Information and Software Technology 73 (2016) 37–51

Contents lists available at ScienceDirect

Information and Software Technology

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

A component recommender for bug reports using Discriminative Probability Latent Semantic Analysis

Meng Yan[b], Xiaohong Zhang[a,b,c,*], Dan Yang[b], Ling Xu[b], Jeffrey D. Kymer[b]

[a] State Key laboratory of Coal Mine Disaster Dynamics and Control, Chongqing 400044, PR China
[b] School of Software Engineering, Chongqing University, Chongqing 401331, PR China
[c] Key Laboratory of Dependable Service Computing in Cyber Physical, Society Ministry of Education, Chongqing 400044, PR China

ARTICLE INFO

Article history:
Received 9 February 2015
Revised 15 January 2016
Accepted 16 January 2016
Available online 23 January 2016

Keywords:
Bug reports
Discriminative topic model
Component recommendation
Bug triage

ABSTRACT

Context: The component field in a bug report provides important location information required by developers during bug fixes. Research has shown that incorrect component assignment for a bug report often causes problems and delays in bug fixes. A topic model technique, Latent Dirichlet Allocation (LDA), has been developed to create a component recommender for bug reports.
Objective: We seek to investigate a better way to use topic modeling in creating a component recommender.
Method: This paper presents a component recommender by using the proposed Discriminative Probability Latent Semantic Analysis (DPLSA) model and Jensen–Shannon divergence (DPLSA-JS). The proposed DPLSA model provides a novel method to initialize the word distributions for different topics. It uses the past assigned bug reports from the same component in the model training step. This results in a correlation between the learned topics and the components.
Results: We evaluate the proposed approach over five open source projects, Mylyn, Gcc, Platform, Bugzilla and Firefox. The results show that the proposed approach on average outperforms the LDA-KL method by 30.08%, 19.60% and 14.13% for recall @1, recall @3 and recall @5, outperforms the LDA-SVM method by 31.56%, 17.80% and 8.78% for recall @1, recall @3 and recall @5, respectively.
Conclusion: Our method discovers that using comments in the DPLSA-JS recommender does not always make a contribution to the performance. The vocabulary size does matter in DPLSA-JS. Different projects need to adaptively set the vocabulary size according to an experimental method. In addition, the correspondence between the learned topics and components in DPLSA increases the discriminative power of the topics which is useful for the recommendation task.

1. Introduction

Bug report is a fundamental artifact for informing developers about software problems. Research on mining bug report repositories has demonstrated success in a variety of software activities, such as tracing the evolution of the project [1], evaluating developer's expertise and contribution [2] and improving the software product quality [3]. However, the use of a bug report depends on correct triaging. Every new bug report has to be triaged to a component when submitted. This allows an efficient fixing by the appropriate development team which is responsible for the component [4]. Previous research reported that bug reporters frequently make inaccurate decisions in assigning the categorical fields such as the component in a bug report [5]. Unfortunately, bug reports are always triaged manually, which is time consuming and error prone. Take Eclipse as an example, approximately 25% of the bug reports need to be reassigned because of triaging mistakes [6], and it costs nearly two person-hours per day in triaging bug reports [7]. To assist correct bug triaging, this paper presents a novel Discriminative Probability Latent Semantic Analysis (DPLSA) based component recommender, in which a recommender is provided with a small list of component suggestions.

Across the popular issue tracking systems, such as JIRA[1] and Bugzilla[2], the bug reports submitted by reporters possess the following common features: first, there are a variety of mandatory and categorical fields like component, product, status and assigned to. Second, there are some other fields which are non-structured natural language text, including report title, detail description and

* Corresponding author at: School of Software Engineering, Chongqing University, Chongqing 401331, PR China. Tel.: +86 15923238399.
E-mail address: xhongz@cqu.edu.cn (X. Zhang).

[1] http://www.atlassian.com/software/jira/, verified 2015 /08/13.
[2] http://www.bugzilla.org/, verified 2015 /08/13.

http://dx.doi.org/10.1016/j.infsof.2016.01.005
0950-5849/

comments which are written by developers or users. Third, each free-form text field in the bug reports contains rich information which reflects different facts about the bug. For example, the full free-form description text field reflects the detailed bug effects and provides indispensable conditions to track the bug. By utilizing these features, researches have proposed a variety of approaches to investigate different bug report related activities, such as duplicate bug report detection [8], bug report summarization [9] and bug triaging recommendations [6]. In this work, we focus on creating a component recommender by utilizing the full free-form text fields including title, description and comments.

This proposed method is motivated by the recent success of topic modeling in mining bug reports [4,10-13]. The most similar work is a Latent Dirichlet Allocation (LDA) [14] based component recommender proposed by Somasundaram and Murphy [4]. Although they achieved a state-of-art performance, there are still several unsolved issues in this line of research. First, a critical issue in topic modeling is how many topics should be sought [15]. There is no agreed-upon method for choosing the right number of topics among different datasets. In the similar work proposed by Somasundaram and Murphy [4], the number of topics varied from 10 to 120 and it was determined by using a series of experiments. Second, it is rarely discussed that the comments in bug reports decrease the performance of the recommendation or improve it. It is hasty to decide whether or not to consider comments. There are different decisions in related works. For example, Naguib et al. [16] did not adopt the comments in building the recommender. Shokripour et al. [17], Xuan et al. [18] and Zhang et al. [19] used the comments in their recommendation works. Third, the vocabulary in bug reports has an impact on the bug assignment accuracy [20]. Meanwhile, vocabulary plays a significant role in topic modeling [14]. However, it is rarely discussed whether the vocabulary size impacts the component recommender based on topic modeling.

