The Journal of Systems and Software 134 (2017) 355–368

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Enhancing developer recommendation with supplementary information via mining historical commits☆

Xiaobing Sun[a,c,*], Hui Yang[a], Xin Xia[b,d], Bin Li[a]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] Faculty of Information Technology, Monash University, Australia
[c] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[d] Department of Computer Science, University of British Columbia, Canada

ARTICLE INFO

Article history:
Received 10 November 2016
Revised 27 July 2017
Accepted 22 September 2017
Available online 22 September 2017

Keywords:
Collaborative topic modeling
Developer recommendation
Bug assignment
Personalized recommendation
Supplementary information recommendation
Commit repository

ABSTRACT

Given a software issue request, one important activity is to recommend suitable developers to resolve it. A number of approaches have been proposed on developer recommendation. These developer recommendation techniques tend to recommend experienced developers, i.e., the more experienced a developer is, the more possible he/she is recommended. However, if the experienced developers are hectic, the junior developers may be employed to finish the incoming issue. But they may have difficulty in these tasks for lack of development experience. In this article, we propose an approach, *EDR_SI*, to enhance developer recommendation by considering their expertise and developing habits. Furthermore, *EDR_SI* also provides the personalized supplementary information for developers to use, such as personalized source code files, developer network and source-code change history. An empirical study on five open source subjects is conducted to evaluate the effectiveness of *EDR_SI*. In our study, *EDR_SI* is also compared with the state-of-art developer recommendation techniques, *iMacPro*, *Location* and *ABA-Time-tf-idf*, to evaluate the effectiveness of developer recommendation, and the results show that *EDR_SI* can not only improve the accuracy of developer recommendation, but also effectively provide useful supplementary information for them to use when they implement the incoming issue requests.

1. Introduction

Bugs and issues continuously emerge during software maintenance and evolution (Rajlich, 2014; Canfora et al., 2014). Given a new issue request, software managers must assign the issue request to suitable developers to implement it (Xia et al., 2015; Servant and Jones, 2012). To accomplish this, managers or developers need to analyze the information available in the software repositories, e.g., bug or issue repositories, communication archives, and/or source-code change repositories (Zanjani et al., 2014; Wang and Lo, 2014; Kagdi et al., 2007), to recommend developers.

A large number of approaches focusing on developer recommendation have been developed (Hossen et al., 2014; Kagdi et al., 2012; Anvik and Murphy, 2011; Zhang et al., 2015; Shokripour et al., 2015). These techniques mainly focused on optimising the accuracy of developer recommendation. The basic thought behind these approaches is that a developer who has more development experience relevant to an incoming issue request is more suitable to implement this issue request. Based on this thought, a ranked list of developers is recommended in an order by their suitability (Hossen et al., 2014; Shokripour et al., 2013; Xia et al., 2013; Zhang et al., 2014a; Zhang and Lee, 2013). However, these techniques tend to recommend developers who have luxuriant development experience, which prejudices someone who just joined the team (i.e., newcomers). In our previous studies on developer recommendation based on commit repositories (Sun et al., 2017), we found that most developers are the junior ones since they only submitted a small number of commits (Yang et al., 2016). For example, Fig. 1 shows the number of submitted commits of each developer for *K-3D*,[1] *jEdit*,[2] *FreeMat*[3] and *SpringFramework*[4] subjects. We notice that, on average, most developers submitted fewer than 50 commits, considering that we analyzed 45,704 commits of four sub-

☆ Fully documented templates are available in the elsarticle package on CTAN.

* Corresponding author.
E-mail addresses: sundomore@163.com, xbsun@yzu.edu.cn (X. Sun), xxia02@cs.ubc.ca (X. Xia).

[1] http://sourceforge.net/projects/k3d/?source=directory
[2] http://sourceforge.net/projects/jedit/
[3] http://sourceforge.net/projects/jedit/
[4] http://sourceforge.net/projects/springframework/?source=directory

https://doi.org/10.1016/j.jss.2017.09.021
0164-1212/

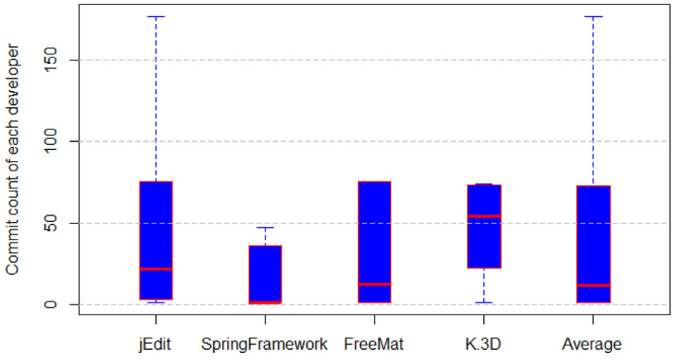

Fig. 1. The number of submitted commits of every developer for *K-3D, jEdit, FreeMat* and *SpringFramework* subjects.

jects in total. Thus, the proposed issues are not always resolved by senior developers with rich maintaining experience. In practice, the developer with rich development experience may be arranged for more important tasks, and they usually have heavy tasks and may be not available for some trivial tasks. On the other hand, the junior developers are not acquainted with the system, and they have difficulty in finishing the incoming issue request.

Hence, recommending developers who have ample relevant experience for the incoming issue request is not always effective to promote the entire process of software evolution (Zhang et al., 2015). Developers' habits are also important during an issue resolution (Sun et al., 2017). For example, in the practical software maintenance, developers tend to modify the source files that they have changed before. The more times they have modified, the more possibility they will change again. The more recent they have modified the files, the more possibly they will change again. In this article, we consider developers' expertise and developing habits to recommend developers, which could avoid only recommending senior developers for the software issue requests. Besides, for the junior developers, they may have difficulties in resolving the issue requests. So personalized supplementary information are also recommended to them and help them get relevant knowledge of the system, which can effectively assist them in finishing the assigned issue requests (Yang et al., 2016).

In this article, we propose an approach named *EDR_SI* (Enhancing Developer Recommendation with Supplementary Information), which enhances developer recommendation by taking their development habits and experience into consideration as well as recommending the personalized supplementary information (for the candidate developers) relevant to the issue request, such as personalized source code files, developer network and source-code change history. *EDR_SI* uses the collaborative topic modeling (*CTM*) technique to analyze the historical commit repositories. *CTM* combines the merits of traditional collaborative filtering and probabilistic topic modeling to provide an interpretable latent structure for users and items (Wang and Blei, 2011). That is, *CTM* could generate more diverse and novel supplementary information for each recommended developer. In our approach, we mainly apply *CTM* to automatically analyze developer expertise and recommend the relevant source code files to them. All the recommended files are personalized for each developer, and these files are potentially needed to be modified for finishing the issue at hand. Then, other supplementary information, such as developer network and source-code change history, are recommended based on the personalized source code files.

To evaluate the effectiveness of our approach, we conduct an empirical study on five open source systems/repositorise (*jEdit, JDT-Debug, Hadoop, Elastic* and *Libgdx*). The results show that *EDR_SI* can not only effectively recommend developers, as well as the personalized supplementary information, but can also improve the accuracy of the developer recommendation compared to the state-of-art developer recommendation techniques, *iMacPro, Location* and *ABA-Time-tf-idf*.

Developer recommendation is more concerned by bug triagers while supplementation information would be finally used by developers. In our approach, we think that when our approach recommends developers with supplementary information, bug triagers can better understand the bug assigning procedure based on the supplementary information why the incoming bugs are assigned to some developers rather than directly used the recommendation results generated by a developer recommendation approach. This article extends a preliminary study published as a research paper in a conference (Yang et al., 2016). It extends the preliminary study in various ways: (1) *EDR_SI* provides more personalized supplementary information for each developer, such as developer network and source-code change history. The empirical study shows the effectiveness of these supplementary information. (2) We evaluate the developer recommendation of *EDR_SI* to show that our approach can recommend junior developers well. That is, if the experience of junior developers can be efficiently extended, they (junior developers) can be recommended effectively. (3) A wider experiment with more subjects and metrics, is performed to fully evaluate the proposed *EDR_SI*.

The main contributions of this article are as follows:

1. *EDR_SI* enhances developer recommendation by considering developers' expertise and developing habits, which can avoid only recommending senior developers for the software issue requests. To help developers implement the issues, *EDR_SI* provides a series of personalized supplementary information for them to use, such as personalized source code files, developer network and source-code change history.
2. An empirical study on a broad range of datasets containing a total of 45,704 commits to demonstrate the effectiveness of *EDR_SI* is conducted. The results show that *EDR_SI* improves the accuracy of developer recommendation over the state-of-art techniques, i.e., *iMacPro, Location* and *ABA-Time-tf-idf*.

The rest of this article is organized as follows. The preliminaries are introduced in Section 2. Section 3 presents our approach. Section 4 shows an example of using *EDR_SI* and illustrates implementation of *EDR_SI*. Section 5 shows our empirical study. Section 6 discusses threats to validity in our empirical study. Section 7 discusses the related work. Finally, Section 8 concludes this article and shows the future work.

2. Preliminaries

In this article, *EDR_SI* employs the Collaborative Topic Modeling (CTM) technique to recommend developers by exploring commit repository. This section introduces preliminaries from three aspects: commit repository, CTM, and developer recommendation.

2.1. Commit repository

In commit repository, a commit message includes the commit date, committer's name, commit description, modified files, and a unique ID, as shown in Fig. 2. These information can well reflect developers' historical developing expertise and habits, which is an important data source for developer recommendation.

