# What Information in Software Historical Repositories Do We Need to Support Software Maintenance Tasks? An Approach Based on Topic Model

**Xiaobing Sun, Bin Li, Yun Li and Ying Chen**

**Abstract** Mining software historical repositories (SHR) has emerged as a research direction Sun, over the past decade, which achieved substantial success in both research and practice to support various software maintenance tasks. Use of different types of SHR, or even different versions of the software project may derive different results for the same technique or approach of a maintenance task. Inclusion of unrelated information in SHR-based technique may lead to decreased effectiveness or even wrong results. To the best of our knowledge, few focus is on this respect in the SE community. This paper attempts to bridge this gap and proposes a preprocess to facilitate selection of related SHR to support various software maintenance tasks. The preprocess uses the topic model to extract the related information from SHR to help support software maintenance, thus improving the effectiveness of traditional SHR-based technique. Empirical results show the effectiveness of our approach.

**Keywords** Software historical repositories · Topic model · Information retrieval · Software maintenance

## 1 Introduction

Software maintenance has been recognized as the most difficult, costly and labor-intensive activity in the software development life cycle [21]. Effectively supporting software maintenance is essential to provide a reliable and high-quality evolution

X. Sun (✉) · B. Li · Y. Li · Y. Chen
School of Information Engineering, Yangzhou University, Yangzhou, China
e-mail: xbsun@yzu.edu.cn

B. Li
e-mail: lb@yzu.edu.cn

Y. Li
e-mail: liyun@yzu.edu.cn

Y. Chen
e-mail: chenying@yzu.edu.cn

R. Lee (ed.), *Computer and Information Science*, Studies in Computational Intelligence 566, DOI 10.1007/978-3-319-10509-3_3

of software systems. However, the complexity of source code tends to increase as it evolves. Recently, the field of software engineering (SE) focused on this field by mining the repositories related to a software project, for example, source code changes, bug repository, communication archives, deployment logs, execution logs [12]. Among these information, software historical repositories (*SHR*) such as source control repositories, bug repositories, and archived communications record information about the evolution and progress of a project. The information in *SHR* have been analyzed to support various software maintenance tasks such as impact analysis, bug prediction, software measures and metrics, and other fields [2, 12, 16, 23, 27]. All these studies have shown that interesting and practical results can be obtained from these historical repositories, thus allowing maintainers or managers to better support software evolution and ultimately increase its quality.

The mining software repositories field analyzes and explores the rich data available in *SHR* to uncover interesting and actionable information about software systems and projects [12]. The generated information can then be used to support various software maintenance tasks. These software maintenance tasks are traditional *SHR*-based techniques, which directly used the information in *SHR* in support of these software maintenance tasks without any filtering process. In practice, use of different types of software repositories, or even different versions of the software project may derive different results for the same technique or approach of a maintenance task [12, 13]. Sometimes, appropriate selection of the information in software repositories may obtain expected effectiveness for some techniques. However, inclusion of unrelated information in *SHR* for practical analysis may lead to decreased effectiveness. Hence, the main research question comes out:

"**What information in *SHR* should be included to support software maintenance tasks**?"

To the best of our knowledge, there is still few work to address this issue. In this paper, we focus on this respect, and attempt to facilitate selection of related *SHR* to support various software maintenance tasks. In the software repositories, the data can be viewed as unstructured text. And topic model is one of the popular ways to analyze unstructured text in other domains such as social sciences and computer vision [4, 15], which aims to uncover relationships between words and documents. Here, we proposed a preprocess before directly using *SHR*, which uses the topic model to help select the related information from *SHR*. After the preprocess, the effectiveness of traditional *SHR*-based techniques for software maintenance tasks is expected to be improved.

The rest of the paper is organized as follows: in the next section, we introduce the background of *SHR* and the topic model. Section 3 presents our approach to select the necessary information from *SHR*. In Sect. 4, empirical evaluation is conducted to show the effectiveness of our approach. Finally, we conclude and show some future work in Sect. 5.

## 2 Background

As our approach is to use topic model to facilitate selection of appropriate information from *SHR* to support software maintenance tasks. In this section, we introduce some background about *SHR* and the topic model.

