	Hit@10
Axis2/Java	0.234	0.409	0.296	0.550	0.627
CXF	0.180	0.283	0.137	0.438	0.562
Hadoop Common	0.363	0.509	0.375	0.684	0.754
Hbase	0.273	0.475	0.344	0.644	0.740
Struts2	0.416	0.492	0.364	0.692	0.740
Average	**0.293**	**0.434**	**0.303**	**0.602**	**0.685**

In addition, the average gain results of *Hit*@5 and *Hit*@10 for our approach is 29.7% and 9.6% over Thung et al.'s approach. For the *Struts2* project, the *Hit*@5 and *Hit*@10 values of our approach are improved by 57.14% and 18.21%, respectively. Moreover, for API method recommendation, the improvement scale of Hit@5 value is higher than Hit@10 as shown in Table 3, which means that useful API methods can be more easily identified for usage in the top five results with our approach. Based on these results, we can conclude that our approach is more accurate to recommend API methods compared with the state-of-the-art technique, i.e., Thung et al.'s approach, especially for the top five recommendation.

2) RQ2: EFFECTIVENESS IN RECOMMENDING API USAGE LOCATIONS

Table 4 shows the accuracy of the results of our approach in recommending API usage locations. The average values of Hit@1, Hit@5 and Hit@10 of our approach are 0.303, 0.602 and 0.685, respectively. With respect to the top one file, our approach can correctly identify the feature related files with the accuracy value between 13.7% and 37.5%. With respect to the results of top five recommended usage locations, our approach can correctly identify the feature related files with the accuracy value between 43.8% and 69.2%. With respect to the top ten recommended locations, our approach can recommend at least one related location with the accuracy value between 56.2% and 75.4%. From the MAP and MRR results, we notice that their average values can reach 0.293 and 0.434, respectively. So from the results discussed above, we can conclude that the API usage location recommendation is also effective, which can help developers find the locations where they will implement the incoming feature request.

TABLE 5. Weights of recommendation by mining different software repositories in our approach.

Subjects	α	β	γ
Axis2/Java	1.00	1.00	0.10
CXF	1.00	0.99	0.28
Hadoop Common	0.49	1.00	0.10
Hbase	0.61	1.00	0.10
Struts2	1.00	1.00	0.26
Average	**0.82**	**0.998**	**0.17**

3) RQ3: EFFECTIVENESS OF SOURCE CODE REPOSITORY IN API RECOMMENDATION

Table 5 shows the results of α, β and γ, which indicate the weights for recommendation from API usage location, recommendation from similar historical requests and recommendation from mining API libraries, respectively. In our approach, API method recommendation based on usage location mainly depends on computing the relevance of the source code files to the incoming feature request, and we use α to show the role of mining source code repository for API recommendation. From the results in Table 5, we notice that recommendation from API usage location (α) is important for recommending API methods, and the average value of α achieves 0.82, which is about five times over recommendation from mining API libraries. That is to say, API usage location is effective for improving the accuracy of API method recommendation. Hence, we can conclude that mining source code repository is effective for API recommendation.

In addition, we see from the results in Table 5 that the effectiveness of API method recommendation based on mining API libraries is the most weak. We attempted to investigate into the experimental data in our study, and found that the main reason is that most of descriptions of these API methods in libraries are too short, which affect their effect in identifying relevant APIs. So using natural language processing technique on the short text is not effective as on those data of other software repositories with more textual descriptions. In contrast, there are more rich information in source code repository. So mining source code repository is more helpful for API recommendation.

F. THREATS TO VALIDITY

First, we discuss threats to external validity used to indicate the generalizability of our findings. In our study, we only used 5000 feature requests from five software systems and recommended API methods from 10 third-party libraries. But during the process of practical feature request implementation, there are many existing APIs for usage. This limitation can be reduced by conducting studies in more subject programs (or even practical evolved projects) from more libraries.

Then, we discuss threats to internal validity indicating experimenter bias and errors in our study. We used a greedy algorithm to obtain weights for recommendations in our study. However, the greedy algorithm may not always find the best weight values. In addition, we only selected historical feature requests from some periods for evaluation or training, which may also affect the evaluation results of our approach. To alleviate this limitation, we have carefully checked our source code and performed testing to guarantee that our techniques was correctly implemented.

Finally, we discuss threats to construct validity, used to indicate the suitability of metrics in our study. We employed Hit@N to evaluate the API method recommendation, which is widely used in existing studies [3], [8], [22]–[24]. In addition, we employed MAP, MRR and Hit@N to evaluate our approach in recommending API usage locations, which are also well-known information retrieval metrics and widely used in existing studies [21], [25]–[27]. However, the evaluation results may be different with other metrics.