The commit description (shown with the red underline in Fig. 2) reflects developers' expertise. For example, the key words in the commit #22000, such as *fix, NPE, remove, plugin*, indicate that *kpouer* has the experience to fix the *NPE* problem (Null Pointer Exception) when removing a plugin. So we can train the key words in

Fix a case where a NPE occurs when removing a plugin (#3557441)

Authored by: kpouer 2012-08-14

Browse code at this revision

Parent(s): [r21999]

Child(ren): [r22001]

change /jEdit/trunk/org/gjt/sp/jedit/PluginJAR.java (diff)
change /jEdit/trunk/doc/CHANGES.txt (diff)

Fig. 2. An example of commit #22000 in the commit repository of the *jEdit* subject.

the commit repository to analyze developers' personalized expertise. Furthermore, the modified files' pathes may reflect the developing habits for some specific issues. For example, for the issue of *NPE, kpouer* mainly changed the source code file of *PluginJAR.java*. Some other source files related to the plugin may exist, such as *PluginManager.java, PluginUpdate.java, PluginOptions.java, Plugin.java* and so on. But *kpouer* never modified them. On the other hand, the commit date also records the changing time of these files, which can reflect the familiarity of developers for the changed files. The more recent these files have been changed, the more probable that developers are familiar to them. So *EDR_SI* applies the data in the commit repository to analyze developers' personalized expertise and developing habits, and then utilized these information for developer recommendation.

2.2. Collaborative topic modeling

EDR_SI uses collaborative topic modeling technique (*CTM*) (Wang and Blei, 2011; Wang et al., 2013; Chen et al., 2014; Purushotham and Liu, 2012; Kang and Lerman, 2013) to recommend the personalized files for candidate developers. Compared with the traditional collaborative filtering, *CTM* combines ideas from collaborative filtering based on latent factor models and content analysis based on probabilistic topic modeling, e.g., the latent Dirichlet allocation (*LDA*) (Blei et al., 2003), which is widely used in dealing with software engineering data (Sun et al., 2015a; Hu et al., 2015; Sun et al., 2015b; 2016).

The two elements in *CTM* are users and items. In our problem, items are relevant source files and users are developers. We assume that there are I users and J items. The rating variable $r_{ij} \in [-1,1]$ denotes whether user i changes file j for the issue request implementation (Hu et al., 2008). If the value of r_{ij} is positive, user i will change file j. This means that the file j is relevant to the issue request and this file is also in line with the developer's development habits. The more greater the value of r_{ij} is, the more possibly user i will change file j. Note that $r_{ij} < 0$ can be interpreted from two aspects. One is that user i is unfamiliar with file j, and the other is that file j is unrelated to the issue request.

CTM recommends items for users based on the topic word unit (Wang and Blei, 2011). That is, *CTM* first analyzes topics of all the items, and then gathers all the similar topic items for users. For example, if the relevant source file j was changed by the developer d, *CTM* can collect other files whose topic(s) is/are similar to the file j, and recommend these files for developer d. Moreover, all the changed files are different for different developers in the commit repository. So the recommended files are personalized for each developer. Hence, we apply *CTM* to recommend personalized files for each developer. Fig. 5 shows the schematic diagram for recommending personalized files by *CTM*. As Fig. 5 demonstrates, relevant developers and files they changed before are set as the *CTM*'s input data. After the iteration calculation by *CTM*, it generates the relevant developers associated with a list of source files.

Fig. 3. Historical developing experience of *arobert* and *vanza* in *jEdit* subject.

All these files are personalized for each developer and sorted with the ranking weight, which ranges from −1 to +1. When the value is positive, *CTM* will recommend the file for the developer.

In summary, *CTM* recommends personalized files for each developer from two perspectives. One is the files that developers changed before, which are recommended based on their historical developing habits. And these files can improve the developers' efficiency of issue resolution. The other is the files that the developers did not change before, but these files are useful for the issue resolution as the topic of these files is similar or relevant to the relevant files that the developers have changed before. So these files can help developers get more knowledge about the incoming issue and generate more diverse and novel suggestions for each developer, which can assist them in comprehending more about the system.

2.3. Developer recommendation

Existing developer recommendation techniques mainly focused on optimising the accuracy of recommendation results (Hossen et al., 2014; Kagdi et al., 2012; Anvik and Murphy, 2011; Zhang et al., 2015; Shokripour et al., 2015). The basic thought is that a developer with more development experience relevant to an issue request is more suitable to implement the incoming issue. In this way, senior developers with more development experience are always recommended.

In practice, the developing experience of the junior developers is generally simple, in other words, they may be more suitable to resolve some special issues. For example, Fig. 3 shows a *wordcloud* diagram which reflects the historical developing experience (extracted from commit repository) of two developers (*arobert* and *vanza*) in the *jEdit* subject. The left part of Fig. 3 represents the developing experience of *arobert*, and he/she submitted 12 commits before Nov. 12, 2014 (seems to be a junior developer). From the left *wordcloud* diagram, some key words, such as *button, panel, tab, selected, color...*, indicate that *arobert* may be skilled at resolving the system interface issues. Perhaps we can recommend the issues related to the system interface to him/her, although *arobert* is not familiar to the whole system.

For senior developers, their experience in the system may be more general, which is reflected by the key words, such as *action, file, project, version, plugin, import...* in the right *wordcloud* of Fig. 3. The right *wordcloud* diagram shows the developing experience of *vanza*, and he/she submitted more than 1000 commits before Nov. 12, 2014. That is, some more complex issues can be assigned to him/her.

Hence, if developers are recommended by considering their personalized developing experience, which can be reflected by the submitted commits and the changed source code files, the junior developers can also be recommended for some issues, which avoid tending to always recommend the senior developers.

3. Approach

In practical software maintenance, on the one hand, the developers with rich work experience (as senior developers) are usually

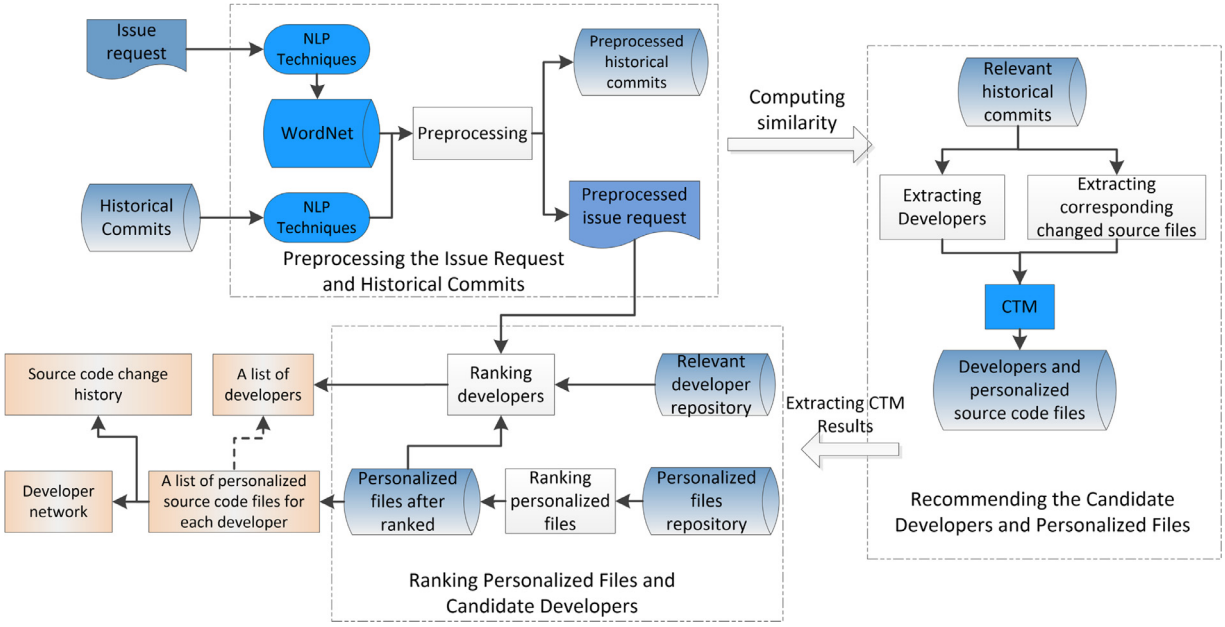

Fig. 4. Process of *EDR_SI*.

assigned with more important or complicated tasks. In this case, the senior developers usually have heavy tasks and may be not available for some trivial tasks. On the other hand, the junior developers who have less relevant expertise should also be arranged to implement some simple change tasks. But the junior developers are not acquainted with the system, and they may have difficulty in finishing the issues.

So faced with a bug, there may be an embarrassing situation, i.e., the senior developers who have ample relevant experience may be unwilling to modify the recommended files as they are unfamiliar with them, while the junior developers may need to comprehend other relevant files except the recommended files because they are not acquainted with the system enough. So recommending the personalized supplementary information fit for developers' expertise can improve the effectiveness of the resolution of an incoming issue.

Hence, *EDR_SI* recommends the relevant source code files by considering developers' expertise. These personalized files can help developer(s) comprehend software bugs and the system. Moreover, *EDR_SI* also constructs developer network and analyzes source-code change history based on personalized source code files, which can help the junior developers comprehend the system and software bugs more easily and conveniently find other developers for communication and guidance.

The process of our approach is shown in Fig. 4. First, the new issue request and historical commits are preprocessed. Then, we analyze and extract the commits relevant to the issue request. The authorship information is extracted from these relevant commits. These authors will be recommended as potential candidate developers for resolution of the current issue request. Meanwhile, we also extract the changed source code files corresponding to these relevant commits. Finally, we take the authorship information and relevant source code files as input data, and use *CTM* to recommend the relevant source code files for each candidate developer to facilitate the issue request implementation. The recommended candidate developers are ranked in a list based on their personalized relevant source code files, and the personalized files are ranked based on their historical development habits.

