Is It a New Feature or Simply "Don't know yet"?
On Automated Redundant OSS Feature Requests Identification

Lin Shi *, Celia Chen†, Qing Wang*, and Barry Boehm†
*Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences, Beijing, China
Email: {shilin,wq}@itechs.iscas.ac.cn
†Center for Systems and Software Engineering, University of Southern California, Los Angeles, USA
Email:{qianqiac,boehm}@usc.edu

***Abstract*—Open source projects rely on issue tracking systems such as JIRA or online forums to keep track of users' feedback, expectations and requested features. However, since users are not fully aware of existing features, when submitting new feature requests, redundant requests often appear in the new feature list. It is a waste of time and effort for project contributors to manually identify and reject them, especially in complex systems with many features. Our research is aiming to find a suitable solution to identify redundant feature requests in OSS projects. We have conducted a survey on a well-known Open Source community, Hibernate and gathered all of its feature requests up-to-date. Through studying and categorizing the characteristics of these feature requests, we have found that about 37% of the feature requests were rejected and the most common rejection reason was redundancy. Also we have found that it is very expensive to identify and resolve these redundant feature requests. In this paper, we have proposed our solution to automatically identify redundant feature requests through a Feature Tree Model along with a future research agenda.**

I. Introduction

Online tracking systems and open forums, such as Bugzilla[1], JIRA[2], and SourceForge[3], are widely used by open source projects as important channels to collect new requirements (often known as feature requests) proposed by users.

When a new feature request has been posted, project contributors will create a discussion thread on the forum to negotiate with reporters and decide whether this feature request should be implemented [1]. The common factors for feature request acceptance include benefit analysis, feature attractiveness, conflicts with current system, feasibility, *et al*. This process is considered as OSS requirements analysis [2].

There are two key challenges in the practice of OSS requirements analysis.

One challenge is that it is difficult for users to submit well-thought-out feature requests due to their lack of comprehensive understanding of all the features in the current system. OSS projects often don't provide any systems requirements specifications in one document. Instead, information about existing features can be found in various sources. For example, key features are often presented in project website; newly added features are often described in release notes; detailed features can be found in user manual; and descriptions about unreleased features can be found in pull requests [3]. As the system evolves, those documentation will also grow in size [4]. When there is a large volume of documentation, users are less willing to carefully read through them to learn about existing features of the system [5]. In this paper, we refer to this kind of feature requests as *redundant feature requests*. As a result, users often submit feature requests that describe existing features in the system. For example, there are about 42 redundant feature requests posted per month to Eclipse community from January to May in 2016, which consumed a large extra effort of OSS developers on discussing and terminating such requests.

Another challenge is that when users submit low quality feature requests, it is a waste of time for project contributors to conduct OSS requirements analysis [6]. For example, when users submit redundant feature requests or questions on how to use the system to the forum, conducting such discussion and analysis on those requests is a waste of time and effort for project contributors.

An automated method that can identify redundant feature requests will alleviate the effort from project contributors and improve the utility of current open forums. The main objective of our work is to provide an automated support for identifying redundant feature requests. We consider redundant feature requests as a significant source of noisy feature requests. In this paper, we highlight the importance of identifying such feature requests through conducting a survey on a well-known open source community. We then propose a road map towards an automated approach to identify redundant feature requests, which will be applied and evaluated with OSS projects in practice.

The rest of this paper is organized like the following: Section II elaborates the motivation and our empirical analysis. Section III presents the framework of our solution. Section IV describes the research agenda in the next future. Section V introduces the related work. Section VI concludes our work.

II. Motivation and Empirical Analysis

To better understand the issues and challenges of OSS requirements analysis, we conduct a survey on Hibernate community. Hibernate is an open source Java persistence framework project, which started in 2001. The reasons we select Hibernate community[4] as our survey target are manifold.