### *2.1 Software Historical Repositories*

Mining software repositories (MSR) has emerged as a research direction over the past decade, which has achieved great success in both research and practice [12]. Software repositories contain a wealth of valuable information about software projects such as historical repositories, runtime repositories, and code repositories [12]. Among these repositories, software historical repositories (*SHR*) record information about the evolution and progress of a project. *SHR* collect important historical dependencies between various software artifacts, such as functions, documentation files, and configuration files. When performing software maintenance tasks, software practitioners can depend less on their intuition and experience, and depend more on historical data. For example, developers can use this information to propagate changes to related artifacts, instead of using only static or dynamic program dependencies, which may fail to capture important evolutionary and process dependencies [11].

*SHR* mainly include source control repositories, bug repositories, communication archives. A description of these repositories is shown in Table 1. The amount of these information will become lager and larger as software evolves. These information

**Table 1** Software historical repositories

| Software historical repositories | Description |
|---|---|
| Source control repositories | These repositories record the information of the development history of a project. They track all the changes to the source code along with the meta-data associated with each change, for example, who and when performed the change and a short message describing the change. CVS and subversion belong to these repositories. |
| Bug repositories | These repositories track the resolution history of bug reports or feature requests that are reported by users and developers of the projects. Bugzilla is an example of this type of repositories. |
| Communication archives | These repositories track discussions and communications about various aspects of the project over its lifetime. Mailing lists and emails belong to the communication archives. |

can be used to support various software maintenance tasks, such as change impact analysis, traceability recovery [12, 14]. These traditional *SHR*-based techniques directly used the information in the *SHR* to perform software maintenance. However, with the evolution of software, some information may be outdated. Therefore, not all historical information are useful to support software maintenance. In this paper, we focus on selection of useful or related information from *SHR* to support practical software maintenance tasks.

### 2.2 Topic Model

As information stored in *SHR* is mostly unstructured text, researchers have proposed various ways to process such unstructured information. An increasingly popular way is to use topic models, which focus on uncovering relationships between words and documents [1]. Topic models were originated from the field of natural language processing and information retrieval to index, search, and cluster a large amount of unstructured and unlabeled documents [25]. A topic is a collection of terms that co-occur frequently in the documents of the corpus. The most used topic models in software engineering community are Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA) [25]. These two topic models have been applied to support various software engineering tasks: feature location, change impact analysis, bug localization , and many others [17, 19, 25]. The topic models require no training data, and can well scale to thousands or millions of documents. Moreover, they are completely automated.

Among the two topic models, LDA is becoming increasing popular because it models each document as a mixture of $K$ corpus-wide topics, and each topic as a mixture of the terms in the corpus [5]. More specifically, it means that there are a set of topics to describe the entire corpus, each document can contain more than one of these topics, and each term in the entire repository can be contained in more than one of these topic. Hence, *LDA* is able to discover a set of ideas or themes that well describe the entire corpus. *LDA* is a probabilistic statistical model that estimates distributions of latent topics from textual documents [5]. It assumes that the documents have been generated using the probability distribution of the topics, and that the words in the documents were generated probabilistically in a similar way [5]. With *LDA*, some latent topics can be mined, allowing us to cluster them on the basis of their shared topics. In this paper, we use *LDA* to extract the latent topics from various software artifacts in *SHR*.

## 3 Approach

Our main focus in this paper is to provide an effective way to automatically extract related information from the *SHR* to provide support of software maintenance tasks. The process of our approach is shown in Fig. 1, which can be seen as a preprocess to traditional *SHR*-based techniques. Given a maintenance task or request, we need some related information in *SHR* to well support comprehension, analysis and implementation of this request. The data source of our approach includes a maintenance request, *SHR* and current software. We extract current software from *SHR* here because the current software usually needs some necessary changes to accomplish this change request. As the data source can be seen as unstructured text, we use *LDA* on these data source to extract the latent topics in them. Then, we compare the similarity among the topics extracted from different data source, and ultimately produce the related software repositories which are related to the maintenance request and current software. For example, in Fig. 1, when there exists a bug repository which has similar topics with the software maintenance request, we can consider this bug repository as related data source to analyze the current software maintenance request. In addition, there is also a *feedback* from this bug repository to its corresponding version repository in source control repositories. More specifically, the corresponding version evolution corresponding to this bug repository is also considered to be a useful data source for analyzing the request in the current software. Hence, the extracted related information from *SHR* includes the related bug repository, useful communication archive, and some evolutionary code versions related to this maintenance request and the current software.