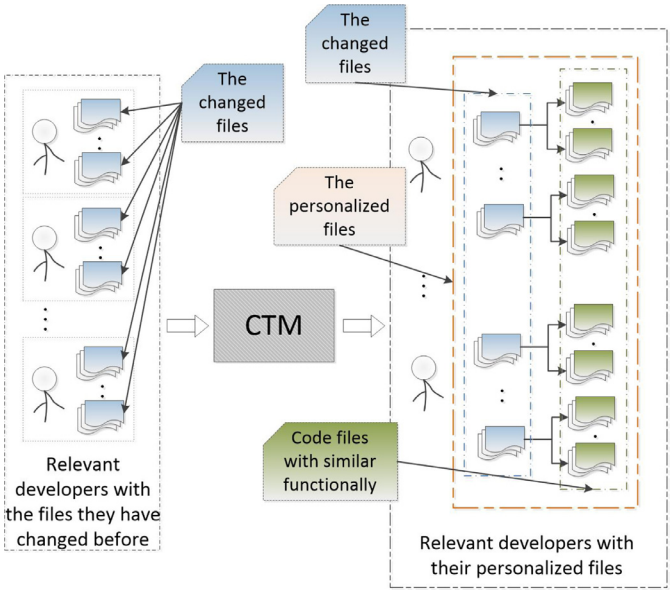

Fig. 5. The schematic diagram for personalized source files recommended by *CTM*.

3.1. Analyzing historical commits

While analyzing the historical commits, we need to first preprocess them. Natural language processing (NLP) techniques are usually used to perform one or more preprocessing operations to remove the noisy data (Sun et al., 2014). There are several typical preprocessing operations for the unstructured commit messages, which include tokenization, unrelated and unimportant words removing, stemming, etc. In our approach, the preprocessing operations are shown as follows:

- Splitting the words such as camel case ("*TestCase*") and underscores ("*test_case*").
- Removing common English stop words ("*the, by, on, and, no...* ") to reduce noise.

- Stemming the remaining words (e.g., "*testing*" becomes "*test*") to reduce the vocabulary size. The stemming algorithm is referred from Porter stemmer (Porter, 2006).
- Searching synonyms of the verbs and nouns via WordNet[5] as extended words, e.g., "evaluate" is a synonym for "test". But some extended words will be removed if these words do not appear in the source code, which can reduce much noise data for *EDR_SI*.

Then, the similarity between the commits and the new issue request is computed based on the cosine function, which is shown as follows:

$$Similarity = \frac{|Historical\ \ Commit \cap Issue\ \ Request|}{|Historical\ \ Commit|} \quad (1)$$

In Formula (1), *Historical Commit* represents the words in the preprocessed commit description. *Issue Request* represents the words in the preprocessed issue request (the new issue description).

Finally, a list of historical commits is ranked by the similarity values, and we extract the relevant historical commits which have the similarity value over a threshold (Θ) for the following recommendation.

In summary, this step preprocesses the historical commits and issue request, and identifies the relevant commits with similarity value over a threshold (Θ).

3.2. Recommending developers and personalized source code

In the above step, we have obtained relevant historical commits. Developers are then recommended based on the relevant historical commits. These recommended developers are considered to have conducted the relevant software maintenance tasks (to the new issue request) in previous maintenance activities. Hence, they are recommended as the appropriate developers (candidate developers) to implement the new issue request.

When obtaining relevant commits and their corresponding developers, we can also extract the changed source code files corresponding to the relevant commits. We define $Author\_Files_{commit_i}$ to record the authorship information and the changed source code files for each commit $commit_i$ in the form of

$$Author\_Files_{commit_i} =< Authors, Files > \quad (2)$$

In Formula (2), *Authors* are the developers who submitted this commit ($commit_i$) and *Files* are the changed source code files in the commit. In our approach, the changed source code files are also recommended to each developer to help them resolve the new issue request. However, a new issue request is usually different from the historical commits. Using only the existing changed source code files is not enough to implement the issue request. In addition, sometimes a new issue request may be relevant to quite a few of historical commits. Simply merging these changed source code files of different historical commits together can make the developers even more difficult to determine how they should implement the new issue request. Here we set $Author\_Files_{commit_i}$ as the input data and use *CTM* to recommend personalized source code files for each developer.

CTM combines traditional collaborative filtering with topic modeling (Wang and Blei, 2011; Wang et al., 2013; Chen et al., 2014; Purushotham and Liu, 2012; Kang and Lerman, 2013). Fig. 5 shows the schematic diagram for recommending personalized files by *CTM*. As Fig. 5 shows, the relevant developers and files they changed before are set as the *CTM*'s input. After the iterative calculation by *CTM*, the output data is the relevant developers associated with a list of source code files. All these files are personalized for each developer and sorted by the ranking weight, which ranges from −1 to +1. When the value is positive, *CTM* will recommend the file for the developer. In summary, *CTM* recommends personalized files for each developer from two perspectives. One is the files that the developer changed before, which are recommended based on his historical developing habits. And these files can improve the developer's efficiency for issue resolution. The other is the files that the developer did not change before, but these files are useful for the issue resolution as the topic of these files is similar or relevant to the files that the developer changed before. So these files can help the developer get more knowledge about the incoming issue and generate more diverse suggestions for each developer, which can assist them in comprehending more about the system.

The other advantage of using *CTM* is that it can help *EDR_SI* recommend junior developers. That is, the experience of junior developers are usually less, and the number of their changed source code files is small. So the junior developers cannot be recommended well only according to their existing historical work data, which is the reason for traditional techniques only recommending senior developers (Hossen et al., 2014; Kagdi et al., 2012; Anvik and Murphy, 2011; Zhang et al., 2015; Shokripour et al., 2015). So we apply *CTM* to extend junior developers' personalized files to resolve this problem. In this way, *EDR_SI* will rank all the candidate developers based on their personalized files (as discussed in Section 3.4), which can recommend junior developers as well.

3.3. Ranking developers

After recommending the developers, we need a ranked list of them to show their suitability for an issue request. An easy way to rank the recommended developers is to rank them by the similarity of their commits to the new issue request. However, the commits in software repositories are often short and cannot fully conform to the current maintenance task. Instead, we mainly apply developers' personalized files to rank them, because the source code text is more comprehensive and useful to show their relevance to a commit. Moreover, all the developers' personalized files have been extended by *CTM*, which can alleviate the cold start problem[6] for the junior developers. So ranking developers based on their personalized files can avoid only recommending senior developers.

We compute the frequency of words in the new issue request occurred in the recommended source code files, defined as follows:

$$Frequecy = \frac{|Issue\ \ Request \cap Relevant\ \ Source\ \ code|}{|Relevant\ \ Source\ \ code|} \quad (3)$$

In Formula (3), *issue request* represents the words in the preprocessed issue request (issue description) based on the preprocessing steps in Section 3.1. *Relevant Source code* represents the words in the relevant source code files (the identifiers in the code and the words in the comments). Finally, a list of recommended developers is ranked based on the *Frequency* metric.

3.4. Ranking personalized source code files

After recommending the personalized source code files by *CTM*, we need to rank these files according to developers' historical developing habits. An easy way to rank the personalized source code files is to rank them by the *CTM* results. In addition, some more useful information recorded in the commit messages can be

[5] http://wordnet.princeton.edu/

[6] The cold start problem is that the existing historical data of junior developers is less than that of senior developers, so the junior developers cannot be well recommended.

used to rank these personalized files, e.g., the commit date, the committer's name, the commit description, and the modified files' paths. Many other approaches also ranked relevant files with commit messages (Kagdi and Poshyvanyk, 2009; Hossen et al., 2014; Shokripour et al., 2013). The premise is that the files that the developer changed with more times and most recently should be ranked higher. We combine these measures together to identify the probability that a developer that is likely to change a file, i.e., *developer-file map*. The developer-file map is represented via the developer-file vector *DF* for the candidate developer *d* and file *f*:

$$DF = < CTM_f, Times_f, Rdate_f, Similarity_f > \quad (4)$$

where:

- CTM_f is the *CTM* value generated by the *CTM* computation.
- $Times_f$ is the number of commits submitted by the developer *d* including file *f*.
- $Rdate_f$ is the negative distance value between the submitting workday of the issue request and the most recent workday the developer *d* submitting a commit that includes the file *f*.
- $Frequency_f$ is the similarity value between the issue request and the file *f* changed by the developer *d*, which is calculated by Formula (3).

Then, we calculate the sum of each element in the vector *DF* as the $weight_{<d, f>}$ value, which represents the tendency that the developer *d* will change the file *f* to implement the issue request. The bigger is the $weight_{<d, f>}$ value, the more possible the developer *d* will change the file *f*.

3.5. Constructing developer network

Senior developers sometimes are not available to implement the incoming issues as they have more complex or emergent change tasks. To help junior developers quickly comprehend and resolve software issues, *EDR_SI* constructs a developer network, which can help find other developers for communication. Each recommended developer has his/her own personalized files, and these files can well reflect the individual developer's relevant expertise related to the incoming issue request. The larger number of the same personalized files is, the more similar of relevant expertise between the two developers is. Hence, we can construct the network of recommended developers based on the number of their shared personalized files.

3.6. Analyzing files' change history

During the process of issue resolution, developers may encounter problems. At this time, they may seek some relevant experienced developers to communicate. So we analyze the changing history of each personalized file for the recommended developers, i.e., we analyze the developers who have changed these files, the changing counts of each developer, and their recent changing time. These information can assist developers in quickly finding other experienced developers to communicate with when they have problems during the issue resolution.

4. Implementation

In this section, we describe an example of using *EDR_SI* and illustrate its implementation. We developed *EDR_SI* with *Java* language. Fig. 6 outlines a screenshot of *EDR_SI*.