To address the aforementioned issues, we present a Discriminative Probability Latent Semantic Analysis (DPLSA) model with discriminative power to create a recommender that assists with the component assignment task. We divide our recommender into three phases, namely the training phase, the testing phase and the recommendation phase. In the training phase, we use past assigned bug reports as our training datasets. The number of topics is simply set to the number of components. We develop a novel method to initialize the word distributions for different topics. This introduces the past component labels to initialize the topic-conditional probability of a particular word. Thus, the topics are estimated in a supervised way. As a result, it creates a correspondence between learned topics and components. This increases the discriminative power of the learned topics and overcomes the difficult in determining an appropriate topic number in LDA. In the testing phase, given a new bug report, we fix the obtained word-topic distribution in the training phase and compute the test sample's topic distribution by the standard EM algorithm of Probability Latent Semantic Analysis (PLSA). In the recommendation phase, the component recommendations are decided by ranking the divergence with each component's centroid topic representation in the training set. A small list of top-k most suitable components are recommended. The similarity with Somasundaram and Murphy [4] is that both of us use the topic modeling and divergence measuring technique. The main differences from Somasundaram and Murphy [4] lie in: first, we investigate the impacts of comments and vocabulary size which is not discussed in their work. Second, we use DPLSA instead of LDA. In summary, the contributions in our paper are threefold:

- We propose DPLSA, which performs an initialization method in the PLSA modeling step. It assigns the category-conditional probability of a specific word conditioned on the corresponding topic by using the assigned bug reports. Such that a correspondence between components and topics is created. It is utilized to enhance the discrimination of the estimated topics and overcomes the difficulty in choosing the right number of topics. (See Section 3.2.1).
- We evaluate our DPLSA-JS recommender on five open source projects to validate the effectiveness. Compared with two LDA based methods [4], we show that DPLSA-JS outperforms LDA-KL and LDA-SVM on the component recommendation problem by a substantial range. (See Section 4.5.1).
- We explore the impact of vocabulary size and comments on the recommendation performance by conducting a comparative study on five datasets. As a result, we find that the vocabulary size and whether or not using the comments impact the recommendation performance (See Sections 4.5.2 and 4.5.3).

The paper is structured as follows: Section 2 presents the related work of our research, including bug triage recommenders, topic modeling in mining bug reports and similar supervised topic models. We describe our research preparation, models and techniques in Section 3. Section 4 presents the research questions, sketches the experiment design, results and comparisons. Section 5 provides the threats to validity, including internal validity and external validity. Then at last in Section 6, we draw a conclusion about our findings and provide our future plans.

2. Related work

In this section, we discuss related literature from three aspects: bug triage recommenders, topic modeling in mining bug reports and similar supervised topic models.

2.1. Bug triage recommenders

A variety of techniques have been investigated to guide the bug triaging process, such as duplicate bug report detection [8,21,22], developer recommenders [7,12,13,16,17,23-27] and component recommenders [4-6].

Many researches have focused on automating the process of detecting duplicate bug reports by analyzing the free-text fields which is also used in this work. Several kinds of methods were employed, such as the statistical model, vector space model (VSM) [28], and TF-IDF [21]. In 2005, Anvik et al. [22] built a statistical model, using cosine similarity and detected 28% of all duplicate reports on Firefox. Hiew [29] improved on it by applying VSM to detect duplicate bug reports. The TF-IDF term weighting measurement was employed in their work and achieved a 50% recall and 29% precision on Firefox and 20% and 14% for Eclipse, respectively. The similarity between their work and this work is the textual processing on non-structured natural language fields of bug reports. While they differ from our work in how the training data is labeled and what data is used.

There have been several researches on creating developer recommenders which automatically assign a bug report to a particular developer. Cubranic [30] first modeled a bug report developer recommender by using a text classification method. After that, Anvik et al. [6,7] enhanced the above work by removing inactive and less active developers (according to the count of recent bug fixes). In their experiments on five open source projects, they increased the precision accuracy up to 64% by using three machine learning classifiers, namely C4.5, Naive Bayes and SVM. Based on these works, several approaches such as bug tossing (i.e., bug reports are reassigned to other developers) and source location based methods were proposed to enhance the existing performance. For example, Jeong et al. [24] presented an enhanced

bug triage technique by capturing bug tossing history. Their model reduced the tossing steps by up to 72% and improved the bug assignment accuracy by up to 23%. Chen et al. [31] combined the tossing graph and VSM text similarity to enhance the bug triage efficiency on Mozilla and Eclipse. Bhattacharya et al. [32] provided a refined classifier by learning a precise ranking method for tossing recommendation and reduced an average of 1.5–2 tosses on tossing path lengths. Linares-Vasquez et al. [26] and Shokripour et al. [17] provided a novel perspective, i.e., a source location based method for developer recommendation. Their common feature lies in that the relevant source location is addressed before the recommendation. The difference of the two works is that code authorship is used in Linares-Vasquez et al [26] while code commit is used in Shokripour et al [17]. Our work differs from these works in selecting topics as features instead of using terms. Also it does not require mining other software repositories, such as source code files and code commits.

With respect to the component recommender, Di Lucca et al. [33] were the first to perform the component recommendation task by using various machine learning methods. Their recommender triaged a bug report to one of eight components on a commercial software dataset. In their conclusion, they stated that the SVM based component recommender performed better than other algorithms. However, one issue is that the reports in their evaluation were created by a technical support team. It may have used a more constrained language than the reports which come from a diverse population. Anvik et al. [6,34] improved their work on five open source projects with 11–34 components by using SVM. The bug reports used in [6] came from open source projects which possessed a diverse population. Sureka [5] was the first to enhance the component recommendation by making use of the incorrect assignments. They suggested that learning the correspondence between words in bug reports and components was significantly useful in performing the component recommendation task. It also supported the method in this work.