[1]http://www.bugzilla.org
[2]http://www.atlassian.com
[3]http://www.sourceforge.net
[4]http://hibernate.org/

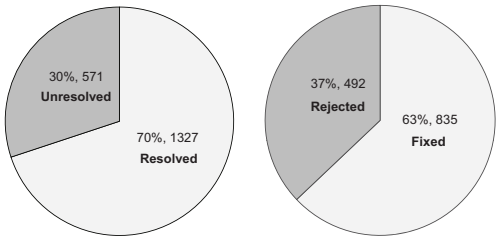
2332-6441/16 $31.00 © 2016 IEEE
DOI 10.1109/RE.2016.65

RE 2016, Beijing, China
RE@Next! Paper

Hibernate community is one of the first open source community. Throughout the years, the projects remain popular among developers and the community stays active. The source code is hosted and maintained on Github[5] while feature requests including new feature requests are only managed on JIRA[6] issue tracking system. There are 20 projects listed in the JIRA forum. 9 out of 20 projects have short history and have received less than 20 feature requests in total. We exclude those projects and analyzed the feature requests in the rest 11 projects, including Hibernate ORM, Hibernate Search, Hibernate Validator and *et al.*

We retrieve all feature requests in the 11 projects from the beginning until now. There are 1898 feature requests in total. We consider feature requests with label *UNRESOLVED* as *unresolved feature requests* while other feature requests are considered as *resolved feature requests*. 30% (571 out of 1898) feature requests are unresolved and 70% (1327 out of 1898) are resolved, as shown in Figure 1(a).

Among the resolved feature requests, we consider feature requests with label *FIXED* as *fixed feature requests*, while others are *rejected feature requests*. 63% (835 out of 1327) feature requests are fixed and 37% (492 out of 1327) are rejected, as shown in Figure1(b).



(a) All feature requests (b) Resolved feature requests

Fig. 1. Status of feature requests in Hibernate community

We notice that there is a relatively high percentage (37%) of resolved feature requests that were rejected by Hibernate core contributors. We further investigate the reasons of those feature requests rejections in the following section.

A. Understand why feature requests get rejected

In order to understand why those 492 feature requests got rejected, we analyze the comments of each rejected feature request. Contributors typically provide explicit explanation in the comments before they reject any feature requests. After going through the discussion threads of 492 rejected feature requests, we summarize the reasons in Table I.

Redundant. 23.37% of the rejected feature requests got rejected due to the proposed features are already existed in the current system. Existing research [5] shows that people are often unwilling to read user manual carefully. If users miss certain features they desire while reading the documentation, they often submit those feature requests to open forum and suggest them to be implemented in major or critical priority.

[5]https://github.com/hibernate

[6]https://hibernate.atlassian.net

TABLE I
THE REASONS WHY FEATURE REQUESTS GOT REJECTED

Reasons	Occurrence	Percentage
Redundant	115	23.37%
Duplicate	93	18.90%
Dead threads	42	8.54%
Merged into other threads	38	7.72%
Developers don't like	30	6.10%
Usage questions	26	5.28%
Conflict	23	4.67%
No longer maintained	22	4.47%
Senseless	20	4.07%
Incomplete	19	3.86%
Out of scope	13	2.64%
Expensive	6	1.22%
Killed by reporter	5	1.02%
Unrealistic	4	0.81%
Unfeasible	4	0.81%
Alternative	3	0.61%
Reduce portability	3	0.61%
Misclassification	2	0.41%
Other	24	4.88%
Total	492	100%

For example, a new feature request HB-1162[7] was proposed by a reporter, and marked as *Major* priority. This request got a quick response from one of the project leaders with a comment as "this feature has existed for three years."

Duplicate. 18.9% of the rejected feature requests got rejected due to the proposed requests are duplicate with existing feature requests. Existing research [7][8] on duplicate detection for bug tracking systems could alleviate this issue.

There are some other reasons such as Dead threads, Merged into other threads, Developers don't like, Usage questions, conflict, out of scope, etc. that also need to pay attention to. For example, 7.72% of the rejected feature requests got merged into other threads due to sharing similar topics. This result confirm the findings that has been reported by J. Cleland-Huang *et al*[6].