Given a new issue request, *EDR_SI* recommends a ranked list of suitable developers in Interface (1) in Fig. 6, where the top one of List *A* is the most suitable developer. Meanwhile, *EDR_SI* also recommends some personalized files for the recommended developer to refer, as shown in List *B*. Moreover, if the recommended developer has few knowledge about the incoming issue request, they can refer to the extended description of the issue request in Box *C*. The description includes some topic words which directly reflect the content of a change task. On the other hand, if the most suitable developer is not available for the issue implementation, the junior developers can be recommended from the Network *D*. Network *D* in Interface (2) reflects the relevant expertise of other recommended developers to the most suitable one (e.g., *Jingzhao*). In the network, nodes represent developers, and edges represent the personalized files of other developers which are similar to *Jingzhao*. So if the junior developer has difficulties in understanding the issue request, he/she can conveniently find the most suitable or other experienced developers to communicate with. Furthermore, List *E* shows the same personalized files among the recommended developers. Developers can also refer to the source code (shown in Interface (4)) to see whether this source-code file is related to the issue request.

In addition, if developers encounter some problems when they change a source code file to implement the issue request, they can seek other experienced developers to communicate or collaborate. To do this, *EDR_SI* analyzes the changing history of the personalized files. The analyzed results mainly include the developers who changed the files, the changing count, and the recent changing time. All the information of source-code change history is shown in Table *F* of Interface (3). Developers can refer to this table and quickly find relevant experienced developers to communicate and/or consult.

Table 1
Subject systems and their characteristics.

Subject	Commit	Author	File	Time Interval
jEdit	23,724	133	113,224	From 2006.07.01 to 2014.11.12
Hadoop	10,394	82	7592	From 2001.05.18 to 2015.03.31
JDT-debug	9104	47	1860	From 2009.09.04 to 2015.03.26
Elastic	22,191	661	4722	From 2010.02.08 to 2016.05.26
Libgdx	12,414	345	1860	From 2010.03.07 to 2016.05.26

5. Empirical study

5.1. Study subject

To evaluate the effectiveness of *EDR_SI*, we conducted an empirical study on five open source systems, e.g., *jEdit, hadoop, JDT-Debug, Elastic* and *Libgdx*. The characteristics of each subject system are shown in Table 1. The "Subject" column shows the selected subject system; the "Commit" column shows the number of historical commits used in our empirical study; the "Author" column shows the number of developers in our study; the "File" column shows the average number of files for each system; and the "Time Interval" column shows the time interval of selected historical commit messages for each subject. *jEdit*[7] is a programmer's text editor written in Java. *Hadoop*[8] is an open-source software for reliable, scalable, and distributed computing. *JDT-debug*[9] implements Java debugging support and works with any JDPA-compliant target Java VM. *Elastic*[10] is a distributed, open source search and analytics engine, designed for horizontal scalability, reliability, and easy management. *Libgdx*[11] is a Java game development framework

[7] http://sourceforge.net/projects/jedit/
[8] https://hadoop.apache.org/
[9] https://projects.eclipse.org/projects/eclipse.jdt.debug
[10] https://www.elastic.co/
[11] https://libgdx.badlogicgames.com/

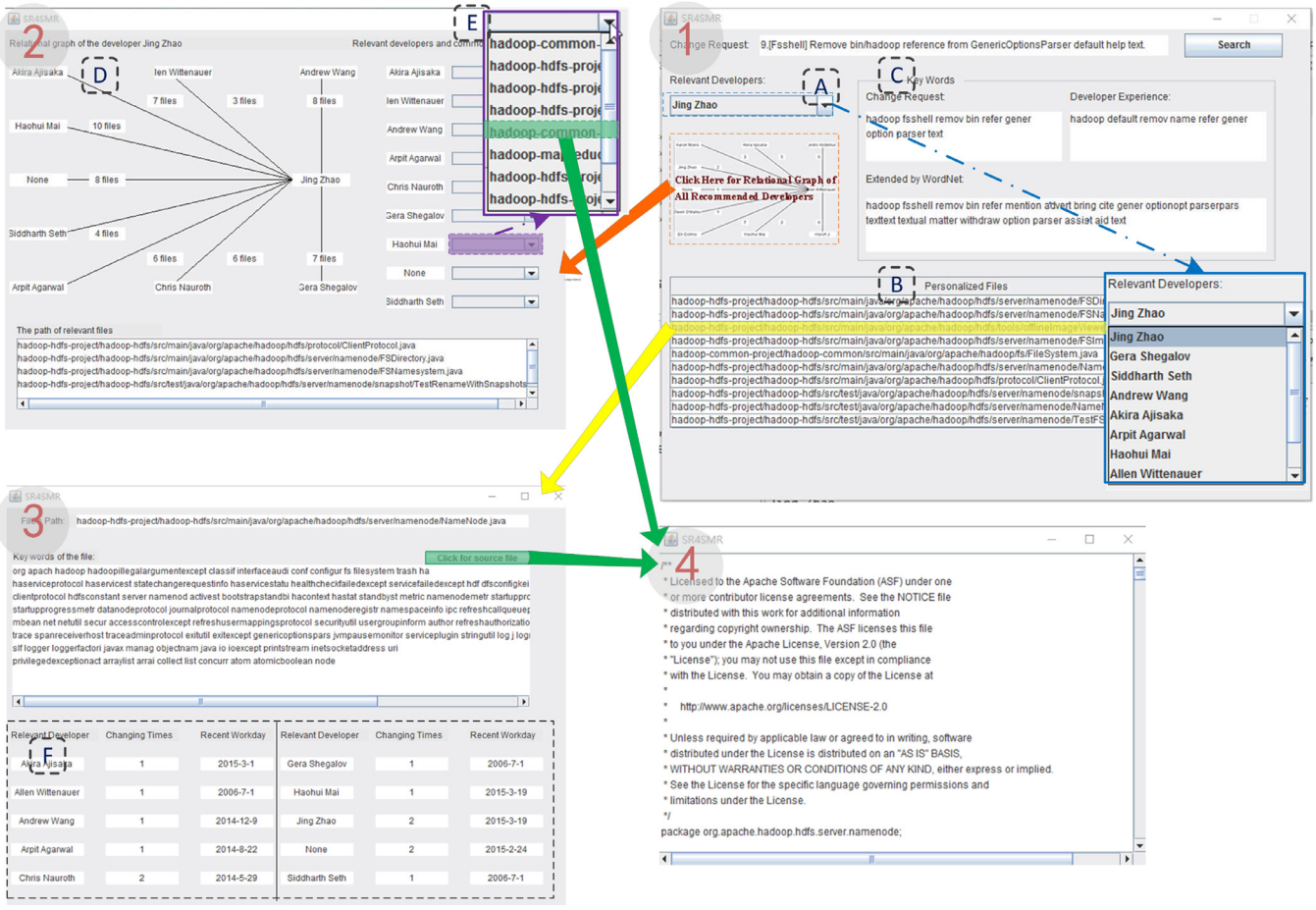

Fig. 6. Screenshot of *EDR_SI*.

which provides a unified API that works across all supported platforms.

5.2. Empirical setup

The purpose of *EDR_SI* is to recommend developers, which can avoid tending to recommend senior developers for a new issue request. Moreover, for each recommended developer, *EDR_SI* aims to recommend personalized supplementary information to help them implement the issue request. The following research questions are studied to show the effectiveness of *EDR_SI*.

RQ1: Can *EDR_SI* improve the accuracy of recommending developers compared to previous developer recommendation approaches, i.e., *iMacPro*, *Location* and *ABA-Time-tf-idf*?

RQ2: Does *EDR_SI* tend to recommend senior developers when evaluated on open-source systems?

RQ3: Is the effectiveness of *EDR_SI* to recommend personalized files improved compared to a typical bug localization technique, i.e., *BRTracer*?

RQ4: Can the developer network effectively assist junior developers in comprehending and resolving software issues by identifying other developers?

RQ5: Can the source-code change history effectively identify other developers to communicate with if some difficulties occur while changing a specific source file?

Traditional developer recommendation approaches mostly recommended developers without considering their development experience and habits. *EDR_SI* combines developers' development experience and habits to recommend developers, so *RQ1* is to evaluate how well the accuracy of *EDR_SI* recommending developers is compared to the state-of-art approaches, i.e., *iMacPro, Location* and *ABA-Time-tf-idf*.

There are also junior developers in a development team, who need to perform some development tasks. *EDR_SI* cannot only recommend the senior developers, but also the junior developers. So *RQ2* is to evaluate whether *EDR_SI* can identify correct junior developers to implement the issue requests.

There have been a large number of bug location techniques, which aimed at optimizing the accuracy of predicting faulty files. *EDR_SI* can also help locate the faulty files considering developers' development experience and habits. So *RQ3* is to evaluate how well the accuracy of *EDR_SI* locating the personalized source code files is and how well the recommended developers are familiar with these files compared to those by the bug location techniques, such as *BRTracer*.

Sometimes the junior developers may have difficulties in understanding and resolving software issues. *EDR_SI* also constructs the developer network, which can recommend senior developers for junior developers to communicate with. *RQ4* is to evaluate the effectiveness of developer network in recommending the senior developers.

Developers usually encounter difficulties while changing the source code. Communicating with other experienced developers who are familiar with the source code is a good way to comprehend the source code at hand. *EDR_SI* also analyzes the code change history. *RQ5* is to evaluate the effectiveness of the code change history in finding the experienced developers related to a specific source code file.

5.3. Study method

To perform our study, we randomly chose 200 commits from each subject as the training issue requests, and other commits as the test issue requests for study.