2.2. Topic modeling for mining bug reports

The idea of extracting high-level topics from bug reports using topic models (e.g. LDA and PLSA) has been investigated in recent years. It has been used to enhance diverse activities, such as bug report retrieving, duplicate bug report detection and bug triaging. Compared with the traditional VSM method which does not consider polysemy or synonymy, topic models have been found to be superior in several software activities [35,36]. Specially, LSI [37] demonstrated success in quite a few recent works on mining bug reports. Dit et al. [38] presented an LSI based method to measure the textual coherence of bug reports by extracting topics from the textual content found in the descriptions and comments of the bug reports. They confirmed that the topic similarity was a good indicator of textual coherence of bug report comments. Ahsan et al. [39] were the first to create a developer recommender use topic modeling. They applied LSI to obtain a reduced term-document matrix on the titles and descriptions. The best recommender in their approach achieved 45% classification accuracy. However, one drawback of LSI is that it is difficult to interpret the results which are represented with numeric spatial representation [40]. Besides, LSI can only sketch the synonymy, it cannot handle the polysemy as well [40]. To overcome this, the generative topic models such as PLSA and LDA (with a more complete foundation in statistics) were widely used in analyze bug reports. Details about comparing LSI and the generative topic model are presented in Hofmann's work [41].

Along with the progress of topic models, several researches have applied generative topic models and their variants and extensions to enhance bug report triaging. Asuncion et al. [42] developed TRASE, which recovered traceability links amongst diverse artifacts in software repositories, such as bug reports and change requests. The authors demonstrated that LDA outperforms LSI by comparing recall and precision. Lukins et al. [40] developed an automatic bug localization approach using LDA and demonstrated the performance over open source projects. Xie et al. [23] proposed a DRETOM recommender based on LDA to recommend a small list of potential developers for fixing bugs in a collaborative way. Naguib et al. [16] proposed a novel activity profile based approach for deverloper recommendation. They utilized LDA to help create a bug-tracking activity profile for every user. Geunseok et al. [13] combined LDA and multi-feature to create a developer recommender. Xie et al. [12] proposed a composite developer recommender which performs both bug report based analysis and developer based analysis. They employed LDA to characterize topic features to measure the distance between bug reports. However, a common issue in most of these works is that they adopted the original topic model approach. The original goal of the topic model was not for inference or classification, but rather representation and compression of signals. Very similar to this work, Somasundaram and Murphy [4] provided a component recommender by using LDA and the Kullback–Leibler divergence. They achieved a state-of-art recall on three open source projects. However, there are several unclear issues, such as how to decide the number of topics and how to decide the vocabulary size. We implemented their work on our datasets to compare the recommendation performance.

2.3. Similar supervised topic models

Compared with DPLSA, there are three similar supervised models, namely sLDA [43], L-LDA [44] and Semi-supervised LDA [45]. The sLDA added a response variable (e.g., the category of a document or the ratings attached to a movie) into LDA which is associated with every document. Then, in order to find latent topics to predict the response variables for newly given documents, sLDA jointly modeled the documents and the responses. The similarity between DPLSA and sLDA is that all the documents in the training set are labeled. The difference lies in that DPLSA supervised the model using an initialization method while sLDA supervised the model by jointly modeling the document and response. Besides, the number of topics in DPLSA is equal to the number of categories. While in sLDA, there is no agreed-upon method to choose the correct number of topics.

L-LDA [44] extended the LDA model by creating a 1-1 correspondence between topics and categories. The similarities between L-LDA and DPLSA are (1) all the documents in the training set are labeled; (2) the number of topics is equal to the number of categories. The differences lies in that L-LDA [44] is mainly focused on a multi-labeled corpora. There is an assumption in L-LDA: that the documents are multiply tagged with human labels, both at learning and inference time [44]. While the initialization method in DPLSA makes it feasible to handle single-category corpora, such as the labeled bug reports which only belong to one component.

Fu et al. [45] presented a semi-supervised LDA method for classifying software change messages. They added signifier documents into the training set to transform the unsupervised training into a semi-supervised one. A signifier document is a sample which is labeled with a category. It is constructed by a pre-defined vocabulary. The similarities between the semi-supervised LDA and DPLSA are that (1) both of them use labeled documents and topic modeling techniques, (2) the number of topics is equal to the number of categories. The difference lies in that (1) the signifier documents which are necessary in semi-supervised LDA need a pre-defined vocabulary. It provides the keyword list for each category. While in DPLSA, this information is not needed. (2) Part of the training set in semi-supervised LDA is labeled as signifier documents. While

all the training samples are labeled in DPLSA. (3) Semi-supervised LDA differs from the original topic model in the training set. DPLSA differs from the original topic model by using a supervised initialization method.

3. Proposed approach

3.1. Preliminary

In our work, a bug report is regarded as a document. A topic is an abstract concept which possesses a distribution of words. The topic model technique in bug report analysis is used to cluster bug reports into topics. It is done by categorizing the terms occurring in bug reports into topics. For example, a bug report about a UI bug is more likely to correlate with the topic which highly corresponds to terms such as "button" and "style". With this technique, we can analyze the bug reports on the topic representations.

Among all the topic models, a common and fundamental idea is that documents are mixtures of topics, while topics are multinomial distributions of terms like a unigram language model. The differences in various topic models are the statistical approaches and the generation process [46]. For example, in PLSA, the topic mixture is conditioned on each document. In comparison, the topic mixture in LDA is drawn from a conjugate Dirichlet prior. The PLSA model is an extension of the Latent Semantic Analysis. One assumption of PLSA is that topics are decomposed from documents, while terms are generated from a mixture of topics. In addition, the occurring term order in a document is ignored in PLSA. We briefly sketch the main principles of PLSA and its extension in this work. The details of PLSA can be found in Hofmann's work [41].

