Automatic Recommendation of API Methods from Feature Requests

Ferdian Thung[1], Shaowei Wang[1], David Lo[1], and Julia Lawall[2]
[1]Singapore Management University, Singapore
[2]Inria/Lip6 Regal, France
{ferdiant.2013,shaoweiwang.2010,davidlo}@smu.edu.sg, julia.lawall@lip6.fr

***Abstract*—Developers often receive many feature requests. To implement these features, developers can leverage various methods from third party libraries. In this work, we propose an automated approach that takes as input a textual description of a feature request. It then recommends methods in library APIs that developers can use to implement the feature. Our recommendation approach learns from records of other changes made to software systems, and compares the textual description of the requested feature with the textual descriptions of various API methods. We have evaluated our approach on more than 500 feature requests of Axis2/Java, CXF, Hadoop Common, HBase, and Struts 2. Our experiments show that our approach is able to recommend the right methods from 10 libraries with an average recall-rate@5 of 0.690 and recall-rate@10 of 0.779 respectively. We also show that the state-of-the-art approach by Chan et al., that recommends API methods based on *precise* text phrases, is unable to handle feature requests.**

I. Introduction

Developers often receive requests for new features submitted via systems such as JIRA[1]. Given the requirements expressed in these feature requests, developers need to locate code units that should be changed and then implement the required changes. While a number of concern localization techniques have been proposed for locating code units of interest [8], [23], [24], [32], [36], [37], there is still little automated support to help developers implement the changes required to satisfy a feature request.

Many software systems rely on a variety of external libraries for various functionalities. Accordingly, developers often use external libraries to implement required changes. However, using these libraries effectively, requires knowledge of the relevant methods and classes that they provide. Given the large number of libraries, and the large number of methods and classes that they provide, it can be a challenge for developers to identify the methods and classes of interest, given a target requirement document expressed as a feature request.

Considering the above issues and opportunities, there is a need for an automated approach that could help developers to better harness the power of libraries. The automated approach should be able to recommend library methods given a feature request. We refer to our problem as *method recommendation from feature requests*.

A number of techniques have been proposed to recommend code units given a requirement. Mandelin et al. [20] and Thummalapenta and Xie [31] propose a technique to generate code snippets that can convert an object of a particular type to another object of a different type. While this technique is useful for a number of situations, it requires the information about the desired functionality to be expressed at code level. Chan et al. propose a code search technique that takes in text phrases and returns a graph of API methods that best match the phrases [5]. Their approach requires *precise* text phrases that match some words in the API methods. These techniques are not sufficient to automatically process feature requests, which typically describe high-level requirements, written in natural language. In this work, we propose a complementary approach that recommends relevant library methods directly from feature requests.

Our proposed approach learns from a training dataset of changes made to a software system recorded in repositories (i.e., issue management systems, and version control systems). Each change in the dataset has three parts: the textual description describing the change (text), the code before the change (pre-change), and the code after the change (post-change). Our approach takes as input a new textual description (text) and then recommends methods from a set of libraries to be used in the post-change code.

To recover methods that can be used to construct the post-change code, our approach performs a two-pronged approach to rank relevant methods. First, it searches for similar *closed* or *resolved* feature requests in the training data. A *closed* or *resolved* feature request is one that has been addressed by developers and where appropriate changes have been made to the software system. It then looks into the API methods that are used to implement these feature requests and measures the relevance of various methods based on the number of similar closed requests which use them. Second, our approach measures relevance by looking into the similarity between the textual description of the feature request and the descriptions of the API methods. Our approach then learns an integrated ranking function that is used to recover a list of potentially relevant library methods that are then recommended to the developers.

We have evaluated our solution on feature requests stored in the JIRA issue management systems of 5 Java applications: Axis2/Java, CXF, Hadoop Common, HBase, and Struts 2. Each feature request in JIRA can be linked to the commits in the corresponding version control system that implement

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

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

the requested feature. We recommend methods from 10 third party libraries: commons-codec, commons-io, commons-lang, commons-logging, junit, servlet-api, easymock, log4j, slf4j-api, and slf4j-log4j12. These are the most popular libraries used by Java applications developed under the Apache Foundation. These libraries provide various functionalities including testing, logging, I/O, etc. The accuracy of our proposed approach is measured using recall-rate@5 and recall-rate@10; these measures have also been used to evaluate past studies on bug report analysis [11], [22], [30], [34]. Our experiments show that we can achieve a recall-rate@5 and recall-rate@10 of 0.690 and 0.779 respectively. On the other hand, we show that the state-of-the-art code search approach by Chan et al. [5] that recommends API methods from precise textual phrases is not effective to directly process feature requests, which often contain high level requirements. Indeed, we find that their approach returns no relevant methods.

Our contributions are as follows:

1) We propose a new problem of *method recommendation from feature requests*.
2) We propose a technique that leverages information from past similar closed or resolved feature requests and compares the textual description of a feature request with those of library methods. Our technique learns an integrated ranking function that is then used to recommend library methods to be used in the post-change code.
3) We evaluate our approach on change requests of 5 applications and recommend methods from 10 libraries. We show that our approach achieves a recall-rate@5 and recall-rate@10 of 0.690 and 0.779, respectively.

The structure of this paper is as follows. In Section II, we describe some preliminary concepts. In Section III, we present an overview of our proposed approach. We elaborate the three processing components of our approach in Sections IV, V, & VI. We highlight our experimental methodology and results in Section VII and describe related studies in Section VIII. Finally, we conclude and mention future work in Section IX.

II. Preliminaries

In this section, we describe some preliminary materials that are needed for latter sections. We first describe the issue management system JIRA and show how it stores feature requests. We then describe some text pre-processing techniques and the vector space model.

A. Feature Requests and JIRA

JIRA is an issue management system developed by Altassian.[2] It is used in many software projects to capture and store issues that are reported by users and developers. Among its users are the many projects developed by the Apache Software Foundation. Figure 1 shows a sample issue stored in the JIRA repository of an Apache project. An issue contains a number of fields including: summary, description, type, priority, component, etc. For our work, we are especially interested in the fields listed in Table I.