5.3.1. For developer recommendation

To answer **RQ1**, we first evaluate the accuracy of developer recommendation using the recall metrics as in the previous work (Hossen et al., 2014). For r number of issue requests in the benchmark of a system and k number of recommended developers, *recall@k* is defined as follows:

$$recall@k = \frac{1}{r} \frac{\sum_{i=1}^{r} |RD(r_i) \cap AD(r_i)|}{|AD(r_i)|} \quad (5)$$

where $RD(r_i)$ and $AD(r_i)$ are the number of the recommended developers by *EDR_SI* and the actual developers who resolved the issue request r_i, respectively. The metric is computed for the lists of recommending developers with different sizes, i.e., k = 1, k = 5, and k = 10. The reason for not using the other popular metric precision is that an issue request typically has one developer implementing it, i.e., $|AD(r_i)| = 1$. Therefore, for k = 1 to 10, there is typically only one correct answer and others are incorrect.

Then, we use three previous developer recommendation techniques for the comparative study. Hossen et al. proposed the *iMacPro* approach, a typical text-based approach, which integrates authors and maintainers of relevant source-code files, which are change prone, to a given issue request (Hossen et al., 2014). Shokripour et al. proposed an approach to recommend developers that is based on location of potential faulty files (*Location*) (Shokripour et al., 2013). *Location* utilizes a noun extraction process on four information sources to determine bug location information and a simple term weighting scheme to recommend candidate developers. Later, they further proposed the Time-Text-Based approach, called *ABA-Time-tf-idf*, which applies the time-metadata in tf-idf (Time-tf-idf) technique. The recency of using the term by the developer is considered to determine the developers' expertise (Shokripour et al., 2015). More details for *iMacPro, Location* and *ABA-Time-tf-idf* can refer to Hossen et al. (2014), Shokripour et al. (2013) and Shokripour et al. (2015), respectively. *iMacPro, Location* and *ABA-Time-tf-idf* represent three types of developer recommendation techniques, respectively. *iMacPro* represents the approach for the developer recommendation based on the content analysis; *Location* represents the approach for the developer recommendation based on the location analysis; and *ABA-Time-tf-idf* represents the approach which recommends the developers considering the time factor. So those three approaches can represent the state-of-the-art techniques.

We calculate the developers' accuracy (*recall@k*) of *EDR_SI, iMacPro, Location* and *ABA-Time-tf-idf*, as well as the recall gain of *EDR_SI* over *iMacPro*, the recall gain of *EDR_SI* over *Location*, and the recall gain of *EDR_SI* over *ABA-Time-tf-idf*, as shown in Formula (6), (7) and (8), respectively:

$$gain@k_{\text{EDR\_SI}-\text{iMac}} = \frac{recall@k_{EDR\_SI} - recall@k_{\text{iMac}}}{recall@k_{\text{iMac}}} \times 100\% \quad (6)$$

$$gain@k_{\text{EDR\_SI}-\text{Location}} = \frac{recall@k_{\text{EDR\_SI}} - recall@k_{\text{Location}}}{recall@k_{\text{Location}}} \times 100\% \quad (7)$$

$$gain@k_{\text{EDR\_SI}-\text{ABA}} = \frac{recall@k_{\text{EDR\_SI}} - recall@k_{\text{ABA}}}{recall@k_{\text{ABA}}} \times 100\% \quad (8)$$

where $recall@k_{EDR\_SI}$, $recall@k_{\text{iMac}}$, $recall@k_{Location}$, and $recall@k_{\text{ABA}}$ represent the recall values of developer recommendation calculated by *EDR_SI, iMacPro, Location*, and *ABA-Time-tf-idf* for different k values, i.e., k = 1, k = 5, and k = 10, respectively.

Table 2
Four groups of issue requests classified by the actual developers' developing experience.

Group	G1	G2	G3	G4
Commit	<= 200	200 ~ 500	500 ~ 1000	> 1000

To answer **RQ2**, the selected 200 issue requests are averagely grouped into four groups by developers' development experience, and the specific characteristics of each group are illustrated in Table 2. The "Group" row represents four groups of issue requests, and the "Commit" row shows the number of commits submitted by the actual developers. The number of submitted commits reflects developers' historical development experience, i.e., the more the commits were submitted, the more experienced the developer is. For each group, we calculate the accuracy *recall@k* of developer recommendation. We compare the value of *recall@k* between different groups. If the difference of *recall@k* is not big, we consider *EDR_SI* could not tend to recommend senior developers.

5.3.2. For personalized supplementary information recommendation

For **RQ3**, we evaluate the effectiveness of these files from two aspects: (1) whether *EDR_SI* can identify correct files for developers to use; (2) whether the recommended developers are skilled to these personalized files. These metrics are defined as follows:

- *ATNF* (accuracy of top N files) (Wong et al., 2014) represents the percentage of issues whose associated files are ranked in the top N (N=1,5,10) returned results. Given an issue request, if the top N query results contain at least one file in which the bug should be fixed, we consider that the issue request is correctly located. The higher the metric value is, the better the faulty files are identified.
- *STNF* (skillful for top N files)[12] represents the percentage of issues whose recommended developers (the most suitable developers) are familiar with the top N (N=1,5,10) personalized files. Given an issue, if the top N query results contain at least one file that the recommended developer changed in the practical maintenance activity (for a commit), we consider that the issue was located. The higher the metric value is, the better the source-code files are identified.

Then, we compared *EDR_SI* with a typical bug location technique, i.e. *BRTracer* (Wong et al., 2014), based on the measures defined above. *BRTracer* divides the corpus extracted from source-code files into several segments to match the bug report for the faulty files, as well as applying stack-trace information, which focuses on optimizing the accuracy of locating the faulty files for the relevant developer(s) to use. More details about *BRTracer* can refer to Wong et al. (2014).

For **RQ4**, we calculate the ratio of effective networks which contain the senior developer(s) to help junior developers resolve the issue requests. The calculation is defined as follows:

$$ratio = \frac{effective\ network}{networks} \quad (9)$$

where *effective network* represents the number of issues for which the actual developer is a junior developer (who submitted fewer than 100 commits), but more than one senior developer(s) (who submitted more than 1000 commits) are in the recommended network. And *networks* represent all the issues in the evaluation, in

[12] For *BRTracer* in our comparative study, *STNF* is the percentage of bug reports (resolved bugs) whose actual developers are familiar with the top N (N=1,5,10) recommended files. Because *BRTracer* only recommends faulty files without developer recommendation. Given a bug report, if the top N query results contain at least one file that the actual developer changed in the historical maintenance activity, we consider that the bug report is located.

Table 3
Recall@1,5,10 of developer recommendation of *EDR_SI, iMacPro, Location* and *ABA-Time-tf-idf* for five subjects.

Subject	*Top k*	*EDR_SI*	*iMacPro*	*Location*	*ABA-Time-tf-idf*	Gain over *iMacPro%*	Gain over *Location%*	Gain over % *ABA-Time-tf-idf %*
jEdit	1	0.280	0.150	0.264	0.279	87.0%	6.1%	0.4%
	5	0.601	0.461	0.570	0.559	30.4%	5.4%	7.5%
	10	0.798	0.545	0.732	0.718	46.4%	9.0%	11.1%
Hadoop	1	0.085	0.072	0.068	0.079	23.2%	18.1%	7.6%
	5	0.301	0.227	0.242	0.250	32.6%	24.3%	20.4%
	10	0.503	0.417	0.448	0.462	20.6%	12.2%	8.9%
JDT-debug	1	0.144	0.127	0. 129	0.137	13.4%	11.6%	5.1%
	5	0.466	0.437	0.457	0.451	6.6%	2.0%	3.3%
	10	0.664	0.536	0.603	0.585	23.9%	10.1%	13.5%
Elastic	1	0.136	0.130	0.131	0.128	4.7%	3.8%	6.3%
	5	0.436	0.363	0.387	0.417	20.1%	12.7%	4.6%
	10	0.752	0.706	0.712	0.718	6.5%	5.6%	4.7%
Libgdx	1	0.220	0.216	0.215	0.217	1.9%	2.3%	1.4%
	5	0.513	0.448	0.453	0.456	14.5%	13.2%	12.5%
	10	0.696	0.578	0.618	0.612	20.4%	12.6%	13.7%
Average	1	0.174	0.138	0.152	0.168	26.1%	15.5%	3.6%
	5	0.463	0.387	0.433	0.427	19.6%	6.9%	8.4%
	10	0.683	0.556	0.604	0.619	22.8%	13.1%	10.3%

which the actual developer is a junior developer. The value of *ratio* represents the probability of *EDR_SI* to recommend an effective network. If most recommended networks contain the senior developer(s), we consider *EDR_SI* can help these junior developers find senior developer(s) more easily and conveniently. Then the junior developers can communicate with senior developer(s) to comprehend and resolve software issues.

For **RQ5**, we selected ten students to evaluate the source-code change history for each subject. Each of them randomly read one source code file of each subject in our empirical study. Most of them cannot directly understand the code as they are not the developers of these subjects. *EDR_SI* recommends source-code change history for them to use, and they can choose the experienced developers who changed the file recently and/or changed with the most times. Then, they can communicate with the experienced developer(s), or read relevant commits submitted by the experienced developer(s). If the recommended source-code change history can help them understand the selected code file, 1 score is recorded; if not, 0. Finally, we summarize the scores of each subject.

5.4. Empirical results

5.4.1. RQ1 (*effectiveness of* EDR_SI)

Our study evaluates the accuracy of developer recommendation using the recall metric. The *EDR_SI* column in Table 3 shows the *recall@k* values of *EDR_SI* for developer recommendation on the *jEdit, hadoop, JDT-Debug, Elastic* and *Libgdx* subjects. As expected, the recall values generally increase with the increasing of the *k* value. The *Average* column shows the results of average recall values of the five subjects. From Table 3, we notice that the average values of *recall@1, recall@5*, and *recall@10* are 0.174, 0.463, and 0.683, respectively. That is, on average, *EDR_SI* is able to recommend the correct developer for 17.4%, 46.3%, and 68.3% of issue requests by recommending one, five, and ten developers, respectively.