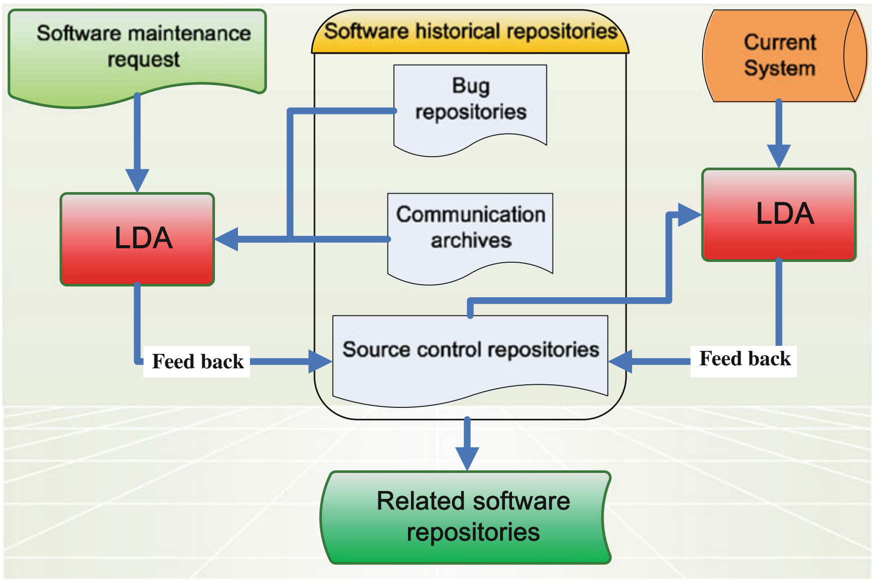

Software maintenance request
Software historical repositories
Current System
Bug repositories
LDA
Communication archives
LDA
Feed back
Source control repositories
Feed back
Related software repositories

**Fig. 1** Process of our approach

## *3.1 Extracting Related Information from Bug Repositories and Communication Archives*

First, we extract the related information from bug repositories and communication archives to see what information is related to the maintenance request. We use the *LDA* model to extract the latent topics in these data source, i.e., maintenance request, bug repositories, and communication archives. The topics extracted from these data source can be represented with a vector (for maintenance request) and matrices (for bug repositories and communication archives). Then, we compute the similarity between the vector and these two matrices, respectively. There are many approaches to compute the similarity result, for example, distance measures, correlation coefficients, and association coefficients [22]. Any similarity measure can be used in our approach. According to the similarity results, the bug repositories and communication archives which are similar to the maintenance request can be extracted as useful data source. These extracted information can be helpful to guide the change analysis. For example, when the maintenance request is to fix a bug and if there is an existing bug report similar to this bug fixing request, we can use the information in related bug repositories and communication archives to see "who fixes the bug?", "how to fix the bug?", etc.

## *3.2 Extracting Related Information from Source Control Repositories*

Source control repositories, a fundamental unit of software evolution, are important data source in software maintenance research and practice. Source control repositories include three main different types of information, that is, the software versions, the differences between the versions, and metadata about the software change [12]. These different types of information have been used in both research and practice to well support various software maintenance tasks such as impact analysis. However, most of these research and practice seldom consider the usefulness of the information in source control repositories. As software evolves, some information may be outdated and become awful, or even "*noise*" to current software analysis. Hence, we need to extract the related and necessary versions in *SHR* to support software maintenance tasks.

To fully obtain the related information from source control repositories, we perform this step in two ways. First, we select the versions corresponding to the bug repositories and communication archives from the previous step, where we have obtained the bug repositories and communication archives related to the maintenance request. In addition, we can also obtain the **consecutive** source code versions corresponding to the bug repositories and communication archives.[1] Thus, this kind of source code versions is produced based on the maintenance request. Second, we extract the source versions similar to current software. We extract the latent topics from current software and represent them as a vector. Then, we compute the similarity result between the current software vector and the matrices (for source control repositories). Here, we also extract consecutive versions which are related to the current software.

Until here, all the related information have been extracted from various *SHR*. We believe that such a preprocess on [12] can effectively improve the effectiveness of traditional software maintenance activities which are not preprocessed or filtered.

## 4 Evaluation

Our approach aims to improve traditional *SHR*-based software maintenance techniques after filtering the unrelated information without decreasing the completeness of the results.