B. Effort for Identifying Redundant Feature Requests

In Section II.A, we identified redundant feature requests as the most noisy feature requests among 492 rejected requests. In this section, we further investigate the effort spent on identifying redundant feature requests in terms of duration and number of participants.

TABLE II
DESCRIPTIVE STATISTICS OF DURATION AND NUMBER OF PARTICIPANTS

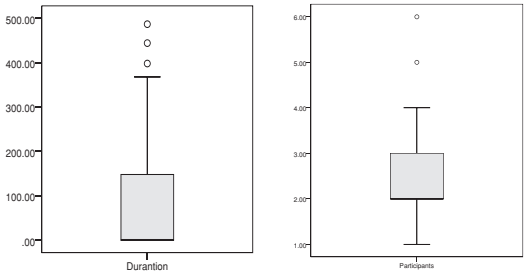
	Mean	MIN	MAX	Standard Deviation
Duration (in days)	260	0.002	2995.81	638.44
#Participants	2.4	1	5	0.73

We define duration of a feature request as the time span between the request is created and the request is resolved. We define number of participants as the number of individuals who involved in the discussion of a feature request.

[7]http://tinyurl.com/jjkz9bd

The descriptive statistics about the duration and the number of participants of the 115 redundant feature requests are shown in Table II. For each redundant feature request, it takes 260 days on average to resolve the request, and on average there are 2.4 contributors including the original reporter devoted to the discussion threads.

The distribution of duration of redundant feature requests is shown in Figure 2(a). The medium is 0.7 days, which means half of the requests get resolved in less than 0.7 days while the other half takes more than 0.7 days to be resolved. The duration of redundant feature requests varies in a wide range, which means some redundant feature requests take longer to be resolved while some take less. Figure 2(b) shows the distribution of the number of participants involved in redundant feature requests. The medium number is 2, which means half of the requests have more than 2 contributors that were involved.



(a) Duration (b) Number of participants

Fig. 2. Distribution of Duration and Participants

All the above results show that identifying redundant feature requests is very expensive. Since redundant feature requests are just noisy requests, which provide no benefits to the OSS community as well as waste efforts of the core contributors, our objective is to develop an automated method to identify such feature requests by comparing the feature request with the existing features.

III. Automate Redundant Requests Identification

In this section, we report our solution to identify redundant feature requests. The overview of the procedure is shown in Figure 4. First, we construct three candidate feature trees from user manual, release notes, and pull requests. Then we merge the three feature trees into a compound one. Given a new coming feature request, we search the merged feature tree and find the most similar sub-tree. Then we further find the most similar leaf node from the sub-tree. By comparison with the similarity threshold, we decide whether the input feature request is redundant or not. We present the procedure in details in the following sections.

A. Construct Candidate Feature Trees for OSS projects

We define a *feature tree model* as shown in Figure 3 to present features for OSS projects. The *RootNode* denotes the OSS project. The *StructNode* denotes the component that links to a set of features. The attribute *Keywords* summarizes the functionality of the component, and the group attribute $< Source, Summary, Description >$ includes source documentation where the description comes from, the summary of the feature, and the detailed description about the feature. We use plus sign to represent multiple entries and minus sign for single entry. Since source, summary, and description could be more than one entry, we use the plus sign. In this study, there are three sources: *UM*(User Manual), *RN*(Release Notes), and *PR*(Pull Requests).

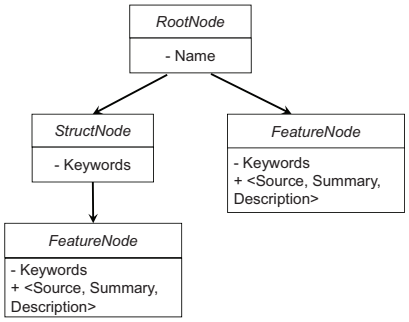


Fig. 3. Feature tree model.