TABLE I
Fields in a JIRA Issue

Name	Description
Summary	the summary/title of the issue
Description	the detailed description of the issue
Component	the component affected by the issue
Reporter	the name of the person who submitted the issue report
Priority	the urgency of the issue to be addressed

Each issue in JIRA can be categorized into one of these types: “Bug”, “New Feature”, “Task”, etc. In this study, we are interested in feature requests reported in JIRA. A feature request in JIRA can be seen as an issue of type: “New Feature”, “Improvement”, or “Wish”. Each issue also has a priority. The priority indicates the urgency of the issue to be addressed. Table II lists the priorities available in JIRA along with their descriptions.

TABLE II
Priority in JIRA

Name	Description
Blocker	Blocks development and/or testing, production could not run
Critical	Crash, loss of data, severe memory leak
Major	Major loss of functionality
Minor	Minor loss of functionality, or other problem where an easy workaround is present
Trivial	Cosmetic problem like misspelled words or misaligned text

An issue can be assigned various status labels: “open”, “in progress”, “resolved”, “closed”, etc. A new issue is typically given the status “open”. An issue that has been addressed to completion by developers is given the status: “resolved” or “closed”. Each issue report in JIRA has a unique issue identifier (id) used to identify the report. The format of this identifier is typically a short name for the project followed by the issue number in the project (e.g., HBASE-3850). JIRA can be integrated with version control systems like svn, git, etc. Each issue in JIRA can then be linked to the commits in the version control system that address the issue. The issue identifier is added to the log messages of the commits that address the issue. This provides an easy identification of changes made to address the issue. We show an example of this link in Figure 2. We can see that it is easy to identify the commits in the version control system that address the *HBASE-3850* issue.

B. Text Pre-processing

Text pre-processing is an important task in text mining [21]. Its purpose is to convert a piece of text into a common representation easily processed by a text mining algorithm and to remove certain noise. Widely used text pre-processing strategies include tokenization and stemming.

[2]http://www.atlassian.com/software/jira/overview

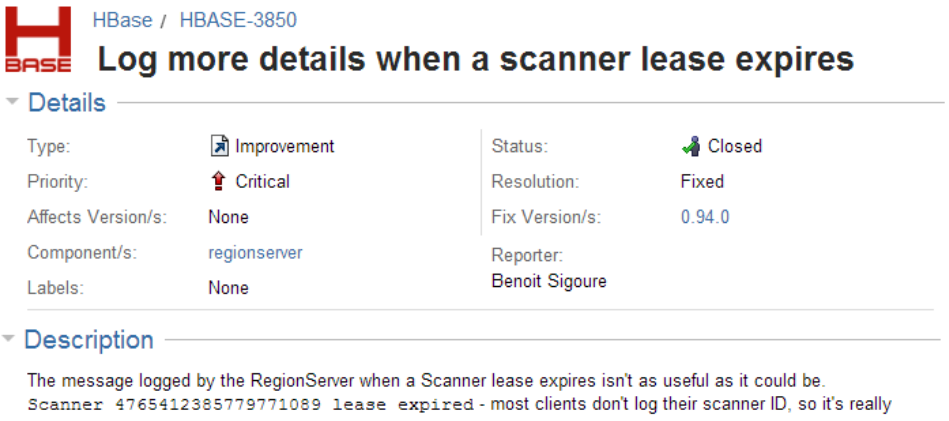

Fig. 1. A Sample JIRA Issue

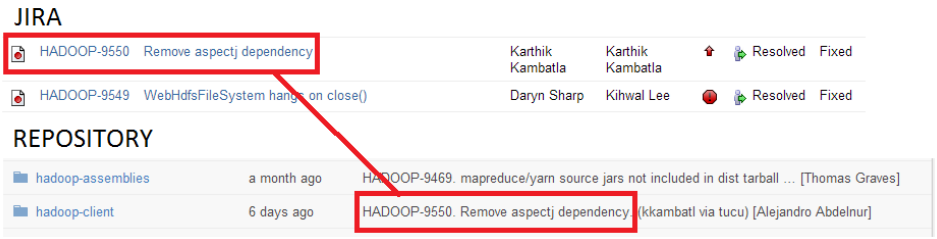

Fig. 2. Sample Link Between A JIRA Issue And A Commit in A Version Control System

Tokenization refers to the process that breaks a document into word tokens. Delimiters are used to demarcate one token from another. Typically, space and punctuation are used as delimiters. After tokenization, a document is converted to a bag (i.e., a multiset) of word tokens. This is often referred to as the *bag of words* representation.

Stemming is the process of converting a word to its base form. This base form is usually called the stem word. For example, word "argue", "argues", "argued", and "arguing" have a common stem word "argu". Even though word "argu" is not a dictionary word, the conversion assures that we can identify a word in its different forms and link these word forms together. Without stemming, the multiple forms would be treated as different words altogether, which is not desirable in many cases. In our work, we use the Porter stemmer[3] to stem the words. It employs several rule based heuristics to convert a word to its stem word by stripping a suffix of the word. The Porter stemmer has been used in many past software engineering studies, e.g., [9], [10].

C. Vector Space Model

After the text pre-processing step, the document is now represented as a bag of words. The vector space model represents a bag of words as a vector of weights. Each word in the bag becomes an element in the vector. The weight of each word indicates its importance. Term frequency and inverse document frequency are often used to compute the weight of a word and thus quantify its importance in a document.

[3]http://tartarus.org/martin/PorterStemmer/

Term frequency (TF) refers to the number of times a term (i.e., a word, or a token) appears in a document. The more times a term appears in a document, the more important that term is considered to be. Inverse document frequency (IDF) is the reciprocal of the document frequency (DF). The document frequency of a term is the number of documents in the corpus (i.e., a set of documents or a document collection under consideration, e.g., all feature requests, all method descriptions in the API documentation) that contain the term. The higher the inverse document frequency is, the more important is the term, as it can better differentiate one document from another. TF-IDF is often used to compute the weight of a term i in a document D considering a corpus C in the following way:

$$\begin{aligned} w_{i,D,C} &= TF_{i,D} \times IDF_{i,C} \\ IDF_{i,C} &= \frac{1}{DF_{i,C}} \end{aligned} \quad (1)$$

$TF_{i,D}$ refers to the number of times a word i appears in a document D. $DF_{i,C}$ refers to the number of documents in C that contains the word i.