### *4.1 Empirical Setup*

We conduct our evaluation on an existing benchmark built by Poshyvanyk et al.[2] We select three Java subject programs from open projects for our studies. The first subject program is *ArgoUML*, which is an open source UML modeling tool that supports standard UML 1.4 diagrams. The second subject is *JabRef*, which is a graphical application for managing bibliographical databases. The final subject is *jEdit*, which is a text editor written in Java. For each subject program, we used four concecutive versions ($V0 \rightarrow V3$) for study. Then, we compare the effectiveness of our approach with traditional software maintenance techniques on this benchmark. Here, we use change impact analysis as the representation of one of the software maintenance techniques for study [14]. Change impact analysis is a technique used to identify the potential effects caused by software changes [6, 14]. In our study, we employ *ROSE* (Reengineering of Software Evolution) tool to represent the *SHR*-based CIA [28]. *ROSE* is shown to be effective for change impact prediction [24], and it applies data mining to version histories to guide impact analysis. The input of *ROSE* can be coarser file or class level changes, or finer method-level changes. Its output is the corresponding likely impacted entities at the same granularity level as the change set. In this paper, the chosen granularity level is class level.

[1] Here we need consecutive versions since many software maintenance tasks are performed based on the differences between consecutive versions in source control repositories.

[2] http://www.cs.wm.edu/semeru/data/msr13/.

To show the effectiveness of CIA, we used *precision* and *recall*, two widely used metrics of information retrieval [26], to validate the *accuracy* of the CIA techniques. They are defined as follows:

$$P = \frac{|Actual \quad Set \quad \cap \quad Estimated \quad Set|}{|Estimated \quad Set|} \times 100\,\%$$

$$R = \frac{|Actual \quad Set \quad \cap \quad Estimated \quad Set|}{|Actual \quad Set|} \times 100\,\%$$

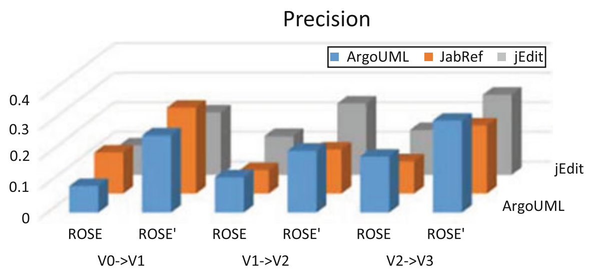
*Actual Set* is the set of classes which are really changed to fix the bugs for the system. *Estimated Set* is the set of classes potentially impacted by the *change set* based on *ROSE*. The change set is composed of a set of classes used to fix each bug. The change set is mined from their software repositories. Specific details on the process of the identification of the bug reports and changed classes can refer to [20]. With the change set, we applied *ROSE* to compute the *Estimated Set*. Then, *precision* and *recall* values are computed based on the *Actual Set* and *Estimated Set*. Here, *ROSE* is performed twice, one is on the original software historical repositories, i.e., *ROSE*; the other is on the extracted repositories preprocessed by our approach, i.e., *ROSE*′.

## 4.2 Empirical Results

We first see the precision results of *ROSE* before (*ROSE*) and after (*ROSE*′) software historical data filtering. The results are shown in Fig. 2. From the results, we see that all the precision values of *ROSE* are improved after software historical data filtering. It shows that there is indeed some unrelated information in software historical repositories for software maintenance and evolution. Hence, we should filter these unrelated software historical information to improve the precision of change impact analysis.

In addition, during the process of filtering the software historical data, there may be some related information which is filtered by our approach, so we need to see

**Fig. 2** Precision of ROSE before and after information filtering



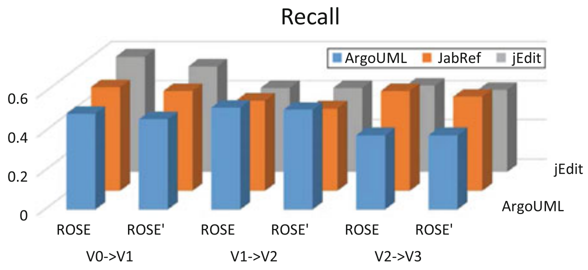


**Fig. 3** Recall of ROSE before and after information filtering

whether the recall of *ROSE* is seriously decreased after information filtering. The recall results are shown in Fig. 3. It shows that most of the recall values are decreased after information filtering process. However, the degree of the deceasing values is not big. Hence, we can obtain that only a small amount of related information is filtered during the process, which has few effect on software maintenance and evolution.

## 5 Conclusion and Future Work

This paper proposed a novel approach which can improve the effectiveness of traditional *SHR*-based software maintenance tasks. Our approach extracted related information from *SHR* using the topic model, i.e., LDA. The generated software repositories can eliminate the information that are outdated during software evolution, thus improving the effectiveness of the traditional *SHR*-based software maintenance tasks. Finally, this paper evaluated the proposed technique and showed its effectiveness.