Based on the feature tree model, we construct three candidate feature trees of OSS project like the following:

- **Construct the basic feature tree from user manual**. User manual provides instructions to assist users in using the system. It consists feature-related such as descriptions of features of the current system and usage of these features and non-feature-related parts such as software installation and FAQ (Frequently Asked Questions). The feature-related parts are organized in sections and subsections. The titles of these sections and subsections summarize one or more features that are described within, which can be considered as appropriate keywords for these features. Therefore, we use the titles of feature-related parts in user manual to construct the basic feature tree of OSS projects. First, we prune the user manual by excluding the non-feature-related parts such as overview and reference. Second, we extract the structure of the document to build *StructNodes*. We record the title of each section that are not the lowest level as keywords. Third, we consider the lowest level of subsection as the *FeatureNode*. We extract keywords from the title, and extract the paragraphs in the *Description*, and mark the *Source* as *UM*. Figure 5(a) shows an example of a partial feature tree built from user manual[8] of Hibernate ORM 5.1.
- **Construct the feature tree from release notes**. Typically each entry in release notes provides a link to its fixed issue report listed on issue tracking system. First, we collect all the entries in the release notes with their linked issue reports. Then we check each linked issue report and classify its corresponding entry as a newly added

[8]http://tinyurl.com/guodtsy

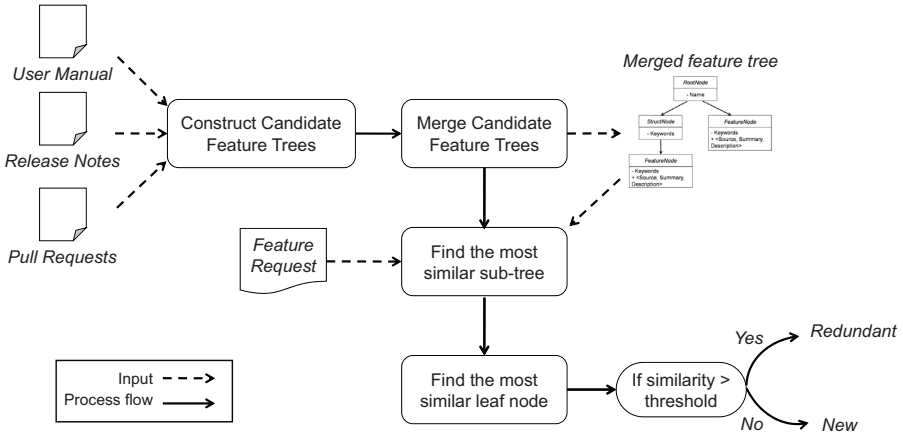


Fig. 4. The procedure of identifying redundant feature requests

feature if the linked issue report is labeled as *NEWFEATURE*. Third, we build the *StructNodes* by extracting the *component* attribute from the linked issue report. Finally, we build *FeatureNode* for each newly added feature by extracting the short summary from the release note, the textual description as *Description*, and mark the *Source* as *RN*. Figure 5(b) shows an example of a partial feature tree built from the release note[9] of Hibernate ORM 5.1.

- **Construct the feature tree from pull requests**. Since features that have already been implemented but not yet released are recorded in the pull requests, we consider the pull requests to be the third source to extract current features. Construct the feature tree from pull requests is similar to building from release notes. Pull requests contain linked issue reports as well as corresponding descriptions, so we can use the same procedure as we described above for release notes. Figure 5(c) shows an example of a partial feature tree built from pull request[10] of Hibernate ORM 5.1.