From the above, given a bag of words representing a document D in a corpus C, we can convert it to its corresponding term vector by computing the weight of each word in C and putting them into a vector. The weight of a word in C, but not in D, would be 0. We denote the term vector representation of document D considering a corpus C as $VSM_C(D)$. Implementation-wise, a sparse vector representation is typically used (i.e., only the non-zero entries of the vector are stored).

Given two documents, we can compute the similarity between them by comparing their representative vectors. Cosine similarity is often used to measure the similarity between two vectors. Let V_1 and V_2 denotes two vectors of weights of size N, then the cosine similarity of these two vectors is given by the following equation:

$$Cosine(V_1, V_2) = \frac{\sum_{i=1}^{N} w_{i,V_1} \times w_{i,V_2}}{\sqrt{\sum_{i=1}^{N} w_{i,V_1}^2} \sqrt{\sum_{i=1}^{N} w_{i,V_2}^2}} \quad (2)$$

$w_{i,V}$ refers to the i^{th} weight in vector V.

III. Overall Framework

The overall framework of our approach is shown in Figure 3. Our framework consists of three important components: *History Based Recommender*, *Description Based Recommender*, and *Integrator*.

History Based Recommender takes as input the description of the new feature request (*Textual Description*) and a historical database containing old "closed" or "resolved" feature requests (*Historical Feature Request Database (HDB)*). The recommender compares the new feature request with those in the historical database and finds the closest ones. It then recommends relevant methods based on the methods that were used to implement those closest feature requests.

Description Based Recommender takes as input the description of the new feature request (*Textual Description*) and the documentation of API libraries (*API Documentations (ADoc)*). The recommender computes the similarity of the textual description of the new feature request with the description of each method in the API documentations of the libraries. It recommends methods whose textual descriptions have the highest similarity with the textual description of the new feature request.

Integrator combines the recommendations from *History Based Recommender* and *Description Based Recommender*. It takes as inputs recommendation scores from the two components and outputs a final list of methods to be recommended to the user.

IV. History-Based Recommendation

In the history-based recommender component, we first find the nearest neighbors of a new feature request from the historical database of "closed" or "resolved" feature requests that we have. We compare two feature requests based on the contents of their summary, description, component, reporter, and priority fields (see Table I). We compute a similarity score for each field and combine the scores into an aggregate score that specifies the similarity between two feature requests. We define the similarity score for each field as follows.

1) ***Summary* and *Description***. The contents of these fields are free-form texts. We pick only alphanumeric terms from these texts. We employ standard text preprocessing (tokenization and stemming) and convert the terms into a bag of word and its corresponding term vector (see Section II). We have 3 options: we can take all terms in the summary field, we can take all terms in the description field, and we can take all terms in both summary and description fields. We compute the following 3 similarity scores between the new feature request F_1 and a historical feature request F_2 in terms of their summary/description fields using cosine similarity:

$$\begin{aligned} &SimSum(F_1, F_2) = \\ &\quad Cosine(VSM_{HDB}(F_1^S), VSM_{HDB}(F_2^S)) \\ &SimDesc(F_1, F_2) = \\ &\quad Cosine(VSM_{HDB}(F_1^D), VSM_{HDB}(F_2^D)) \\ &SimSumDesc(F_1, F_2) = \\ &\quad Cosine(VSM_{HDB}(F_1^{SD}), VSM_{HDB}(F_2^{SD})) \end{aligned} \quad (3)$$

   In the above equations, F_1^S denotes the content of the summary field of F_1. F_1^D denotes the content of the description field of F_1. F_1^{SD} denotes the contents of the summary and description fields of F_1. *HDB* is the Historical Feature Request Database (see Figure 3).
2) ***Component***. A feature request, if implemented, can affect multiple components in the system. Thus, the content of the component field of a feature request is a set of values. We compute the similarity score $SimComp$ between a new feature request $F1$ and a historical feature request $F2$ in terms of their components as follows:

$$SimComp(F_1, F_2) = \frac{|Nc_{F_1} \cap Nc_{F_2}|}{\sqrt{|Nc_{F_1}|} * \sqrt{|Nc_{F_2}|}}$$

   Nc_F denotes the set of components of feature request F.
3) ***Reporter***. The similarity score $SimReport$ between a new feature request F_1 and a historical feature request F_2 in terms of their reporters is 1 if both of them have the same reporter and 0 otherwise.
4) ***Priority***. Each priority in JIRA can be assigned an ordinal value to quantify its level of urgency. We assign value 1 for "Blocker", 2 for "Critical", 3 for "Major", 4 for "Minor", and 5 for "Trivial". A lower value means a higher priority or level of urgency. We compute the similarity score $SimPrio$ between a new feature request F_1 and a historical feature request F_2 in terms of their priority based on these values. The formula is as follows:

$$SimPrio(F_1, F_2) = \frac{1}{1 + |Prio_{F_1} - Prio_{F_2}|}$$

   $Prio_F$ denotes the ordinal value corresponding to the priority of feature request F.

Example. Consider the example feature request shown in Figure 1 as a historical feature request and a new feature request having values as shown in Table III. We can compute the similarity between these two feature requests for each field as follows.

1) ***Summary* and *Description***. Since the computation steps for both summary and description are basically the same, in this example, we only compute the similarity score for the summary. We convert a summary to a vector of $TF - IDF$ weights of its stemmed alphanumerical words. Each word has a term frequency TF equal to 1.