Though we have shown the effectiveness of our approach through real case studies based on *ROSE*, it can not indicate its generality for other real environment. And we will conduct experiments on more real programs to evaluate the generality of our approach. In addition, we would like to study whether the effectiveness of other software maintenance tasks (e.g., feature location [8, 9], bad smell detection [10, 18], traceability recovery [3, 7], etc.) can be improved based on the *SHR* preprocessed by our approach.

**Acknowledgments** The authors would like to thank anonymous reviewers who make the paper more understandable and stronger. This work is supported partially by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant No. 13KJB520027, partially by National Natural Science Foundation of China under Grant No. 61402396 and No. 61472344, partially by the Innovative Fund for Industry-Academia-Research Cooperation of Jiangsu Province under Grant No. BY2013063-10, and partially by the Cultivating Fund for Science and Technology Innovation of Yangzhou University under Grant No. 2013CXJ025.

## References

1. Anthes, G.: Topic models versus unstructured data. Commun. ACM **53**(12), 16–18 (2010)
2. Antoniol, G., Huffman Hayes, J., Gaël Guéhéneuc, Y., Di Penta, M.: Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date. In: 24th IEEE International Conference on Software Maintenance, pp. 147–156 (2008)
3. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Software Eng. **28**(10), 970–983 (2002)
4. Barnard, K., Duygulu, P., Forsyth, D.A., de Freitas, N., Blei, D.M., Jordan, M.I.: Matching words and pictures. J. Mach. Learn. Res. **3**, 1107–1135 (2003)
5. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
6. Bohner, S., Arnold, R.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
7. De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S.: Applying a smoothing filter to improve ir-based traceability recovery processes: An empirical investigation. Inf. Softw. Technol. **55**(4), 741–754 (2013)
8. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Evol. Process **25**(1), 53–95 (2013)
9. Dit, B., Revelle, M., Poshyvanyk, D.: Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empir. Softw. Eng. **18**(2), 277–309 (2013)
10. Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: an experimental assessment. J. Object Technol. **11**(2), 5:1–5:38 (2012)
11. Hassan, A.E., Holt, R.C.: Predicting change propagation in software systems. In: 20th International Conference on Software Maintenance, pp. 284–293 (2004)
12. Kagdi, H.H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Maintenance **19**(2), 77–131 (2007)
13. Kagdi, H.H., Gethers, M., Poshyvanyk, D.: Integrating conceptual and logical couplings for change impact analysis in software. Empir. Softw. Eng. **18**(5), 933–969 (2013)
14. Li, B., Sun, X., Leung, H., Zhang, S.: A survey of code-based change impact analysis techniques. Softw. Test., Verif. Reliab. **23**(8), 613–646 (2013)
15. Li, D., Ding, Y., Shuai, X., Bollen, J., Tang, J., Chen, S., Zhu, J., Rocha, G.: Adding community and dynamic to topic models. J. Informetrics **6**(2), 237–253 (2012)
16. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and mozilla. ACM Trans. Softw. Eng. Methodol. **11**(3), 309–346 (2002)
17. Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N.: A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 263–272 (2011)
18. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: Detecting bad smells in source code using change history information. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 268–278 (2013)
19. Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: 35th International Conference on Software Engineering, pp. 522–531 (2013)
20. Poshyvanyk, D., Marcus, A., Ferenc, R., Gyimothy, T.: Using information retrieval based coupling measures for impact analysis. Empir. Softw. Eng. **14**(1), 5–32 (2009)
21. Schneidewind, N.F.: The state of software maintenance. IEEE Trans. Softw. Eng. **13**(3), 303–310 (1987)
22. Shtern, M., Tzerpos, V.: Clustering methodologies for software engineering. Adv. Softw. Eng. **2012**, 18. doi:10.1155/2012/792024 (2012)

23. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005)
24. Sun, X., Li, B., Li, B., Wen, W.: A comparative study of static cia techniques. In: Proceedings of the Fourth Asia-Pacific Symposium on Internetware, pp. 23 (2012)
25. Thomas, S.W.: Mining software repositories using topic models. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1138–1139 (2011)
26. van Rijsbergen, C.J.: Information Retrieval. Butterworths, London (1979)
27. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Trans. Softw. Eng. **31**(6), 429–445 (2005)
28. Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S.: Mining version histories to guide software changes. IEEE Trans. Softw. Eng. **31**(6), 429–445 (2005)