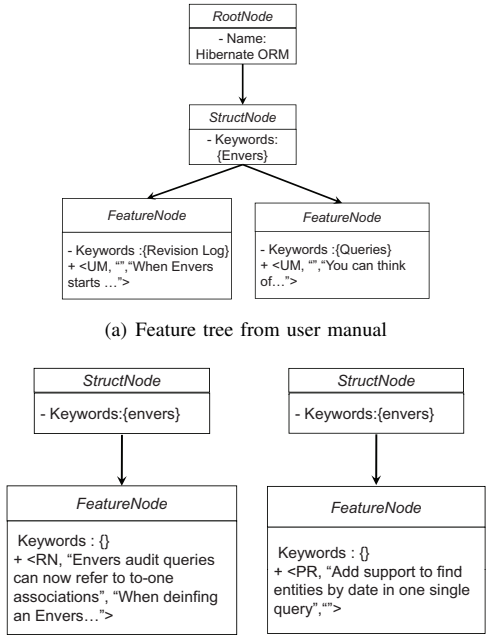
B. Merge Candidate Feature Trees

Once we have three candidate feature trees, we merge them together to obtain the final feature tree of the OSS project.

We use the feature tree FT_U constructed from user manual as the basic tree. First, for each *FeatureNode* f from feature tree constructed from release notes and pull requests, we retrieve the *StructNode* s with the highest textual similarity in FT_U. Second, for each *FeatureNodes* linked with s, we retrieve the *FeatureNode* f_U with the highest similarity between f. Third, we merge f into f_U. The example of the merging result of the three example feature trees in Figure 5 is shown in Figure 6.

C. Identify Redundant Feature Request

Once we derive the merged feature tree *FT*, we can use it to locate the lowest level of *StructNode S* that new feature request



(a) Feature tree from user manual

(b) Feature tree from release note

(c) Feature tree from pull request

Fig. 5. Examples of candidate feature trees

R should belong to by calculating the similarity between *R* and all *StructNodes* in *FT* and returning *S* with highest similarity.

With the *StructNode S* we found from previous step, we then can calculate the similarity between *R* and all *FeatureNodes* linked to *S*. If the highest similarity is greater than a predefined threshold *t*, which is calculated and trained using project historical data, then *FeatureNode F* with the highest similarity with *R* is considered the redundant feature request. Otherwise, *R* is a new feature request. The detailed procedure is described in Algorithm 1.

IV. Research Agenda and Application

The current paper examines the importance of identify redundant feature requests and briefly describes our solution.

[9]http://tinyurl.com/ha4j3ea
[10]http://tinyurl.com/h4j3kth

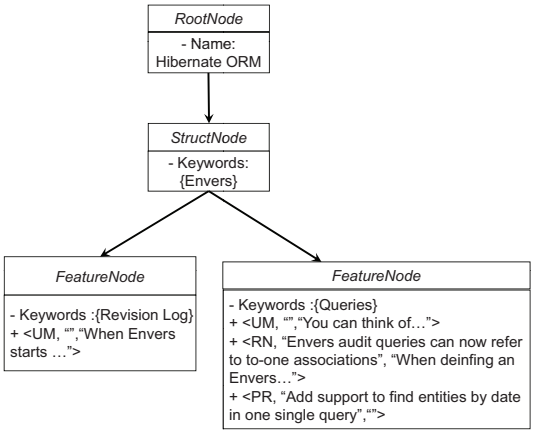


Fig. 6. Merged Feature tree.

Algorithm 1 Identify Redundant Feature Request

```
1: procedure REDUNDANTIDENT(Merged Feature Tree FT, New Feature Request R)
2:     s ← StructNode
3:     matchStructNode ← 0
4:     matchFeatureNode ← 0
5:     max ← 0
6:     for each StructNode s : FT do
7:         sim ← similarity(s, R)
8:         if sim > max then
9:             max ← sim
10:            matchStructNode ← s
11:    max ← 0
12:    for each FeatureNode f linkto matchStructNode do
13:        sim ← similarity(f, R)
14:        if sim > max then
15:            max ← sim
16:            matchFeatureNode ← f
17:    if max >= threshold then
18:        return true
19:    else
20:        return false
```

In the near future we plan to complete the research in the following steps.

Implement tools to build feature trees. We plan to implement tools that can use user manual, release notes, and pull requests as input to build and merge feature trees as we introduce in Section III. The key problem is how to accurately merge the feature trees according to the extracted keywords. Since release notes and pull requests may use a different terminology with the one used in the user manual, we plan to build words dictionary for core features.