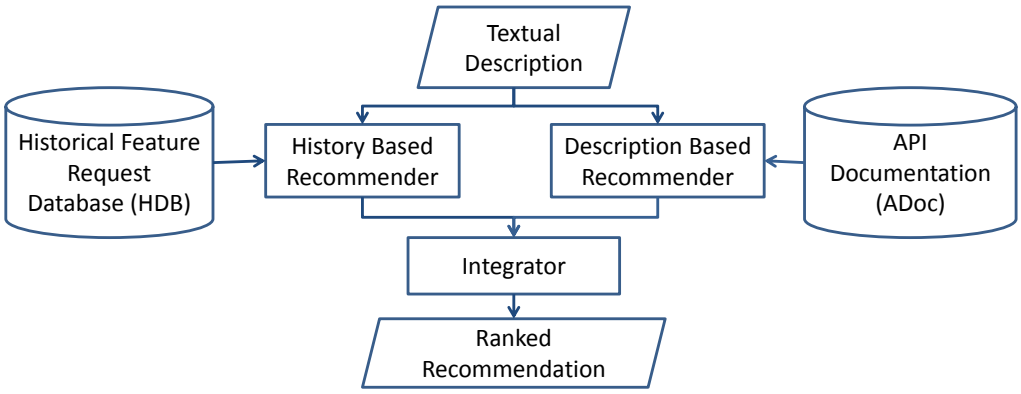

Fig. 3. Our Recommendation Framework

Assuming that the IDF of each word is 1, which means the word only appears in one document in the historical database of feature requests (HDB), the similarity score for the summaries of the historical and new feature request (i.e., $SimSum$) is $1/(\sqrt{8} * \sqrt{6}) = 0.144$.

2) ***Component***. The historical feature request and the new feature request do not share any component. Thus, the $SimComp$ score is 0.
3) ***Reporter***. The feature requests are reported by different reporters so the $SimReport$ score is 0.
4) ***Priority***. The historical feature request has "Critical" priority which corresponds to the ordinal value 2, while the new feature request has "Minor" priority which corresponds to the ordinal value 4. The denominator of $SimPrio$ is equal to $1 + |2 - 4| = 3$. Thus, the $SimPrio$ score is $1/3 = 0.333$.

TABLE III
EXAMPLE OF A NEW FEATURE REQUEST

Field	Values
ID	HBASE-6372
Summary	Add scanner batching to Export job
Description	When a single row is too large for the RS heap then an OOME can take out the entire RS. Setting scanner batching in custom scans helps avoiding this scenario, but for the supplied Export job this is not set. Similar to HBASE-3421 we can set the batching to a low number - or if needed make it a command line option.
Components	mapreduce
Reporter	Lars George
Priority	Minor

Finally, to compute the final similarity score between two feature requests, we aggregate the similarity scores of their constituent fields. We compute the final similarity $Sim^{HISTORY}$ between a new feature request F_1 and a historical feature request F_2 in the historical database using the following formula:

$$Sim^{HISTORY}(F_1, F_2) = \alpha_1 * SimSum(F_1, F_2) + \alpha_2 * SimDesc(F_1, F_2) + \alpha_3 * SimSumDesc(F_1, F_2) + \alpha_4 * SimReport(F_1, F_2) + \alpha_5 * SimComp(F_1, F_2) + \alpha_6 * SimPrio(F_1, F_2) \quad (4)$$

$SimSum(F_1, F_2)$, $SimDesc(F_1, F_2)$, $SimSumDesc(F_1, F_2)$, $SimReport(F_1, F_2)$, $SimComp(F_1, F_2)$, and $SimPrio(F_1, F_2)$ denote the similarity scores between F_1's and F_2's summary, description, combination of summary and description, reporter, components, and priority respectively. α_1-α_6 denotes the weights of each field contributing to the $Sim^{HISTORY}$ score.

Given a new feature request, we rank the historical feature requests in the *Historical Feature Request Database* based on their $Sim^{HISTORY}$ scores when compared to the new feature request. The higher the score is, the more similar a historical feature request is to the new feature request. We then pick the top-k feature requests with the highest $Sim^{HISTORY}$ scores. If there are feature requests with rank greater than k that have the same score as the k-th feature request, we group the feature requests having this score. We then randomly select feature requests from this group until we have k nearest neighbors (i.e., ties are randomly broken).

Next, we compute the recommendation scores for each method based on these top-k nearest neighbors. We collect the methods that are used to implement the feature requests in the top-k nearest neighbors and compute a score for each method. Given a method m, the *history based recommendation score* of API method m for feature request F, denoted as $RecScore^{HISTORY}(F, m)$ is computed as follows:

$$RecScore^{HISTORY}(F, m) = \frac{NNCount_{Method}(F, m)}{k} \quad (5)$$

$NNCount_{Method}(m)$ denotes the number of nearest neighbors of feature request F that use API method m, and k denotes the total number of nearest neighbors. By default, we set the number of nearest neighbors k to be 5. The API method with the highest $RecScore^{HISTORY}$ score is the most suitable API method based on our history-based recommender.

Example. Consider a top-2 nearest neighbor list containing N_1 and N_2. Feature request N_1 was implemented using method m_1 and m_2 while feature request N_2 was implemented using method m_2. Thus, the value of $NNCount_{Method}$ is 1 for m_1 and 2 for m_2. We can then compute $RecScore^{HISTORY}$ score of m_1 and m_2 which are 0.5 and 1.0 respectively.