To simplify the notations in this paper, we adopt the simple notations used by [46]. A bug report is regarded as a document and a component is regarded as a category. Let D represent the collection of all the bug reports, V represents the term vocabulary, $P(z|d)$ represents the distribution over topics for a particular bug report d and $P(w|z)$ represents the probability distribution over words for a topic z. To simplify, we use $\phi_w^{(j)} = P(w|z=j)$ to represent the $P(w|z)$ for topic j and $\theta_j^{(d)} = P(z=j|d)$ to represent the $P(z|d)$ for document d. For PLSA, the conditional probability $P(w|d)$ over latent topics is represented as follows.

$$P(w|d) = \sum_{j=1}^{K} \phi_w^{(j)} \theta_j^{(d)} \tag{1}$$

The Expectation-Maximization (EM) algorithm [47] is employed for fitting the model. The parameters $\hat{\theta}$ and $\hat{\phi}$ are estimated by maximizing the log likelihood of the observed data:

$$\log P(D|\phi,\theta) = \sum_{d\in D}\sum_{w\in V}\left\{ c(w,d)\log\sum_{j=1}^{K}\phi_w^{(j)}\theta_j^{(d)}\right\} \tag{2}$$

In which $c(w, d)$ denotes the count of term w in bug report d. The hidden variable z is estimated by using the previous iteration in the E-step.

$$P(z_{d,w}=j) = \frac{\phi_w^{(j)}\theta_j^{(d)}}{\sum_{j'=1}^{K}\phi_w^{(j')}\theta_{j'}^{(d)}} \tag{3}$$

Then, in the M-step, the parameters $\theta_j^{(d)}$ and $\phi_w^{(j)}$, respectively, are updated as:

$$\theta_j^{(d)} = \frac{\sum_{w\in V} c(w,d)P(z_{d,w}=j)}{\sum_{j'}\sum_{w\in V} c(w,d)P(z_{d,w}=j')} \tag{4}$$

$$\phi_w^{(j)} = \frac{\sum_{d\in D} c(w,d)P(z_{d,w}=j)}{\sum_{w'\in V}\sum_{d\in C} c(w',d)P(z_{d,w'}=j)}. \tag{5}$$

The parameter $\phi_w^{(j)}$ denotes which words are informative for a topic while $\theta_j^{(d)}$ denotes which topics are relevant for a document. In fact, the purpose of PLSA is to find the most informative words for a particular topic. However, the basic PLSA ignores the features of the training samples for the classification problem, such as the structure of the training samples from a single category, the relationship between the samples from a single category and the word-topic distributions, and the relationship between topics and categories. In addition, PLSA randomly initializes the word-topic distributions $\phi_w^{(j)}$ so that words in a document are connected to topics in an uncontrolled way. It is difficult to obtain the discriminative information in $\hat{\theta}_j^{(d)}$.

We address the above issues in the following sections by capturing the features of bug reports from the same component and designing a word-topic distribution initialization method. After that, the estimated model provides insights about how to interpret each topic with the discriminative information using $\phi_w^{(j)}$ and how to obtain the topic-document distribution $\hat{\theta}_j^{(d)}$ to perform the recommendation task.

3.2. Data extraction and preprocessing

In this paper, bug reports are extracted from Bugzilla repositories. Since the bugs stored in Bugzilla have one characteristic, the description field of a report is unchangeable after a bug report is submitted. This feature allows us to easily trace the information from when the bug was submitted [4]. The preprocessing step is similar to our previous work [45] using Lucene[3] and Snowball[4]. The preprocessing step consists of sentence splitting, term splitting, stop words filtering, and stemming. We stemmed the words (e.g. "submitting" becomes "submit") to decrease the vocabulary size and reduce duplication due to the word form. In addition, non-informative words were removed to reduce noise, such as prepositions and pronouns. The stop words list in this work is derived from Mallet.[5]

3.3. Building component recommender

This section provides a running example of the proposed approach first. Then we describe the two steps in creating the recommender in detail: (1) the DPLSA modeling step, including model training and testing, and (2) the recommendation step.

3.3.1. Running example

This section provides a running example to explain our approach as illustrated in Fig. 1. Four training bug reports ($D1$,$D2$,$D3$ and $D4$) and a test bug report ($D5$) are abstracted to give the example. Suppose that $D1$ and $D2$ are already assigned to component UI while $D3$ and $D4$ are already assigned to component Core. The process is divided into four phases. In Fig. 1(a), the recommender starts by a simple supervised initialization step such that the topics and the components are correlated. In detail, the training bug reports from the same component are employed to initialize the topic-conditional probability of a particular word. Hence, semantically similar words are forced to connect to the topic partially with a dominant probability (i.e., relatively higher than most of the other entries). As Fig. 1(b) shows, words $W1$, $W2$ and $W3$ connect

[3] http://lucene.apache.org/, verified 2014 /09/28.
[4] http://snowball.tartarus.org/, verified 2014 /09/28.
[5] http://mallet.cs.umass.edu/, verified 2014 /09/28.