In addition, the *iMacPro, Location* and *ABA-Time-tf-idf* columns illustrate the recall values of developer recommendation calculated by *iMacPro, Location* and *ABA-Time-tf-idf*, respectively. To compare the results between *EDR_SI* and these three approaches, we can see the columns of "Gain over *iMacPro%*", "Gain over *Location%*" and "Gain over *ABA-Time-tf-idf %*" in Table 3, respectively. From the results in Table 3, we notice that most of *EDR_SI* results are better than *iMacPro, Location* and *ABA-Time-tf-idf*, which ranges from 6.6% to 87.0%, 2.0% to 24.3%, and 0.4% to 20.4%, respectively. Hence, based on the results, we can conclude that the accuracy of *EDR_SI* recommending developers is improved over the state-of-art approaches, i.e., *iMacPro, Location* and *ABA-Time-tf-idf*.

5.4.2. RQ2 (*senior vs. junior*)

We classified 200 issues into four groups according to the actual developers' experience. The more number is the commits submitted by a developer, the more experienced is with the developer. Then we calculate the values of *recall@k* of each group, and compare the values of different groups. Fig. 7 illustrates the recall values of developer recommendation of the four groups. For each line chart in Fig. 7, the *x*-axis represents the number of commits that each developer submitted. The y-axis represents the recall values of each group. From each chart, we notice that the recall value does not increase as the number of commits becomes larger. That is, whether recommending senor developers with more experience or recommending junior developers with less experiences, our approach is both effective. On the other hand, recall values of most groups even decrease while the number of commits becomes larger, which represents that the junior developers are more easily and accurately recommended.

In conclusion, *EDR_SI* can accurately recommend junior developers, which shows that our approach could avoid tending to recommend the senior developers.

5.4.3. RQ3 (EDR_SI *vs. bug localization technique*)

Our study first evaluates the effectiveness of the recommended files with the *ATNF* and *STNF* metrics. Table 4 reports the *ATNF* and *STNF* results of *EDR_SI* and *BRTracer* to recommend 1, 5, and 10 of personalized files. The results show that in most of the cases, *ATNF* and *STNF* values of *EDR_SI* are better than that of *BRTracer*. The "Average" columns show their average values, i.e., the average values of *ATNF*@1,5,10 are 0.28, 0.40, and 0.54, respectively; and the average values of *STNF*@1,5,10 are 0.34, 0.50, and 0.60, respectively. In addition, when we investigate the *ATNF* and *STNF* results respectively, we notice that improvement of *STNF* for *EDR_SI* is large while improvement of *ATNF* is small. As the "Gain over *BRTracer*" column shows, the average gain values of *ATNF*@1,5,10 over *BRTracer* are 17%, 8%, and 8%, respectively; and the average values of *STNF*@1,5,10 are 36%, 108%, and 62%, respectively. That is, *EDR_SI* can recommend more personalized faulty files that are potentially needed to be revised while not decreasing the accuracy of recommended faulty files. The reason for this improvement is that our approach recommends source code files not only considering the accuracy of the buggy files, but also devoting to search files which conform to their development habits (changing times and recent

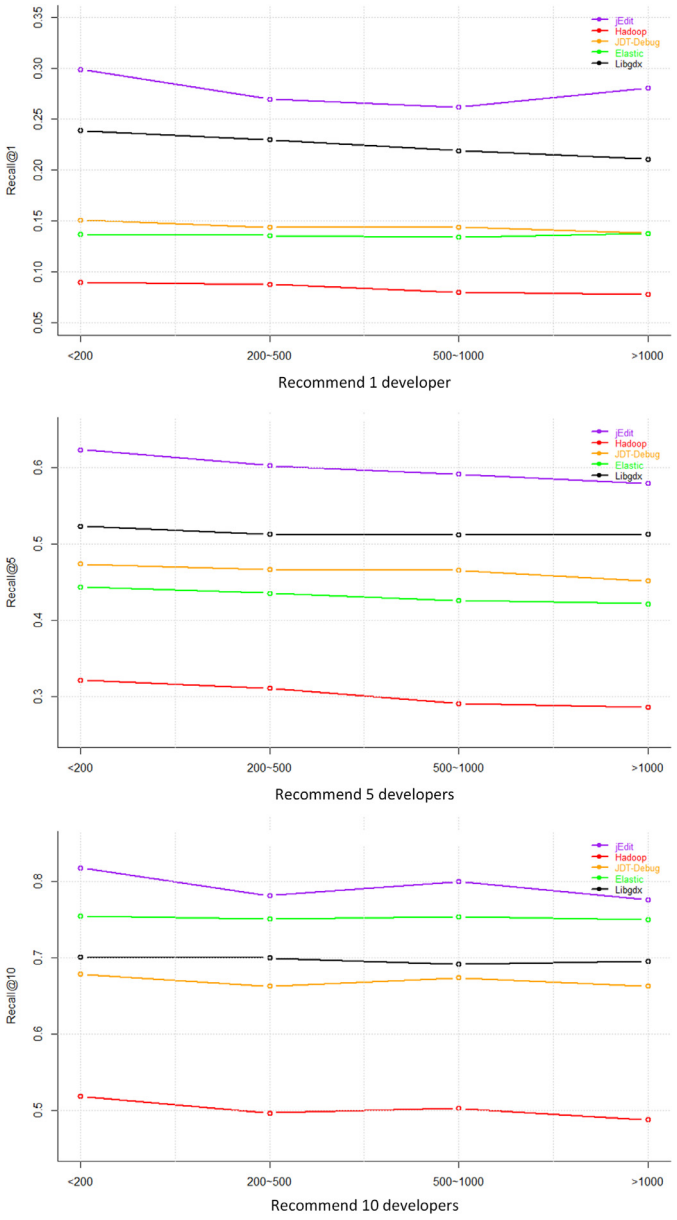

Fig. 7. Recall values of developer recommendation of each group for five subjects.

workday). So the accuracy of the recommended source code files is improved.

In conclusion, *EDR_SI* can recommend more useful and personalized files over the traditional bug location technique, i.e., *BRTracer*.

5.4.4. RQ4 (benefits of developer network)

Our study evaluates how well the effectiveness of *EDR_SI* recommending the network for junior developers is. We calculate the ratio of the effective networks that contain senior developer(s) for the issues whose actual developers are the juniors. Fig. 8 shows the results of the ratio of the effective networks for five subjects, and the values of ratio range from 76% to 82% for the five subjects. The "Average" column shows that the average ratio of the five subjects is 74%. That is, *EDR_SI* is able to recommend an effective developer network for 74% of issue requests on average. In this way, junior developers can easily find the senior developers for consultation to help fix the issues based on the recommended developer network. In conclusion, *EDR_SI* can effectively recommend the developer network for junior developers to use. That is, the developer network can be effectively used to assist junior developers in comprehending and resolving software issues by finding other experienced developers.

Table 4
The *ATNF* and *STNF* results of *EDR_SI* and *BRTracer* to recommend 1,5,10 of personalized files.

System	Top N	*EDR_SI*		*BRTracer*		Gain over *BRTracer*	
		ATNF	*STNF*	*ATNF*	*STNF*	*ATNF*	*STNF*
jEdit	1	0.24	0.42	0.21	0.29	14%	45%
	5	0.41	0.63	0.35	0.37	17%	70%
	10	0.55	0.69	0.53	0.45	4%	53%
Hadoop	1	0.25	0.27	0.22	0.20	14%	35%
	5	0.37	0.51	0.36	0.22	3%	132%
	10	0.50	0.56	0.49	0.29	2%	93%
JDT-debug	1	0.32	0.30	0.25	0.23	28%	30%
	5	0.44	0.52	0.38	0.24	5%	117%
	10	0.58	0.56	0.52	0.36	12%	56%
Elastic	1	0.24	0.40	0.25	0.27	-4%	48%
	5	0.31	0.35	0.30	0.18	3%	94%
	10	0.54	0.56	0.47	0.41	15%	37%
Libgdx	1	0.33	0.31	0.27	0.25	22%	24%
	5	0.46	0.50	0.45	0.19	2%	163%
	10	0.55	0.61	0.52	0.32	6%	91%
Average	1	0.28	0.34	0.24	0.25	17%	36%
	5	0.40	0.50	0.37	0.24	8%	108%
	10	0.54	0.60	0.50	0.37	8%	62%

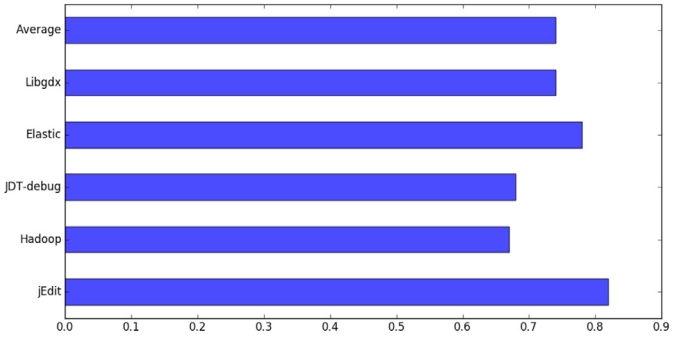

Fig. 8. The ratio of the effective networks that contain the senior developer(s).

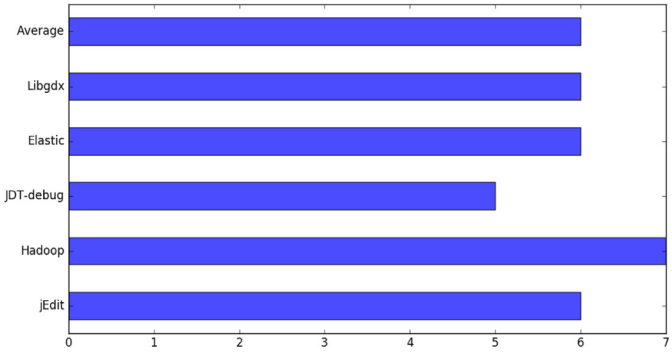

Fig. 9. Scores evaluated by students who have resolved their misunderstandings with the messages of source code change history.