V. Description-Based Recommendation

When adding a new feature to an application, developers often look at the API documentation to see which methods they can use to help them implement the feature. The API documentation contains textual descriptions that explain each method in the library. By looking at the documentation, developers can find out which API methods to use for implementing a feature. Our description-based recommender component mimics this process to find relevant methods given the textual description of a feature request. Given a new feature request F, it proceeds in the following steps:

1) **Feature Request Preprocessing.** We extract the contents of the summary and description fields of the input feature request F. We again perform standard text preprocessing steps to convert them into a bag of words. This bag of word is then converted into its corresponding term vector representation $VSM_{ADoc}(F)$ where each token (i.e., term) in the bag is represented by its TF-IDF weight[4] and $ADoc$ is API Documentation (see Figure 3).
2) **API Method Preprocessing.** For each API method m that we consider, we extract its method signature and its corresponding description in the API documentation. We extract the method descriptions from the Javadoc comments in the code base of the APIs. We make use of Eclipse Java Development Tools (JDT) to extract these Javadoc comments. Javadoc has tags which serve as metadata. Examples include @*param* indicating the start of the description of a method parameter, @*return* indicating the start of the description of the return value of a method, etc. Since these tags only serve as metadata and are not specific to the API, we remove them from the extracted Javadoc comments. Additionally, developers sometimes add HTML tags in the documentation to improve its readability when it is viewed in e.g., a web browser. Since these tags are only meant to improve the look and feel of the API documentation and are again not specific to the API, we also remove all HTML tags. Next, we perform standard text preprocessing (i.e., tokenization and stemming) to convert the cleaned method descriptions in the Javadoc comments into bags of words. We then convert each bag of words into its corresponding term vector representation, $VSM_{ADoc}(m)$.
3) **Similarity Computation.** Next, for each method m, we compute the similarity between $VSM_{ADoc}(F)$ and $VSM_{ADoc}(m)$. We use the cosine similarity to compute the similarity between these two vectors (see Section II). We refer to this similarity score as the *description based recommendation score* between feature request F and API method m and denote it as $RecScore^{DESCRIPTION}(F, m)$:

$$RecScore^{DESCRIPTION}(F, m) = Cosine(VSM_{ADoc}(F), VSM_{ADoc}(m)) \quad (6)$$

[4]The description of text preprocessing and vector space model is given in Section II.

After the above steps, we have the $RecScore^{DESCRIPTION}$ scores of various API methods. The method with the highest score is the most relevant API method based on the description-based recommender.

VI. Unifying History- and Description-Based Recommendation

The last component in our framework is the Integrator, which combines the scores from the previous components. We compute the final recommendation score $RecScore$ between feature request F and API method m by combining $RecScore^{HISTORY}(F, m)$ and $RecScore^{DESCRIPTION}(F, m)$, as follows:

$$RecScore(F, m) = \alpha * RecScore^{HISTORY}(F, m) + \beta * RecScore^{DESCRIPTION}(F, m) \quad (7)$$

α and β are the weights for $RecScore^{HISTORY}$ and $RecScore^{DESCRIPTION}$, respectively.

To set the appropriate values for α and β of $RecScore$ (see Equation 7) and the appropriate values for α_1-α_6 of $RecScore^{HISTORY}$ (see Equation 5), we heuristically find the best set of weights that maximizes an evaluation metric based on a training dataset We employ a greedy approach based on Gibbs sampling [4] that iteratively refines the set of weights. At each iteration, each weight is optimized independently. Several iterations are performed to further optimize the weights. The pseudocode of our approach to tune the set of weights is shown in Figure 4.

Our algorithm takes the input historical feature requests, a set of API documentations, and the number of iterations to perform Gibbs sampling $numIter$. It then outputs the set of best weights. Initially all weights ($\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha, \beta$) are set to 1.0 (Line 8). We then iterate $numIter$ times (Lines 11-23). For each iteration, we try to estimate the best $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha, \beta$ weights independently (Lines 12-22). We go through each of the eight weights and for each weight we investigate 11 settings (i.e., 0.0, 0.1, 0.2, ..., 1.0) (Lines 13-21). We pick the setting that give the best result (Lines 16-19,21). Method $eval$ evaluates how good a particular weight setting is with respect to an evaluation criteria (Line 16). In this study, we make use of recall-rate@k [22], [26], [29], [30], [34] as the evaluation criteria (see Section VII). At the end of the above process, we would have estimated the best weights.

In the end, we want to get the top-k methods based on $RecScore$. To do this, we first get the set of methods with non-zero $RecScore^{HISTORY}$ scores. For all these methods, we compute their $RecScore$ scores. We then return the top-k methods based on the $RecScore$ scores. If there are methods having the same score as the k-th method, we group the methods having this score and randomly select methods from this group until we have k top methods (i.e., ties are randomly broken).

Input:
  $FReqs$ = list of historical closed or resolved feature requests
  $Docs$ = the documentation of APIs
  $numIter$ = number of iteration
Output:
  Estimated best weights: $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha, \beta\}$
Method:
Let $weights$ = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}
Let $maxEvalScore$ = 0
Let $valForMax$ = 0
for $i = 0$ to $numIter$ **do**
  for $j = 0$ to $weights.Length$ **do**
    for $k = 1.0$ to 0.0 by 0.1 **do**
      $weights[j] = k$
      $evalScore = eval(FReqs, Docs, weights)$
      if $maxEvalScore > evalScore$ **then**
        $maxEvalScore = evalScore$
        $valForMax = k$
      end if
    end for
    $weights[j] = valForMax$
  end for
end for
return $weights$

Fig. 4. Pseudocode for our Weight Tuning Algorithm

VII. Experiments & Analysis

In this section, we first describe our dataset. We then outline our experimental methodology and research questions. Next, we present the answers to the research questions and describe some threats to validity.

A. Dataset

We first describe how we select libraries of interest and the projects that we investigate. Next, we describe the feature requests that we use to evaluate our approach.

1) Library Selection: We pick libraries that are frequently used by many projects of the Apache Foundation. We choose Apache projects that use Maven as their project management tool. Maven includes a dependency management feature which helps us resolve the libraries used by the projects. These libraries have standard names in Maven, so it is easy to reliably match the libraries that are used across different projects. We first download Apache projects that use Java as their main programming language. We then filter these projects based on the existence of the *pom.xml* file in their root directory. This *pom.xml* file indicates the use of Maven as the project management tool. After this filtering process, we have 207 Apache projects. For each project, we extract the libraries it depends on and count the number of projects using each library. We then rank the libraries based on the number of projects using it. We list the top-10 libraries in Table IV. These are the target libraries for our study – we recommend methods from these libraries.