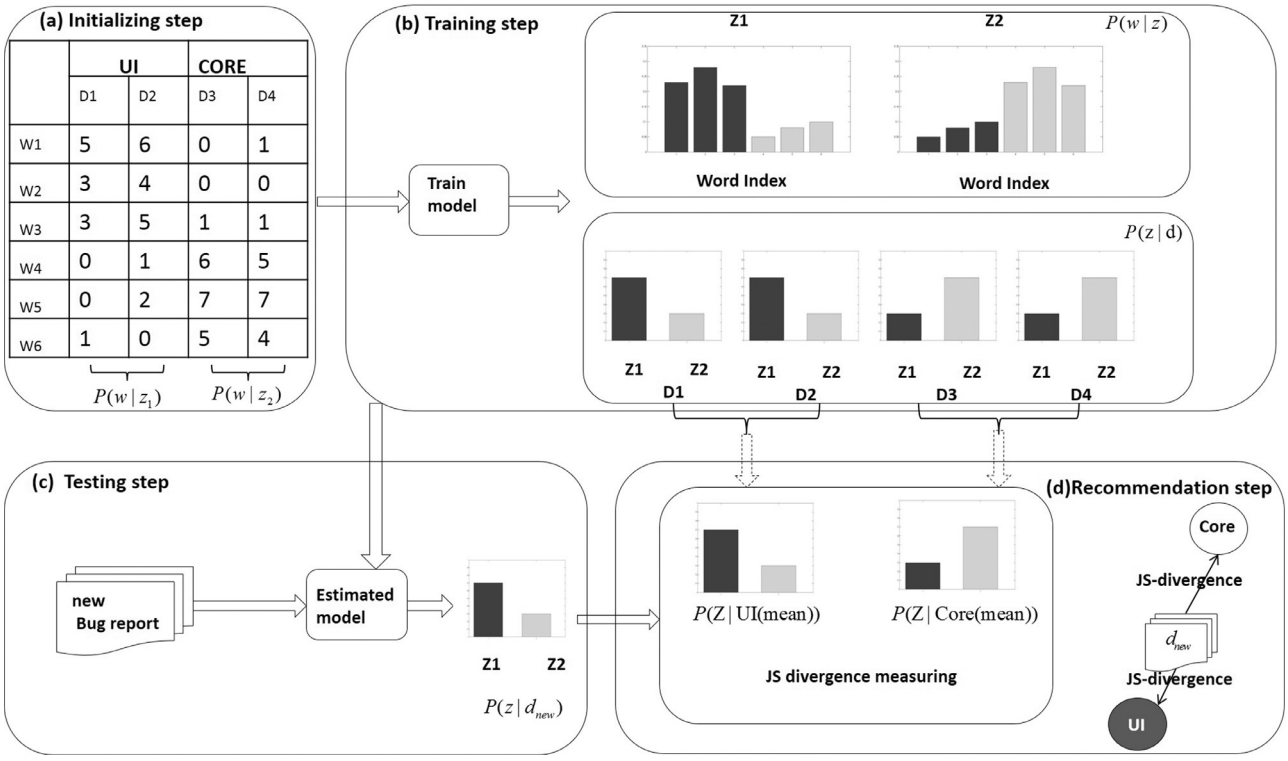


Fig. 1. The component recommendation process using DPLSA-JS: (a) Initialization step. (b) DPLSA training step. (c) DPLSA testing step. (d) Recommendation step.

to the $Z1$ topic with a similar dominant probability. In the following, the standard Expectation-Maximization (EM) algorithm [47] in PLSA is utilized for estimating the conditional distribution of words within the topics. Due to the supervised initialization approach, each topic correlates with one or more closely related components, like topic $Z1$ strongly correlates with component UI in Fig. 1(b). After that, for a test bug report, we fix the obtained conditional distributions of words within topics and compute the test sample's topic distributions by the standard EM algorithm of PLSA. Fig. 1(c) shows the topic representation of the test sample. It reflects that the test sample is closely related with topic $Z1$. Finally, the recommendation step is decided by measuring the divergence with each component's topic distribution centroid which is represented by the average document-topic distribution (i.e., $P(Z|UI(mean))$ and $P(Z|Core(mean))$ in Fig. 1(d)). Take $D5$ as the example, it is closer to $P(Z|UI(mean))$. Thus, we recommend the component UI to the new bug report $D5$ (in the case of 1 component recommendation). Details and formulations on how to create and configure the recommender are below. Section 3.3.2 formulates the DPLSA modeling step, including model initializing, training and testing. Section 3.3.3 provides the recommendation approach.

3.3.2. DPLSA modeling step

Suppose we are given a collection of N components in a project. In the training set, a bug report is already labeled by a component in the bug repository. Set $D^{(j)}$ $(j = 1, 2, 3, ..., N)$ denotes the collections of bug reports for each component j. It is worth noting that $D^{(j)}$ is a subset of D.

In the training step, we suggest a simple initialization approach in DPLSA which replaces the random initialization approach in PLSA. It initializes $\phi_w^{(j)}$ using the training samples from a single category such that word w dominantly connects with one or more topics and the redundant features are pruned for $\phi_w^{(j)}$. Meanwhile, this initialization method determines the number of topics directly equal to the number of components which also alleviates the difficult of choosing the right number of topics. For formulation, the initialization step is as formula (6) and formula (7) shows.

$$\phi_w^{(j)} = c(w, D^{(j)})/c(D^{(j)}) \tag{6}$$

$$c(w, D^{(j)}) = \sum_{d \in D^{(j)}} c(w, d), \quad c(D^{(j)}) = \sum_{v \in V} \sum_{d \in D^{(j)}} c(v, d) \tag{7}$$

In which $\phi_w^{(j)}$ is the objective of the initialization, it represents the word-topic distribution for topic j. $D^{(j)}$ denotes the collection of bug reports in component j, $c(v, d)$ represents the count of each word v in the vocabulary V occurring in document d, $c(w, D^{(j)})$ represents the count of word w occurring in the collection $D^{(j)}$, $c(D^{(j)})$ is the count of all words in the vocabulary V in the collection $D^{(j)}$.

In this manner, DPLSA reduces redundant connections between topic j. The estimated $P(w|z)$ over the vocabulary V turns out to be a matrix which contains several entries with dominant values for each topic, thus, the $\hat{\phi}_w^{(j)}$ effectively reflects which words are informative for topic$j$. After training, given a test bug report d_{new} and fixing the estimated value $\hat{P}(w|z)$ from DPLSA, we obtain the topic document distribution $\theta_j^{(d_{new})}$ of the test bug report d_{new} by the EM algorithm. The topic coverage distribution for a bug report provides an alternative discriminative semantic representation of the bug report, which is promisingly superior to the original word-based representation in performing the recommendation task because topics which correlate with components are discriminative.

3.3.3. Recommendation step

For the estimated topic distribution θ_j of all the training bug reports in D, we compute the average document topic probability $\theta_j^{d_{mean}}$ of all reports in one component. Let $\theta_j^{d_{mean}}$ serve as the centroid of the component. The divergence of a test bug report $\theta_j^{(d_{new})}$ from the $\theta_j^{d_{mean}}$ of a component determines the recommendation.