Implement redundant identification algorithm. To experiment with various similarity comparison methods, we plan to include Dice's coefficient, Jaccard Index, cosine similarity [9][10], semantic similarity [11], and domain-related feature analysis [12] to improve the accuracy of the results. To increase the generalization of the classification threshold, we plan to train the threshold from large set of historical cross-project data. The key problem need to be solved is how to transform a natural language requests into a format that supports the similarity measurement with features in the feature tree. We plan to decompose the requests according to the textual metedata such as component, summary, and description, and then compute a balanced similarity according to the weights.

Prepare experiment data. we plan to collect all the rejected requests of enhancement and new features from Eclipse, Apache, Mozilla issue tracking systems. We select the requests those contain "already exists" or "already support" in their comments. Then we manually identify redundant feature requests by ourselves or by outsourcing. For identified requests, we collect all their meta attributes such as priority, component, and summary.

Evaluate the solution. We plan to evaluate the proposed approach by applying the algorithm on multiple OSS projects, such as Eclipse, Mozilla, Linux Kernel, and Apache. We can use the experiment data as new requests, and evaluate the precision and recall of the approach.

Develop two Bugzilla plugins. To collect feedback from OSS community to refine our solution, we plan to develop one plugin that prompts similar features to users when they type new feature requests, and another plugin that identifies redundant feature requests to contributors when there are new feature requests.

Empirical Analysis. Some feature requests got rejected due to they request features on a module that is no longer being maintained or request features that are out of the scope of the system. Some feature requests were rejected due to low quality such as senseless, incomplete, or unrealistic contents. We also plan to expand the scope of our empirical studies to investigate the categories of feature requests that are worthy of developers' attention.

Meanwhile, there are some limitations of this study. As the computation of the similarity between the issue and the node is based on the terms used, the approach will not allow identifying redundant issues where the reporter uses a different terminology than the one used in the user manual. Besides, the feature trees built in our approach is for descriptive purpose. Unlike the feature trees built in feature-oriented requirements engineering for verification purpose. It cannot clearly reflect the decomposed modules of the system since the cross-cutting features are not considered. However, the feature tree can be used as an initial source of input to a further design-oriented analysis.

V. Related Work

Automated analysis for OSS requirements. Cledland-Huang et al [6] designed an automated forum management (AFM) system, which is used to manage feature requests. This research provided a more accurate way to group feature requests, therefore, to make it easier for users to decide where to place new feature requests. Gill et al [13] proposed a semi-automation framework that solves the burdening natural language ambiguity problem in Open Source Software Development (OSSD). Through combining the positive attributes

of automation oriented domains and support the humans to arrive at precise and unambiguous requirements, the authors concluded that this framework can be used as a guide to OSSD community members to resolve ambiguities. Thung et al [14] proposed an automated recommendation approach that takes as input a textual description of a feature request. It then recommends methods in library APIs that developers can use to implement the feature. The results of the experiment shows that this approach is able to recommend the right methods from 10 libraries. Maalej and Nabil [15] leverage probabilistic techniques as well as text classification, natural language processing, and sentiment analysis techniques to classify app reviews into bug reports, feature requests, user experiences, and ratings. Their results show that the classification can reach the precision between 70-95% and recall 80-90% actual results. Although app reviews could be substantially different from feedback provided in OSS issue tracking system, the natural language processing and text classification techniques could benefit the feature-tree building process in our approach.

Analysis of continuous flow of industrial requirements. Dag et al [9] used automated similarity analysis, which is commonly used in industry for supporting requirements engineers to identify requirements duplicates and inter-dependencies. Through empirical studies, the authors concluded that they do not believe that the presented technique can replace human judgment, but the results suggest that automated similarity analysis on a syntactic level using information retrieval techniques may be effective in pinpointing true duplicates and inter-dependencies.