TABLE IV
Top-10 Most Popular Libraries in 207 Apache Projects

Name	Description
commons-codec	common encoder and decoder library for string, URL, etc
commons-io	common library for input/output functionality
commons-lang	common library providing extra methods for manipulating Java core classes
commons-logging	common library which encapsulates the logging process for different logging implementations
easymock	a library that provides easy way to use mock objects in unit testing
junit	unit testing framework
log4j	logging library
servlet-api	library providing contracts between a servlet and the runtime environment
slf4j-api	an abstraction library for various logging framework
slf4j-log4j12	a binding library for slf4j and log4j

2) Project Selection: Next, we want to pick large projects ($>$ 100,000 lines of code) from the 207 Apache Foundation projects whose feature requests we use to evaluate our approach. We omit "toy" and small projects. We filter out projects that only use a few of the 10 selected libraries. We choose these projects as we only recommend methods from the 10 libraries. We also filter out projects that do not use JIRA issue management system. We choose these projects as we need reliable links between bug reports and corresponding commits in the version controls system. These links are well maintained in JIRA issue management systems, c.f., [1]. Table V lists the projects that we use in this study and their descriptions.

TABLE V
Selected Apache Projects

Name	Description
Axis2/Java	server-client web service engine
CXF	open source web service framework
Hadoop Common	common utilities used in other Hadoop modules
Hbase	scalable distributed database based on Big Table [6]
Struts 2	enterprise-ready web application framework

3) Feature Request Selection and Gold Standard Extraction: We pick feature requests from the JIRA issue management system of the selected projects. We pick only issues in JIRA that are of relevant types. As mentioned in Section II-A, there are 3 issue types in JIRA that correspond to a feature request, namely "New Feature", "Improvement", and "Wish". For these three issue types, we pick issues that are either "closed" or "resolved".

JIRA contains explicit links between issue reports and repository commits. Using these links, we find the changed files for each commit that addresses an issue. These files have a pre-change and a post-change version. We extract the methods from the libraries shown in Table IV that are invoked in the post-change version of the file as the evaluation benchmark

or gold standard (c.f. [21]). A good recommendation system should be able to recommend these methods. There are three cases that we need to consider when extracting method calls for gold standard:

1) *File is added in the post-change version.* If the file does not exist in the pre-change version, we take all the methods from the 10 libraries that are invoked in the post-change version as members of the gold standard.
2) *File is changed in the post-change version.* If the file exists in both pre-change and post-change versions, we take as the gold standard all the methods from the 10 libraries that are invoked in the post-change version, but not invoked in the pre-change version.
3) *File is deleted in the post-change version.* If the file is deleted, the file does not contribute any API method to the gold standard.

Since we only recommend methods from the top-10 libraries, we only take feature requests whose gold standard set contains at least one method from the 10 libraries. We ignore very rare methods that are only used in one feature request. Our history-based approach requires a method to be used in at least 2 feature requests. Table VI shows the number of feature requests for each of the projects that we use in this study.

TABLE VI
NUMBER OF FEATURE REQUESTS IN OUR DATASET

Project	#Feature Request
Axis2/Java	108
CXF	106
Hadoop Common	79
Hbase	161
Struts 2	53
Total	**507**

B. Methodology & Research Questions

In order to measure the effectiveness of our approach, we use a commonly used evaluation measure namely *average recall-rate@k* [26]. *Recall-rate@k* has a value of either 1 or 0 where k is the number of the returned items. It has value 1 if at least one of the k returned items (i.e., recommended method) is a member of the gold standard and 0 otherwise. We use *recall-rate@k* as it has also been used in many past studies that also analyze entries in issue management systems [22], [26], [29], [30], [34].

For each project, we perform stratified ten-fold cross validations to evaluate the effectiveness of our approach. We divide the feature requests of a project randomly into ten groups of roughly equal sizes (± 1) and then perform ten iterations. For each iteration, one group is used as the test data (i.e., they form the set of new feature requests) and the remaining nine groups are combined to be the training data (i.e., they form the historical feature request database (HDB)). The test data is used to evaluate the effectiveness of our approach. We report the average effectiveness over the ten iterations.

We consider the following research questions:

RQ1 What is the effectiveness of our proposed approach?
RQ2 What are the relative contributions of the various components of our approach?
RQ3 How effective is our approach compared to a state-of-the-art code search based approach in recommending relevant methods for a feature request?

C. Experimental Results

We describe the answers to each of the research questions below.

1) RQ1: Effectiveness of the Proposed Approach: Table VII shows the effectiveness of our approach. The average recall-rate@5 and recall-rate@10 are 0.690 and 0.779 respectively. We show that, by only returning 5 methods, our approach can correctly recommend relevant methods for 57.1% to 80.5% of the feature requests in a project. In other words, our approach can put relevant methods in high ranking positions. If we increase the recommendation size to be 10 methods, our approach can correctly recommend at least one relevant method for 70.9% to 90.8% of the feature requests.

TABLE VII
EFFECTIVENESS OF OUR APPROACH

Project	Recall-Rate@5	Recall-Rate@10
Axis2/Java	0.805	0.908
CXF	0.628	0.725
Hadoop Common	0.571	0.709
HBase	0.789	0.839
Struts 2	0.657	0.713
Overall	**0.690**	**0.779**

2) RQ2: Relative Contributions: Our proposed approach has two main components: the history-based recommender and the description-based recommender. Our history-based recommender computes the similarity between two feature requests by aggregating 6 similarity scores (based on summary, description, summary + description, reporter, component and priority). In this research question, we want to investigate the relative contributions of the various components and sub-components of our approach.

We employ Gibbs sampling to tune 8 weights to yield a semi optimal setting. Thus, we can estimate the contributions of the various components and sub-components of our approach from their corresponding weights. The average values of the 8 weights across the ten fold cross validation performed for computing recall-rate@5 and recall-rate@10 are shown in Figure 4. We find that all components and sub-components of our approach are important as none of them is given a weight of zero. For both $k = 5$ and $k = 10$, α_1 has the lowest weight compared to the other parameters, suggesting that it is less useful than the other information leveraged by our approach. In our approach, each recommended method needs to have a non-zero $RecScore^{HISTORY}$ score (see Section VI). However, this does not mean that the $RecScore^{DESCRIPTION}$ score is not useful. Indeed, we note that the weight of our description-based recommendation

score (*Description*) is higher than that of our history-based recommendation score (*Historical*). This indicates that among methods with non-zero $RecScore^{HISTORY}$ score, we can use $RecScore^{DESCRIPTION}$ scores to rank them.