Table 1
Experimental data sets.

Projects	Domain	Triage process	Developers	Components
Platform	Programming tool	Developer-based	151	18
Bugzilla	Issue tracking	Volunteer-based	43	21
Mylyn	Programming tool	Developer-based	13	14
Gcc	Compiler	Developer-based	81	39
Firefox	Browser	Volunteer-based	343	55

When measuring the divergence, we adopt the Jensen–Shannon divergence (JSD) to perform this task. It is based on the Kullback–Leibler divergence, extended by several notable differences, including that it is always well-defined, symmetric, and bounded [48]. The JSD between two distributions is computed as formula (8) and (9) shows.

$$d_{JS}(H,K) = d_{KL}\left(H, \frac{H+K}{2}\right) + d_{KL}\left(K, \frac{H+K}{2}\right) \tag{8}$$

$$d_{KL}(H,K) = \sum_i h_i \log \frac{h_i}{k_i} \tag{9}$$

In Eqs. (8) and (9), H denotes the $\theta_j^{(d_{new})}$, K denotes $\theta_j^{d_{mean}}$, h_i and k_i denote the entry of topic i. After computing the JSD of a test bug report, our DPLSA recommender (referred to DPLSA-JS) will recommend a small list (e.g. top three) of the component recommendations, whose average document topic probability $\theta_j^{d_{mean}}$ diverge the least from the test bug report.

In summary, the recommendation step in this work is similar to the recommendation step in the LDA based method proposed by Somasundaram and Murphy [4]. The similarity is that both of them use the divergence measuring policy. Namely, the recommendation is decided by measuring the divergence of a test bug report's topic-document distribution from the average topic-document distribution of all training reports in each component. The differences lie in (1) the topic-document distribution in this work is estimated by DPLSA while their work relied on LDA, (2) we use the JS-divergence while their work used the KL-divergence to measure the divergence.

4. Experiments

4.1. Data sets

We chose five open source projects to build, test, and verify our recommender, i.e., Eclipse platform (referred to Platform), Bugzilla, Mylyn, Gcc and Firefox (Table 1). The following criteria were used to select the projects:

- ***Bug reports repositories accessible.*** The candidate open source projects were mature projects and had public accessible bug reports repositories.
- ***Number of bug reports, components and developers.*** For this work, we chose to use projects which have many components, mature development communities, and have a large set of past assigned bug reports for training. We consider only projects with at least 6 000 bug reports and 10 components. In order to validate if the topic modeling understands natural languages and the expression diversity of different developers, we consider only projects with at least 15 developers.
- ***Repository type.*** As mentioned before, we consider only projects with Bugzilla repositories.
- ***Previous research.*** To make our work comparable, all of the candidate projects were investigated by previous researches [6].

4.2. Experimental setup

In setting up our experiment, we use a time-sequential policy [6] rather than the general cross validation to divide the training set and testing set. The advantage of the time-sequential policy is that it is closer to the realistic situation when using the recommender. In detail, the recommender was trained by using a few months of bug reports and validated on the bug reports of the following months [6]. Under this policy, we determine the number of reports in the training phase and testing phase by fixing the proportion of the training time period and the testing time period. For example, if we collect the reports for one year, we will test the method on the latest month. However, the active degree of the project community differs in different projects. For example, there are only 159 bug reports which satisfy our retrieving criteria from 2011-8-31 to 2014-8-31 in project Mylyn. Therefore, the time period may vary in different projects. Besides, at any moment a bug report in Bugzilla is at a particular phase of the report life cycle. The reports in the phase of NEW or UNCONFIRMED do not provide information in creating the component recommender [6]. As a result, we ignored these reports. In addition, we consider all bug reports with the status FIXED and WONTFIX, which is the same as the work by Somasundaram and Murphy [4]. We also removed reports with the INVALID status, such as the #427193 bug report in Eclipse which does not contain any valid information in its title and description. Table 2 outlines the time periods of the bug reports in our training and testing set for each project.

4.3. Performance measure

Since there is only one component in a bug report, measuring the precision did not provide insightful results [4,6]. The recall and the precision are the same when performing one recommendation. Also, when performing two and three recommendations, the achieved performance would be a 50% and 33% precision, respectively. Hence, we evaluate the recommendation performance by using the standard measure recall which is also used in previous component recommenders [4,6] (see formula (10)).

$$\text{Recall} = \frac{\#of\ appropriate\ recommendations}{\#of\ reports\ intest\ set} \tag{10}$$

For simplification, we adopt "recall @k" [12] to represent the recall when we recommend top-k (e.g., top-1, top-3, and top-5) components.

4.4. Research questions

To investigate the effectiveness of our method, we designed the following research questions:

- **RQ1** *Is the DPLSA-based method better than LDA-KL or LDA-SVM for recommending components to incoming bug reports?* The similar work [4] proposed LDA-KL and LDA-SVM methods which were comparable to those machine learning methods previously developed. We wish to answer this question by evaluating the performance of the DPLSA based method

Table 2
Characterization of training set and testing set.

Projects	Training time period	Training reports	Testing time period	Testing reports
Platform	2011.8.31–2014.5.30	4002	2014.5.31–2014.8.31	247
Bugzilla	2007.8.31–2014.1.30	3302	2014.1.31–2014.8.31	192
Mylyn	2005.8.31–2011.2.27	4899	2011.2.28–2011.8.31	53
Gcc	2011.8.31–2014.5.30	5204	2014.5.31–2014.8.31	250
Firefox	2013.8.31–2014.7.30	3850	2014.7.31–2014.8.31	256

and compare it with LDA-KL and LDA-SVM methods on five datasets.