VI. Conclusion

In this paper, we conducted a survey in the Hibernate community to examine the characteristics of feature requests. There were 1898 feature requests retrieved from Hibernate JIRA page from the beginning of the project until 2016. 30% of the feature requests are unresolved and 70% are resolved. Among all the resolved feature requests, 63% of the feature requests are fixed while 37% are rejected. Among all the rejected feature requests, the highest percentage for rejection (23.37%) is due to redundancy, which means the proposed feature request describes feature that already exists in the current system. Developers in the community have to waste hours to manually go through and identify these redundant feature requests. These results led us to propose a procedure to automatically identify these redundant feature requests. With this initial algorithm, we plan to conduct further research with implementing it as a Bugzilla plugin and then evaluate with OSS projects to collect feedback from OSS community. We believe that our solution will benefit OSS project contributors from saving time and effort on manually identifying redundant feature requests.

VII. Acknowledgments

This material is based upon work supported by National Natural Science Foundation of China No.91318301, 61432001, 91218302. It is also supported by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract H98230-08-D-0171. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology. It is also supported by the National Science Foundation grant CMMI-1408909, Developing a Constructive Logic-Based Theory of Value-Based Systems Engineering.

References

[1] I. Morales-Ramirez, A. Perini, and R. Guizzardi, "An Ontology of Online User Feedback in Software Engineering," *Applied Ontology*, vol. 10, no. 3-4, pp. 297–330, 2015.

[2] Y. Cavalcanti, P. da Mota Silveira Neto, I. Machado, T. Vale, E. de Almeida, and S. Meira, "Challenges and Opportunities for Software Change Request Repositories: a Systematic Mapping Study," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 620–653, 2014.

[3] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer Recommender of Pull-Requests in GitHub," in *30th IEEE International Conference on Software Maintenance and Evolution,Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 609–612.

[4] E. Ben Charrada, A. Koziolek, and M. Glinz, "Supporting Requirements Update During Software Evolution," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 166–194, 2015.

[5] D. G. Novick and K. Ward, "Why don't People Read the Manual?" in *Proceedings of the 24th Annual International Conference on Design of Communication, SIGDOC 2006, Myrtle Beach, SC, USA, October 18-20, 2006*, 2006, pp. 11–18.

[6] J. Cleland-Huang, H. Dumitru, C. Duan, and C. Castro-Herrera, "Automated Support for Managing Feature Requests in Open Forums," *Communications of the ACM*, vol. 52, no. 10, pp. 68–74, 2009.

[7] A. Alipour, A. Hindle, and E. Stroulia, "A Contextual Approach Towards More Accurate Duplicate Bug Report Detection," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 183–192.

[8] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate Bug Report Detection with a Combination of Information Retrieval and topic modeling," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 70–79.

[9] J. N. och Dag, B. Regnell, P. Carlshamre, M. Andersson, and J. Karlsson, "A feasibility Study of Automated Natural Language Requirements Analysis in Market-driven Development," *Requirements Engineering*, vol. 7, no. 1, pp. 20–33, 2002.

[10] C. Rolland and C. Proix, "A Natural Language Approach for Requirements Engineering," in *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*, 2013, pp. 35–55.

[11] F. Zhao, F. Fang, F. Yan, H. Jin, and Q. Zhang, "Expanding Approach to Information Retrieval Using Semantic Similarity Analysis Based on WordNet and Wikipedia," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 2, pp. 305–322, 2012.

[12] N. Hariri, C. Castro-Herrera, M. Mirakhorli, J. Cleland-Huang, and B. Mobasher, "Supporting Domain Analysis through Mining and Recommending Features from Online Product Listings," *IEEE Trans. Software Eng.*, vol. 39, no. 12, pp. 1736–1752, 2013.

[13] K. D. Gill, A. Raza, A. M. Zaidi, and M. M. Kiani, "Semi-automation for Ambiguity Resolution in Open Source Software Requirements," in *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*. IEEE, 2014, pp. 1–6.

[14] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic Recommendation of API Methods from Feature Requests," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 290–300.

[15] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *2015 IEEE 23rd international requirements engineering conference (RE)*. IEEE, 2015, pp. 116–125.