3) RQ3: Comparison with a Code Search Based Approach: Code search can also be used to recommend relevant API methods. Chan et al. propose a graph-based approach that can search an API library for relevant methods given a set of text phrases [5]. To the best of our knowledge, this is the closest study to our work. Their approach processes the text queries and returns a connected graph whose nodes are methods. They have evaluated their approach on a set of *precise* text queries that contain keywords that match desired methods. For example, for input queries containing phrases: store, folder, open, and search, they output several relevant methods including: *javax.mail.Store:getDefaultFolder()*, *javax.mail.Folder:open(int mode)*, etc. Note that the input queries contain keywords that must appear in the signatures of the relevant methods. We want to investigate if their approach can also handle feature requests.

To do this, we preprocess a feature request into text phrases. We treat each word that appears in the summary and description fields of a feature request as a text phrase. We then run their tool on our processed data. Table IX show the average number of methods that are returned in the connected graphs outputted by their tool. Even though the tool returns a number of methods, unfortunately none of them are relevant for each of the 507 feature requests (i.e., their recall-rate@5 and recall-rate@10 are both 0). This shows that approaches that process precise text queries cannot handle feature requests. Indeed, feature requests often contain high level requirements while methods contain low level requirements. Our proposed approach tackles this problem by leveraging historical feature requests.

TABLE IX
AVERAGE # OF RETURNED METHODS BY CHAN ET AL.'S APPROACH

Project	Average # of Returned Methods
Axis2/Java	2.2
CXF	1.8
Hadoop Common	1.7
HBase	1.8
Struts 2	1.7
Average	**1.84**

D. Threats to Validity

Threats to internal validity refers to experimental bias and errors. We have checked our code and data for errors. Still there could be errors that we have not noticed. We also ensure that we do not mix training and test data in our cross validation.

Threats to external validity refers to the generalizability of our proposed approach. In this study, to address this threat, we have considered a few hundred feature requests from 5 software systems. We have also recommended methods from 10 libraries. In the future, we plan to reduce this threat further by analyzing more feature requests from additional software systems and recommend methods from more libraries. We have also performed cross validation, which is the standard approach to assess how a proposed approach would perform on an independent dataset.

Threats to construct validity refers to the suitability of our evaluation metrics. We make use of recall-rate@k which is a commonly used metric in many past studies [22], [26], [29], [30], [34]. Thus, we believe there is little threat to construct validity.

VIII. RELATED WORK

Mandelin et al. propose the tool Prospector, which recommends objects and method calls, referred to as jungloids, to convert an object of a particular type, to an object of another type [20]. Prospector takes as input a query consisting of a pair of the input type and the output type. It then analyzes signatures of API methods and constructs a signature graph to recommend jungloids based on the query. A jungloid is ranked based on the number of methods it contains and the output type. Thummalapenta and Xie investigate the same problem [31]. However, the make use of a code search engine to solve the problem. The code search engine is used to collect code examples which are then analyzed to recover the method sequences. While the approach by Mandelin et al. analyzes library code, the approach by Thummalapenta et al. analyzes client code retrieved by code search engines. Our work differs in several respects: we consider different problems (method recommendation given a feature request vs. method recommendation given an input-output type pair), and we leverage different resources (historical feature requests + API documentation vs. library code or client code returned by code search engine).

Bruch et al. propose a code completion system that recommends method calls by looking for code snippets in existing code repositories that share a similar context as the context that a developer is working on [3]. They propose three code completion systems based on frequency of method call usage, an association rule mining algorithm, and a k-nearest neighbor algorithm. The k-nearest neighbor algorithm performs the best. Our work differs in several respects: we consider different problems (method recommendation given a feature request vs. method recommendation given a code context), and we leverage different resources (historical feature requests + API documentation vs. code repositories).

Robillard proposes a technique, Suade, that recommends methods or other program elements of interest to help developers perform a software maintenance task [25]. Suade takes as input a set of program elements, and outputs other program elements that potentially interest developers. It works by investigating the structural dependencies of program elements and considers two criteria: specificity and reinforcement. A program element of interest needs to be specific enough to the input set and its relationship to the input set is reinforced by existing relationships among program elements in the input

TABLE VIII
AVERAGE WEIGHTS FOR K = 5 AND 10

Project	k	Weight							
		α_1	α_2	α_3	α_4	α_5	α_6	α	β
Axis2/Java		0.40	0.43	0.59	0.31	0.44	0.70	0.61	0.70
CXF		0.60	0.61	0.72	0.87	0.74	0.82	0.72	0.81
Hadoop Common	@5	0.70	0.61	0.56	0.35	0.63	0.70	0.51	0.60
HBase		0.38	0.29	0.36	0.20	0.23	0.60	0.52	0.70
Struts 2		0.58	0.10	0.49	0.18	0.05	0.61	0.42	0.71
Average		**0.532**	**0.408**	**0.544**	**0.382**	**0.418**	**0.686**	**0.556**	**0.704**
Axis2/Java		0.54	0.59	0.45	0.19	0.27	0.42	0.71	0.80
CXF		0.50	0.51	0.51	0.63	0.54	0.68	0.77	0.60
Hadoop Common	@10	0.52	0.65	0.58	0.51	0.64	0.54	0.52	0.51
HBase		0.35	0.54	0.36	0.31	0.42	0.15	0.10	0.10
Struts 2		0.77	0.62	0.73	0.36	0.79	0.70	0.73	1.00
Average		**0.536**	**0.582**	**0.526**	**0.400**	**0.532**	**0.498**	**0.566**	**0.602**

set. Saul et al. addresses a similar problem [27]. Their goal is to recommend a set of methods that are related to a target method. To achieve this goal, structural information in a call graph is analyzed. They propose a new algorithm named FRAN which performs random walk on the callgraph. Several other approaches recommend methods related to a target method using static analysis [19], [35]. Long et al. propose Altair that recommends method based on variables that are shared among related methods [19]. Zhang et al. enhance a call graph with control flow information and use it for method recommendation [35]. In this work, we consider a different problem (method recommendation given a feature request vs. method recommendation given a set of methods of interest), and leverage a different set of resources (historical feature requests + API documentation vs. structural dependencies).