V. RELATED WORK

A. API RECOMMENDATION

API recommendation is widely studied in the field of code recommendation and reuse. In our previous work, we proposed an approach called *MULAPI* [10]. *MULAPI* improves API method recommendation based on the API usage location. However, there are many parameters in *MULAPI*, which is not well scaled for practice. In this work, we simplify *MULAPI* to reduce its complexity, and the number of parameters is greatly reduced. In addition, with respect to the information retrieval technology for computing similarity, we employ *CNN_NPL* technology. Zheng *et al.* proposed an approach to integrate relevant Web search results for a given API to recommend potential APIs [28]. Thung *et al.* developed an approach that uses the APIs within a project to recommend additional relevant APIs for developers to use. Later, they proposed a new technique, WebAPIRec. WebAPIRec uses a personalized ranking model to recommend web APIs for potential usage [22]. Nguyen *et al.* proposed an API recommendation approach that provides relevant API recommendation based on statistical learning from fine-grained code changes and the context of change modifications [29]. Rahman *et al.* developed a technique to recommend relevant APIs by analyzing keyword-API associations in the crowdsourced knowledge of StackOverflow [23]. Chan *et al.* [4] proposed a technique, which uses the code search to get a graph of API methods. First, they search for similar *closed* or *resolved* feature requests in the software repositories, and identify the modified API methods in these feature requests. Then, they compute the similarity between the description of feature request and the description of API methods to recommend potential API methods. The above work mainly focuses on API recommendation. In this paper, we further recommend API usage locations to guide developers to implement the new proposed feature request.

There are some other work on finer-level code recommendation. Robillard proposed a technique, Suade, which explores the topology of structural dependencies in the program to recommend potential methods or program elements [5]. Long *et al.* [7] proposed an approach which recommends methods to a target method by analyzing variables. These work mainly focuses on analyzing the structural

dependency in the program, and recommends the potential program elements for usage during their development and maintenance activity. In this paper, we recommend the probable API methods and their usage locations for developers to refer for implementing a feature request during the software maintenance activity.

B. FEATURE LOCATION

In our work, feature location is employed to recommend API usage locations. The goal of feature location is to get the source code corresponding to a description of a feature request [11].

Poshyvanyk *et al.* proposed an approach, which uses Latent Semantic Indexing to obtain the feature related program code, and then uses Formal Concept Analysis to cluster the results as a concept lattice [30]. Wang *et al.* proposed to recommend potentially relevant program code in an interactive feature location process [31]. They considered ongoing user context in an interactive manner and performed example-based reasoning to determine relevance of program elements. Wang *et al.* performed a study to compare the effectiveness of ten information retrieval based feature location techniques [32]. The results showed that traditional information retrieval techniques such as vector space model can work better than more recent and complicated information retrieval models, such as the Latent Dirichlet Allocation model. Gethers *et al.* proposed an approach, which combines different techniques, such as information retrieval, dynamic analysis, and software repository mining together to perform feature location [33]. Daiki *et al.* combined both static and dynamic constraints in object-oriented programs, and performed an interactive feature location process [34].

In this paper, we also combined information retrieval and software repository mining technique to recommend relevant files as API usage locations corresponding to a feature request. Then, we integrated the API usage locations to recommend API methods.

VI. CONCLUSIONS

To accelerate the process of feature request implementation during software maintenance and evolution, developers usually utilize third-party APIs or libraries. However, it is tedious and difficult to find suitable API methods and use them in the original software. To address this challenge, we use the feature location technique to recommend API usage locations and API methods for developers to refer based on our previous work [10]. We simplified the *MULAPI* by using fewer parameters to improve its generalizability. The empirical study showed that our approach can still improve the Hit@5 and Hit @10 results by 29.7% and 9.6% respectively in API method recommendation.

For future work, we plan to use the query reformulation technique to pre-process the input of feature request to further improve the accuracy of our approach [35], [36]. We also plan to study personalized API usage location technique based on developers' experiences and habits [37], [38].