RQ2 *Do comments provide information that increases component recommendation performance of our approach?* There is no agreed-upon decision whether comments in bug reports decrease the performance of the recommendation or improve it. There are different decisions in the state-of-art works, such as some works did not use the comments [16] and other works used the comments [17-19,34]. We wish to answer this question by performing a comparative study on five datasets.

RQ3 *What is the effect of varying the size of the vocabulary to the performance of our approach?* Vocabulary varies in different projects. It's hasty to state that the larger the vocabulary size is the better the performance will be. Some odd words which only occur once are useless for the recommendation task. Thus, we wish to answer this research question by conducting a correlation analysis over different vocabulary sizes.

RQ4 *Do the learned topics correspond to components in our approach?* The number of topics in the similar work [4] is set from 10 to 120. It is hard to interpret what a topic represents. We wish to create a correspondence between topics and components by using the Discriminative Probability Latent Semantic Analysis. The number of topics is set as the component number and the topic can be interpreted with a few components which are correlated to the topic with a high probability.

4.5. Results and analysis

To analyze the effectiveness of the proposed recommender, to compare this method with the LDA-KL and LDA-SVM methods and to explore the configurations or parameters of our DPLSA method, we divided our experiments into four sub-sections. Section 4.5.1 (RQ1) is to compare our method to two state-of-art methods over the same data. Section 4.5.2 (RQ2) and Section 4.5.3 (RQ3) are designed to evaluate the impact of comments and vocabulary size on the performance of our method. This proposed method is superior to the LDA based methods because of the discriminative property. The goal of Section 4.5.4 (RQ4) is to examine how and why the topics possess discriminative power.

4.5.1. Performance of DPLSA-JS vs. LDA-KL and LDA-SVM (RQ1)

Both the LDA-KL and LDA-SVM methods [4] were implemented to conduct the comparison. In terms of the variables in implementing the two LDA based methods, the number of topics T is set from 10 to 120 (stepping by 10) and keeping the hyper-parameters α to be $50/T$ and β to be 0.1 as they described. However, the number of topics in DPLSA is equal to the number of components. Therefore, to make the comparison more comprehensive, the best topic number for the LDA based methods needs to be decided first. For example, let us assume the vocabulary size is equal to 5000, the recall of LDA-KL and LDA-SVM over a different number of topics is shown in Table 3. It shows that the best setting for the number of topics differs over the number of recommendations. Therefore, for each vocabulary size, the computed final recall attains a peak for a particular recommendation at T which is decided to be the number of topics T. Under this way, the decided number of topics in Table 3 is highlighted in bold. For example, in the case of Mylyn in LDA-KL method, the T is set at 40 when making a top-3 recommendations.

The vocabulary size plays a significant role in topic modeling [14]. The settings of vocabulary size in different projects might be different. Therefore, we make the comparison over various vocabulary sizes (i.e., from 1000 to 6000, stepping by 500). The minimum size which is set to 1000 is an empirical value which may capture a minimum set of informative words. Fig. 2 shows the recall @1, @3 and @5 of DPLSA-JS compared with LDA-KL and LDA-SVM over different vocabulary sizes in five projects. In most cases, DPLSA-JS method outperforms the LDA-KL and LDA-SVM. Also, in some cases, LDA-KL is superior to DPLSA-JS, such as in Mylyn with recall @1 when the vocabulary size is 2500. Considering the number of component recommendations, the most distinct advantage of DPLSA-JS is at one recommendation. In terms of the projects, the most distinct advantage of DPLSA-JS is in Mylyn. The most similar performance is in Platform.

For clarity of the comparison, considering the recall attained the peak over different vocabulary size, we list the recall @1, @3 and @5 of the three methods and the vocabulary size (inside the parenthesis) in Table 4. We adopt the "Improv.LDA-KL" and "Improv.LDA-SVM" to represent the improvement of our method over the LDA-KL and LDA-SVM methods, respectively. The improved values are highlighted in bold. In summary, across five projects DPLSA-JS outperforms the LDA-KL method by 30.08%, 19.60% and 14.13% for average recall @1, recall @3 and recall @5, respectively. DPLSA-JS outperforms the LDA-SVM method by 31.56%, 17.80% and 8.78% for average recall @1, recall @3 and recall @5, respectively.

For statistical tests on the performance comparison in Table 4, we conduct the Friedman test when comparing multiple models as suggested by Demšar [49]. The Friedman test compares whether the difference of the average ranks of the performance of the three methods are statistically significant or not. We translate the question into the null hypothesis H_{null}: There is no significant difference between the average ranks of the performance of the three methods. And the alternative hypothesis H_{alt} is that there is a significant difference between the average ranks of the performance of the three methods. Table 5 shows the Friedman test results. We report the average ranks (the approach with the best performance is ranked in '3') and the *p-value* for the three methods in terms of recall @1, recall @3 and recall @5. The Friedman test computes the p-value as 0.0294, 0.0224 and 0.0224 in terms of the three cases (recall @1, recall @3 and recall @5), respectively. This enables us to reject the null hypothesis (p-value < 0.05 in all the three cases) and accept the alternative hypothesis in all the three cases. It indicates that there is a statistical difference between the average ranks of the performance of the three methods in each case.

Then, we perform the Nemenyi test (significance level $\alpha = 0.05$) as a post-hoc test for each pair of the methods as suggested by Demšar [49]. In this test, the average ranks of two methods is significantly different if the average ranks differ by at least the critical difference (CD) which is computed as suggested by Demšar [49]

Table 3
Recall of LDA-KL and LDA-SVM over different number of topics when the vocabulary size is set to 5000.