The closest work to ours is that of Chan et al., which recommends API methods given textual phrases [5]. Given a query expressed as a set of textual phrases, their approach returns a connected API subgraph. For this, they create an API graph, which is an undirected graph with nodes corresponding to classes and methods and edges corresponding to relationships between them (e.g., inheritance, input, output, and membership). Each node of the graph contains words that appear in the corresponding method. Based on this API graph and an input query, they mine a subgraph that maximizes a particular objective function. The approach is evaluated on a small number of short text phrases. As we have found in Section VII, Chan et al.'s approach performs poorly on feature requests, which are both longer than the phrases considered by Chan et al. and are less closely related to the actual code. Our approach is able to treat feature requests due to the inclusion of information about historical feature requests, in the history-based recommendation component.

IX. CONCLUSION AND FUTURE WORK

In this work, we have proposed a new method recommendation approach that takes as input a feature request and recommends methods from a set of libraries. In contrast to previous approaches, our approach does not require precise input information, such as precise input or output types, or precise matching textual descriptions. Thus, it is suitable for directly processing feature requests stored in bug repositories, which often do not precisely specify relevant code elements. Our approach is a hybrid approach, combining history-based recommendation and description-based recommendation. on 10 libraries and 507 feature requests from 5 software systems we achieve an average recall-rate@5 and recall-rate@10 of 0.690 and 0.779 respectively. We have also compared our approach to the latest method recommendation technique that requires precise textual descriptions from end users and show that it is not useful for recommending methods from feature requests.

In the future, we plan to improve our solution further to achieve even higher recall-rate@k scores. Some possible directions include using state-of-the-art natural language processing [2], [33], taking the information stored in the code base into account, and specification mining e.g., [7], [12], [13], [14], [15], [16], [17], [18], [28]. We also plan to experiment with a larger number of feature requests from more software systems and to perform a more thorough investigation of the factors that affect the effectiveness of the different components of our approach in contributing towards the effectiveness of the proposed solution. Finally, to improve the practical usefulness of our approach, we plan to integrate our proposed solution into an IDE (e.g., Eclipse) and evaluate the resulting tool by means of a user study. We also want to investigate the effectiveness of our approach under different experimental settings, e.g., evaluating on projects with limited number of feature requests, cross-validating across projects, etc.

To extend our approach, we would like to consider how to enable users to specify some constraints to be taken into consideration in the recommendation process. We will also consider whether our proposed approach can be effective for bug reports in addition to feature requests.

ACKNOWLEDGEMENT

We would like to thank Wing-Kwan Chan and Hong Cheng for providing us with the source code of their tool for recommending API methods from precise text phrases. This project is partly supported by MERLION MS12C0014 grant.

REFERENCES

[1] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère, "Empirical evaluation of bug linking," in *CSMR*, 2013, pp. 89–98.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[3] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *ESEC/SIGSOFT FSE*, 2009, pp. 213–222.

[4] G. Casella and E. I. George, "Explaining the Gibbs sampler," *The American Statistician*, no. 3, pp. 167–174, 1992.

[5] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *SIGSOFT FSE*, 2012, p. 10.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[7] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 243–257, 2012.

[8] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[9] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *ICSM*, 2009, pp. 351–360.

[10] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *ASE*, 2011, pp. 524–527.

[11] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *DSN*, 2008, pp. 52–61.

[12] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Mining message sequence graphs," in *ICSE*, 2011, pp. 91–100.

[13] D. Lo and S.-C. Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *SIGSOFT FSE*, 2006, pp. 265–275.

[14] D. Lo, S.-C. Khoo, and C. Liu, "Mining past-time temporal rules from execution traces," in *WODA*, 2008, pp. 50–56.

[15] D. Lo and S. Maoz, "Specification mining of symbolic scenario-based models," in *PASTE*, 2008, pp. 29–35.

[16] ——, "Mining hierarchical scenario-based specifications," in *ASE*, 2009, pp. 359–370.

[17] ——, "Scenario-based and value-based specification mining: better together," in *ASE*, 2010, pp. 387–396.

[18] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," in *WCRE*, 2009, pp. 62–71.

[19] F. Long, X. Wang, and Y. Cai, "Api hyperlinking via structural overlap," in *ESEC/SIGSOFT FSE*, 2009, pp. 203–212.

[20] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI*, 2005, pp. 48–61.

[21] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[22] 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 *ASE*, 2012, pp. 70–79.

[23] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, and G. Antoniol, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," in *TSE*, 2007.

[24] S. Rao and A. C. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *MSR*, 2011, pp. 43–52.

[25] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *ESEC/SIGSOFT FSE*, 2005, pp. 11–20.

[26] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007, pp. 499–510.

[27] Z. M. Saul, V. Filkov, P. T. Devanbu, and C. Bird, "Recommending random walks," in *ESEC/SIGSOFT FSE*, 2007, pp. 15–24.

[28] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 651–666, 2008.

[29] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010, pp. 45–56.

[30] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.

[31] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *ASE*, 2007, pp. 204–213.

[32] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *WCRE*, 2011, pp. 92–96.

[33] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting paraphrases of technical terms from noisy parallel software corpora," in *ACL/IJCNLP*, 2009, pp. 197–200.

[34] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008, pp. 461–470.

[35] Q. Zhang, W. Zheng, and M. R. Lyu, "Flow-augmented call graph: A new foundation for taming api complexity," in *FASE*, 2011, pp. 386–400.

[36] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a static noninteractive approach to feature location," in *TOSEM*, 2006.

[37] 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 *ICSE*, 2012, pp. 14–24.