5.4.5. RQ5 (benefits of source-code change history)

We selected 10 students to evaluate the source-code change history. They randomly read one source code file of each subject and find some misunderstandings of the files. So *EDR_SI* recommends source-code change history for them to use, and they can communicate with the relevant experienced developers of the source code file. They can also read some relevant commits to resolve the problems. Fig. 9 shows the scores evaluated by students

for the five subjects in our empirical study. From Fig. 9, the average score is 6 for the five subjects. That is, 60% students considered that source-code change history can effectively help them to understand a specific source code file. In conclusion, the source-code change history can effectively help find other developers to communicate with if some difficulties occur in a specific source file.

6. Threats to validity

In this section, we discuss threats to validity that could influence the results of our empirical study.

The main threat to validity comes from the project selection. In our study, we selected five open-source projects to conduct the study. In addition, the open source software development and maintenance process may be different from that in the industrial process. Thus we cannot guarantee that the results from our empirical study can be generalized to other projects or industrial practice. However, our subjects are selected from different applications, such as the text editing, distributed computing and Java debugging. Moreover, the size of the five open-source projects is relative large.

A second threat is that when we crawled commit repository data with spider programs, the network is not so good that a small number of commit data were not crawled. These missing data may be important to our empirical study. So the empirical study results may be inaccurate with these missing commits.

A third threat is the process of preprocessing the commit content in our study. We used four preprocessing operations to preprocess the textual content. There are still some other preprocessing operations, for example, pruning (Madsen et al., 2004). Different preprocessing operations would generate different similarity results. In addition, in our study, we only implemented the preprocessing techniques based on some typical approaches. For example, for the stemming algorithm, we used *Porter* stemming algorithm. The accuracy of different preprocessing algorithms is different for different data source. After manually checking the results of the stemming algorithm used in our study, we found that some words stemmed are wrong, for example, the word "*updates*" is stemmed to be "*updat*". If the stemmed words are wrong, the following preprocessing operations will also be wrong. For the above "*updat*" word, we cannot find its synonyms via Wordnet. So some other *NLP* techniques may be more suitable for commit data preprocessing (Corazza et al., 2012; Guerrouj et al., 2013).

A fourth threat is that we evaluate the effectiveness of developer network by checking whether senior developer(s) is/are in the network. But some junior developers may be also skilled to the incoming software issue, as they may resolve the relevant issues recently. On the other hand, the senior developers couldn't be skilled in all the software issues. However, the senior developers are generally more skilled than the juniors. So if a network contains the senior(s), it should be an effective developer network in most of the time. For the source-code change history evaluation, 10 bachelor students participated in our studies, but they are not the developers of the five subjects. So it may be more difficult for them to comprehend the source code. But if these supplementary information are effective to them, we believe that it will be better to assist the developer(s) in comprehending and resolving software issues in practice.

A fifth threat is the quality of commit messages which are used to extract developers' experiences for developer recommendation. In practice, developers just put the bug id in the bug fixing commit message, i.e., there is no meaningful text in the commit message. Then, these commit messages may become noise to represent developers' experiences, which can affect the effectiveness of our approach. We also conducted an empirical study to show the effectiveness of applying commit message to recommend developers. The results show that the meaningful words in the commit messages affect the recommendation accuracy (Sun et al., 2017).

The final threat is from the comparative study with the state-of-art techniques. For the developer recommendation comparison, we just selected three typical approaches, i.e., *iMacPro, Location* and *ABA-Time-tf-idf*. We also selected a representative buggy file location approach *BRTracer* for the personalized files recommendation comparison. Moreover, these approaches were studied mainly based on the bug repository. But in our experiment, all of these studies are mainly based on the commit repository. So for each subject, the quality of the data repositories (bug repository and commit repository) may be different, which may influence the evaluating results.

7. Related work

7.1. For developer recommendation

Many approaches have focused on recommending appropriate developers for a particular issue request (Zhang et al., 2016; Yan et al., 2016; Hossen et al., 2014; Kagdi et al., 2012; Anvik and Murphy, 2011; Anvik et al., 2006; Zhang et al., 2014b; 2014a; Wang et al., 2014). Zhang et al. developed an approach, called *KSAP*, to improve automatic developer recommendation by using historical bug reports and heterogeneous network of bug repository (Zhang et al., 2016). Hossen et al. proposed the *iMacPro* approach, which integrates authors and maintainers of relevant source code files, which are change prone, to a given issue request (Hossen et al., 2014). McDonald et al. developed a tool, Expertise Recommender (*ER*), to recommend developers with the desired expertise, which uses a heuristic that considers the most recent modification date when developers modified a specific module (McDonald and Ackerman, 2000). Yan et al. presented a component recommender by using a latent semantic analysis *DPLSA* model. The proposed *DPLSA* model provides a novel method to initialize the word distributions of different topics for developer recommendation (Yan et al., 2016). Minto et al. designed a tool, Emergent Expertise Locator (*EEL*), which mines the history to determine how files were changed together and who committed those changes (Minto and Murphy, 2007). Tamrawi et al. proposed an approach for developer recommendation, which uses fuzzy-sets to model bug-fixing expertise of developers based on the hypothesis that developers who recently fixed bugs are likely to fix them in the near future (Tamrawi et al., 2011). Xia et al. proposed a tool, *DevRec*, which is a composite approach by performing bug report based analysis and developer based analysis for developer recommendation (Xia et al., 2013). Zhang et al. developed a new approach, *BUTTER*, which applies social network analysis to characterize the collaboration between developers (Zhang et al., 2014b). Zhang et al. recommend the most suitable developer for bug resolution, which combines topic model and developer relations (e.g., bug reporter and assignee) to capture developers' interest and experience on specific bug reports (Zhang et al., 2014a). Wang et al. proposes an approach, *FixerCache*, which recommends developers for new bugs based on developers' activeness in components of products with high prediction accuracy and diversity (Wang et al., 2014). Shokripour et al. proposed an approach to recommend developers that uses four information resources (Shokripour et al., 2013), which includes two phases. First, the source code files that will be changed to resolve a new bug report are predicted. Then, the developers for the new report based on information about who has previously fixed faults in the predicted source code files are recommended as candidate developers.

Most of the above approaches tend to recommend experienced developers to accomplish an issue request, i.e., the more experienced is of the developer, the more possible they are recom-

mended. In this article, we focus on enhancing developer recommendation from the perspective of personalized development expertise, which cannot only improve the accuracy of developer recommendation, but also tend to recommend some junior developers if they are suitable to implement the issue request.

7.2. For personalized supplementary information recommendation

Some studies concentrate on locating the faulty files as appendixes for developers to use during the bug analysis process (Rao et al., 2015; Wong et al., 2014; Saha et al., 2013; Xin Ye, 2014; Zhou et al., 2012). Zhou et al. proposed the *BugLocator* approach (Zhou et al., 2012), which ranks all files based on the textual similarity between the initial bug report and the source code. *BugLocator* uses a revised Vector Space Model (*rVSM*), and considers the information about similar bugs that have been fixed before. Saha et al. developed a tool *BLUiR*, which builds on an open source *IR* toolkit (Saha et al., 2013). *BLUiR* requires the source code and bug reports, as well as taking advantage of bug similarity data if they are available for locating the faulty files. Wong et al. proposed an approach named *BRTracer*, which divides each source code files into a series of segments (Wong et al., 2014). And the divided segments can represent this file to match the similarity with the issue request. Then, *BRTracer* analyzes the bug report to identify possible faulty files. Ye et al. designed a learning-to-rank approach, which applies API descriptions to bridge the lexical gap between bug reports and source code (Xin Ye, 2014). They mined useful relationships between a bug report and source code files to locate the faulty files.

Most of the above faulty file recommendation approaches focus on accurately locating the buggy files, which negatively take developers' development experience and habits, moreover, all the faulty files are the same to all the developers who will implement the incoming issue request. So the faulty files are not personalized for each developer. In this article, *EDR_SI* recommends a series of personalized supplementary information considering developers' development experience and habits as appendixes for them to use, and these supplementary information are not only relevant to the issue request, but also associated with developers' expertise.

8. Conclusions and future work

This article proposed a novel approach, *EDR_SI*, which enhances developer recommendation by considering developers' personalized expertise and developing habits. Moreover, *EDR_SI* also provides a series of supplementary information (such as personalized source code files, developer network and source-code change history) for each developer to use, which can help them (especially for the junior developers) improve the quality and efficiency of software issue implementation. We evaluated our approach on five open source systems (*jEdit, JDT-Debug, Hadoop, Elastic* and *Libgdx*). The results show that *EDR_SI* can improve the accuracy of developer recommendation compared to the state-of-art approaches (*iMacPro, Location* and *ABA-Time-tf-idf*). Furthermore, for each recommended developer, *EDR_SI* can recommend useful personalized source code files, developer network and source-code change history by analyzing developers' personalized expertise.

In our work, *EDR_SI* recommends developers as well as supplementary information mainly based on the commit repository. In future work, we will combine other software repositories, i.e., bug repository, stack overflow and communication archives, to further optimize the effectiveness of developer recommendation and provide more supplementary information for developers to use. In addition, *EDR_SI* recommends buggy files considering the developers' experience. However, its accuracy needs further improvement compared to the state-of-art bug location techniques. So we will attempt to further improve the accuracy of recommendation of buggy files, and conduct more comparative studies to show its effectiveness in our future work. In our approach, *EDR_SI* can help recommend junior developers, however, in the maintenance cycle, junior developers may become senior developers, but they will be able to more quickly fix a task similar to the one previously completed. So we will consider more factors to recommend developers, for example, modeling the evolution of their historical development topics based on the technique proposed by Hu et al. (2015).