REFERENCES

[1] W. Shi, X. Sun, B. Li, Y. Duan, and X. Liu, "Using feature-interface graph for automatic interface recommendation: A case study," in *Proc. 3rd Int. Conf. Adv. Cloud Big Data (CBD)*, Yangzhou, China, Oct./Nov. 2015, pp. 296–303. doi: 10.1109/CBD.2015.55.
[2] X. Sun, B. Li, Y. Duan, W. Shi, and X. Liu, "Mining software repositories for automatic interface recommendation," *Sci. Program.*, vol. 2016, May 2016, Art. no. 5475964. doi: 10.1155/2016/5475964.
[3] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2013, pp. 290–300.
[4] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, p. 10.
[5] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *Proc. 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, vol. 30, no. 5, pp. 11–20.
[6] X. Sun, W. Xu, X. Xia, X. Chen, and B. Li, "Personalized project recommendation on GitHub," *Sci. China Inf. Sci.*, vol. 61, no. 5, 2018, Art. no. 050106. doi: 10.1007/s11432-017-9419-x.
[7] F. Long, X. Wang, and Y. Cai, "Api hyperlinking via structural overlap," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 203–212.
[8] H. Yu, W. Song, and T. Mine, "APIBook: An effective approach for finding APIs," in *Proc. 8th Asia-Pacific Symp. Internetware*, 2016, pp. 45–53.
[9] C. Xu, B. Min, X. Sun, J. Hu, B. Li, and Y. Duan, "MULAPI: A tool for API method and usage location recommendation," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2019.
[10] C. Xu, X. Sun, B. Li, X. Li, and H. Guo, "MULAPI: Improving API method recommendation with API usage location," *J. Syst. Softw.*, vol. 142, pp. 195–205, Aug. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218300840
[11] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Softw., Evol. Process*, vol. 25, no. 1, pp. 53–95, 2013.
[12] X. Sun, B. Li, H. Leung, B. Li, and P. Li, "MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks," *Inf. Softw. Technol.*, vol. 66, pp. 1–12, Oct. 2015. doi: 10.1016/j.infsof.2015.05.003.
[13] M. Hashimoto and A. Mori, "Diff/ts: A tool for fine-grained structural change analysis," in *Proc. 15th Work. Conf. Reverse Eng.*, 2008, pp. 279–288.
[14] M. Hashimoto, A. Mori, and T. Izumida, "A comprehensive and scalable method for analyzing fine-grained source code change patterns," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 351–360.
[15] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the nlp techniques for source code data preprocessing," in *Proc. 3rd Int. Workshop Evidential Assessment Softw. Technol.*, 2014, pp. 32–39.
[16] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *Proc. 20th Work. Conf. Reverse Eng. (WCRE)*, Oct. 2013, pp. 182–191.
[17] G. Casella and E. I. George, "Explaining the Gibbs sampler," *Amer. Statist.*, vol. 46, no. 3, pp. 167–174, 1992.
[18] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2013, pp. 345–355.
[19] B. Sisman and A. C. Kak, "Incorporating version histories in information retrieval based bug localization," in *Proc. 9th IEEE Work. Conf. Mining Softw. Repositories*, Jun. 2012, pp. 50–59.
[20] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng.*, Jun. 2012, pp. 14–24.
[21] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 53–63.
[22] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "WebAPIRec: Recommending web APIs to software projects via personalized ranking," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 1, no. 3, pp. 145–156, Jun. 2017.
[23] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic API recommendation using crowdsourced knowledge," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, 2016, pp. 349–359.
[24] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 631–642.

[25] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2012, pp. 70–79.
[26] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 499–510.
[27] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2011, pp. 253–262.
[28] W. Zheng, Q. Zhang, and M. R. Lyu, "Cross-library api recommendation using web search engines," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur.conference Found. Softw. Eng.*, 2011, pp. 480–483.
[29] A. T. Nguyen *et al.*, "API code recommendation using statistical learning from fine-grained changes," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 511–522.
[30] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, p. 23, 2012.
[31] J. Wang, X. Peng, Z. Xing, K. Fu, and W. Zhao, "Contextual recommendation of relevant program elements in an interactive feature location process," in *Proc. IEEE 17th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2017, pp. 61–70.
[32] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *Proc. 18th Work. Conf. Reverse Eng.*, Oct. 2011, pp. 92–96.
[33] M. Gethers, B. Dit, H. H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proc. 34th Int. Conf. Softw. Eng.*, Jun. 2012, pp. 430–440.
[34] D. Fujioka and N. Nitta, "Constraints based approach to interactive feature location," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 499–503.
[35] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Ree. (SANER)*, Montreal, QC, Canada, Mar. 2015, pp. 545–549. doi: 10.1109/SANER.2015.7081874.
[36] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, and C. Zhou, "Interactive query reformulation for source-code search with word relations," *IEEE* Access, vol. 6, pp. 75660–75668, 2018. doi: 10.1109/ACCESS.2018.2883963.
[37] X. Sun, H. Yang, H. Leung, B. Li, H. J. Li, and L. Liao, "Effectiveness of exploring historical commits for developer recommendation: An empirical study," *Frontiers Comput. Sci.*, vol. 12, no. 3, pp. 528–544, 2018. doi: 10.1007/s11704-016-6023-3.
[38] X. Sun, H. Yang, X. Xia, and B. Li, "Enhancing developer recommendation with supplementary information via mining historical commits," *J. Syst. Softw.*, vol. 134, pp. 355–368, Dec. 2017. doi: 10.1016/j.jss.2017.09.021.

XIAOBING SUN (M'15) received the B.S. degree in computer science and technology from the Jiangsu University of Science and Technology, Jiangsu, China, in 2007, and the Ph.D. degree in computer software and theory from Southeast University, Jiangsu, in 2012. He is currently an Associate Professor with Yangzhou University, Jiangsu. His research interest includes software maintenance and evolution.

CONGYING XU is currently pursuing the degree with the School of Information Engineering, Yangzhou University. His current research interest includes recommendation systems.

BIN LI received the B.S. degree in computer science and technology from Fudan University, Shanghai, China, in 1986, and the Ph.D. degree in computer software and engineering from the Nanjing University of Aeronautics and Astronautics, Jiangsu, China, in 2012. He is currently a Professor with Yangzhou University, Jiangsu. His research interest includes software engineering.

YUCONG DUAN is currently a Professor with the College of Information Science and Technology, Hainan University. His current research interests include empirical software engineering, model-driven development, knowledge engineering, data engineering, and cloud computing.

XINTONG LU is currently pursuing the degree with the School of Information Engineering, Yangzhou University. Her current research interests include recommendation systems and knowledge graph.