Acknowledgments

This work is supported partially by Natural Science Foundation of China under Grant No. 61402396 and No. 61472344, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2016B21, partially by the Jiangsu Qin Lan Project, partially by the China Postdoctoral Science Foundation under Grant No. 2015M571489, and partially by the Natural Science Foundation of Yangzhou City.

References

Anvik, J., Hiew, L., Murphy, G.C., 2006. Who should fix this bug? In: 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20–28, 2006, pp. 361–370.

Anvik, J., Murphy, G.C., 2011. Reducing the effort of bug report triage: recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. 20 (3), 10.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent dirichlet allocation. J. Mach. Learn. Res. 3, 993–1022.

Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M., 2014. How changes affect software entropy: an empirical study. Empir. Softw. Eng. 19 (1), 1–38.

Chen, C., Zheng, X., Wang, Y., Hong, F., Lin, Z., 2014. Context-aware collaborative topic regression with social matrix factorization for recommender systems. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27, -31, 2014, Québec City, Québec, Canada., pp. 9–15.

Corazza, A., Martino, S.D., Maggio, V., 2012. Linsen: an efficient approach to split identifiers and expand abbreviations. In: 28th IEEE International Conference on Software Maintenance, pp. 233–242.

Guerrouj, L., Penta, M.D., Antoniol, G., Guéhéneuc, Y.-G., 2013. Tidier: an identifier splitting approach using speech recognition techniques. J. Softw. 25 (6), 575–599.

Hossen, K., Kagdi, H.H., Poshyvanyk, D., 2014. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In: 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2–3, 2014, pp. 130–141.

Hu, J., Sun, X., Lo, D., Li, B., 2015. Modeling the evolution of development topics using dynamic topic models. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2–6, 2015, pp. 3–12. doi:10.1109/SANER.2015.7081810.

Hu, Y., Koren, Y., Volinsky, C., 2008. Collaborative filtering for implicit feedback datasets. In: In IEEE International Conference on Data Mining (ICDM) 2008, pp. 263–272.

Kagdi, H.H., Collard, M.L., Maletic, J.I., 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Maint. 19 (2), 77–131.

Kagdi, H.H., Gethers, M., Poshyvanyk, D., Hammad, M., 2012. Assigning change requests to software developers. J. Softw. Maint. 24 (1), 3–33.

Kagdi, H.H., Poshyvanyk, D., 2009. Who can help me with this change request? In: The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17–19, 2009, pp. 273–277. doi:10.1109/ICPC.2009.5090056.

Kang, J., Lerman, K., 2013. LA-CTR: a limited attention collaborative topic regression for social media. CoRR. abs/1311.1247. http://arxiv.org/abs/1311.1247.

Madsen, R.E., Sigurdsson, S., Hansen, L.K., Larsen, J., 2004. Pruning the vocabulary for better context recognition.. In: International Conference on Pattern Recognition, pp. 483–488.

McDonald, D.W., Ackerman, M.S., 2000. Expertise recommender: a flexible recommendation system and architecture.. In: Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work, pp. 231–240.

Minto, S., Murphy, G.C., 2007. Recommending emergent teams. In: Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19–20, 2007, Proceedings, p. 5.

Porter, M., 2006. An algorithm for suffix stripping, pp. 211–218.

Purushotham, S., Liu, Y., 2012. Collaborative topic regression with social matrix factorization for recommendation systems. In: Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012.

Rajlich, V., 2014. Software evolution and maintenance. In: Proceedings of the on Future of Software Engineering, pp. 133–144.

Rao, S., Medeiros, H., Kak, A., 2015. Comparing incremental latent semantic analysis algorithms for efficient retrieval from software libraries for bug localization. ACM SIGSOFT Softw. Eng. Notes 40 (1), 1–8.

Saha, R., Lease, M., Khurshid, S., Perry, D., 2013. Improving bug localization using structured information retrieval. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 345–355. doi:10.1109/ASE.2013.6693093.

Servant, F., Jones, J.A., 2012. Whosefault: automatic developer-to-fault assignment through fault localization. In: Proceedings - International Conference on Software Engineering, pp. 36–46.

Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S., 2013. Why so complicated? Simple term filtering and weighting for location-based bug report assignment recommendation. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18–19, 2013, pp. 2–11.

Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S., 2015. A time-based approach to automatic bug report assignment. J. Syst. Softw. 102, 109–122. doi:10.1016/j.jss.2014.12.049.

Sun, X., Li, B., Leung, H.K.N., Li, B., Li, Y., 2015. MSR4SM: using topic models to effectively mining software repositories for software maintenance tasks. Inf. Softw. Technol. 66, 1–12. doi:10.1016/j.infsof.2015.05.003.

Sun, X., Li, B., Li, Y., Chen, Y., 2015. What information in software historical repositories do we need to support software maintenance tasks? An approach based on topic model. In: Computer and Information Science, pp. 27–37. doi:10.1007/978-3-319-10509-3_3.

Sun, X., Liu, X., Hu, J., Zhu, J., 2014. Empirical studies on the nlp techniques for source code data preprocessing. In: Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies, pp. 32–39.

Sun, X., Liu, X., Li, B., Duan, Y., Yang, H., Hu, J., 2016. Exploring topic models in software engineering data analysis: a survey. In: 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2016, Shanghai, China, May 30, - June 1, 2016, pp. 357–362. doi:10.1109/SNPD.2016.7515925.

Sun, X., Yang, H., LEUNG, H., Li, B., LI, H.J., 2017. On the effectiveness of exploring historical commits for developer recommendation: an empirical study. Front. Comput. Sci. doi:10.1007/s11704-016-6023-3.

Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N., 2011. Fuzzy set and cache-based approach for bug triaging. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5–9, 2011, pp. 365–375.

Wang, C., Blei, D.M., 2011. Collaborative topic modeling for recommending scientific articles. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21–24, 2011, pp. 448–456.

Wang, H., Chen, B., Li, W., 2013. Collaborative topic regression with social regularization for tag recommendation. In: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3–9, 2013.

Wang, S., Lo, D., 2014. Version history, similar report, and structure: putting them together for improved bug localization. In: 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2–3, 2014, pp. 53–63.

Wang, S., Zhang, W., Wang, Q., 2014. Fixercache: unsupervised caching active developers for diverse bug triage. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, New York, NY, USA, pp. 25:1–25:10. doi:10.1145/2652524.2652536.

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 181–190. doi:10.1109/ICSME.2014.40.

Xia, X., Lo, D., Wang, X., Zhou, B., 2013. Accurate developer recommendation for bug resolution. In: 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14–17, 2013, pp. 72–81.

Xia, X., Lo, D., Wang, X., Zhou, B., 2015. Dual analysis for recommending developers to resolve bugs. J. Softw. 27 (3), 195–220. doi:10.1002/smr.1706.

Xin Ye, R. B.,. Learning to rank relevant files for bug reports using domain knowledge, 201410.1145/2635868.2635874.

Yan, M., Zhang, X., Yang, D., Xu, L., Kymer, J.D., 2016. A component recommender for bug reports using discriminative probability latent semantic analysis. Inf. Softw. Technol. 73, 37–51.

Yang, H., Sun, X., Bin Li and, Y.D., 2016. DR_PSF: enhancing developer recommendation by leveraging personalized source-code files. In: The 40th IEEE Computer Society International Conference on Computers, Software and Applications, pp. 239–244.

Yang, H., Sun, X., Duan, Y., Li, B., 2016. On the effects of exploring historical commit messages for developer recommendation. Chin. J. Electron. 25 (4).

Yang, H., Sun, X., Li, B., Hu, J., 2016. Recommending developers with supplementary information for issue request resolution. In: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, pp. 707–709.

Zanjani, M.B., Swartzendruber, G., Kagdi, H., 2014. Impact analysis of change requests on source code based on interaction and commit histories. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 162–171.

Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L., Mei, H., 2015. A survey on bug-report analysis. Sci. Chin. Inf. Sci. 58 (2), 1–24. doi:10.1007/s11432-014-5241-2.

Zhang, T., Lee, B., 2013. A hybrid bug triage algorithm for developer recommendation. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1088–1094.

Zhang, T., Yang, G., Lee, B., Lua, E.K., 2014. A novel developer ranking algorithm for automatic bug triage using topic model and developer relations. In: Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference - Volume 01. IEEE Computer Society, Washington, DC, USA, pp. 223–230. doi:10.1109/APSEC.2014.43.

Zhang, W., Han, G., Wang, Q., 2014. Butter: An approach to bug triage with topic modeling and heterogeneous network analysis. In: Proceedings of the 2014 International Conference on Cloud Computing and Big Data. IEEE Computer Society, Washington, DC, USA, pp. 62–69. doi:10.1109/CCBD.2014.14.

Zhang, W., Wang, S., Wang, Q., 2016. Ksap: an approach to bug report assignment using knn search and heterogeneous proximity. Inf. Softw. Technol. 70, 68–84.

Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? - More accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 14–24.

Sun Xiaobing is an associate professor in School of Information Engineering, Yangzhou University. His current research interests include change comprehension, analysis and testing, software data analytics. (Email: xbsun@yzu.edu.cn)

Yang Hui is a student in School of Information Engineering, Yangzhou University. His current research interest is recommendation systems for software maintenance.

Xia Xin is a Postdoctoral Research Fellow in Department of Computer Science, University of British Columbia. His current research interests include software engineering, software data analytics.

Li Bin is a professor in School of Information Engineering, Yangzhou University. His current research interests include web service analysis, cloud computing.