Method	Projects	Recall\Topics#	10	20	30	40	50	60	70	80	90	100	110	120
LDA-KL	Mylyn	@1	0.36	0.4	0.6	**0.62**	0.11	0.49	0.09	0.08	0.08	0.08	0.08	0.09
		@3	0.55	0.68	0.75	**0.79**	0.21	0.72	0.19	0.19	0.13	0.19	0.17	0.17
		@5	0.7	**0.85**	0.79	0.79	0.21	0.83	0.21	0.21	0.23	0.21	0.19	0.21
	Gcc	@1	0.15	0.1	0.18	0.24	0.26	0.28	0.25	0.32	**0.36**	0.33	0.32	0.34
		@3	0.29	0.27	0.36	0.45	0.48	0.49	0.48	0.58	0.57	0.6	0.57	**0.61**
		@5	0.39	0.37	0.48	0.54	0.58	0.63	0.6	0.72	0.71	**0.74**	0.71	0.73
	Platform	@1	0.17	0.17	0.16	0.22	0.25	0.28	0.26	0.26	0.29	0.36	0.31	**0.38**
		@3	0.36	0.4	0.44	0.46	0.54	0.56	0.51	0.55	0.64	**0.66**	0.65	0.64
		@5	0.51	0.53	0.6	0.62	0.7	0.72	0.71	0.68	0.8	0.79	**0.81**	0.79
	Bugzilla	@1	0.13	0.14	0.17	0.16	0.16	0.18	0.19	0.24	0.24	0.24	0.24	**0.29**
		@3	0.31	0.28	0.31	0.39	0.41	0.43	0.45	0.49	0.48	0.54	0.53	**0.62**
		@5	0.44	0.44	0.51	0.54	0.58	0.57	0.63	0.66	0.64	0.69	0.7	**0.74**
	Firefox	@1	0.04	0.06	0.09	0.11	0.13	0.13	0.14	0.15	0.14	0.15	0.13	**0.17**
		@3	0.12	0.21	0.21	0.28	0.25	0.31	0.32	0.33	0.31	0.31	0.28	**0.37**
		@5	0.19	0.27	0.29	0.36	0.39	0.43	0.43	0.5	0.45	0.46	0.42	**0.52**
LDA-SVM	Mylyn	@1	0.13	0.08	0.11	0.13	0.15	0.13	0.09	0.15	0.13	0.15	**0.19**	0.19
		@3	0.19	0.25	0.3	0.19	**0.43**	0.26	0.23	0.4	0.38	0.4	0.34	0.42
		@5	0.32	0.32	0.51	0.51	0.66	0.6	**0.68**	0.62	0.62	0.64	0.64	0.66
	Gcc	@1	0.4	0.41	0.44	0.48	0.46	0.49	0.49	**0.53**	0.51	0.52	0.52	0.52
		@3	0.58	0.63	0.7	0.7	0.72	0.75	0.7	0.74	**0.76**	0.73	0.75	0.75
		@5	0.75	0.77	0.83	0.82	0.84	0.84	0.81	0.85	**0.87**	0.83	0.87	0.84
	Platform	@1	0.39	0.39	0.39	0.41	0.39	0.43	0.41	0.4	**0.44**	0.43	0.44	0.42
		@3	0.64	0.63	0.67	0.68	0.65	0.67	**0.7**	0.68	0.7	0.69	0.68	0.7
		@5	0.79	0.79	0.83	0.83	0.84	0.81	0.84	0.84	0.85	0.83	0.82	**0.85**
	Bugzilla	@1	**0.36**	0.22	0.26	0.31	0.32	0.25	0.31	0.25	0.28	0.3	0.28	0.34
		@3	0.46	0.56	0.62	0.6	0.61	0.61	0.63	0.64	0.67	0.63	0.62	**0.68**
		@5	0.66	0.71	0.75	0.77	0.74	0.78	0.78	0.81	0.78	0.76	0.8	**0.83**
	Firefox	@1	0.13	0.2	0.27	0.27	0.29	0.28	0.3	0.31	0.3	0.32	0.33	**0.34**
		@3	0.33	0.4	0.5	0.47	0.49	0.53	0.54	0.55	0.54	0.52	**0.59**	0.57
		@5	0.45	0.51	0.58	0.57	0.66	0.64	0.66	0.67	0.66	0.66	**0.69**	0.68

Table 4
Performance and the vocabulary sizes (inside the parenthesis) of DPLSA-JS, LDA-KL and LDA-SVM considering the recall attained the peak over different vocabulary sizes.

Recall @1					
Projects	DPLSA-JS	LDA-KL	**Impro.LDA-KL**	LDA-SVM	**Impro.LDA-SVM (%)**
Mylyn	0.6604(2000)	0.6604(2500)	**0.00%**	0.2264(5500)	**191.67**
Gcc	0.6560(6000)	0.3960(3500)	**65.66%**	0.5600(2500)	**17.14**
Platform	0.4777(2000)	0.4291(1000)	**11.32%**	0.4706(4500)	**1.51**
Bugzilla	0.4479(5500)	0.3229(1500)	**38.71%**	0.3948(6000)	**13.46**
Firefox	0.4102(5500)	0.2305(1500)	**77.97%**	0.3641(2500)	**12.66**
Average	0.5304	0.4078	**30.08%**	0.4032	**31.56**
Recall @3					
Mylyn	0.9245(4500)	0.8302(6000)	**11.36%**	0.5094(3000)	**81.48**
Gcc	0.8440(5000)	0.6440(2500)	**31.06%**	0.7840(4500)	**7.65**
Platform	0.7490(2000)	0.7368(3000)	**1.65%**	0.7190(5500)	**4.17**
Bugzilla	0.7500(4500)	0.6198(5000)	**21.01%**	0.7073(2000)	**6.04**
Firefox	0.6367(5500)	0.4336(3500)	**46.85%**	0.5945(3500)	**7.10**
Average	0.7808	0.6529	**19.60%**	0.6629	**17.80**
Recall @5					
Mylyn	0.9811(4500)	0.8868(2500)	**10.64%**	0.7547(3000)	**30.00**
Gcc	0.9040(4000)	0.7760(2500)	**16.49%**	0.8920(4500)	**1.35**
Platform	0.8794(5500)	0.8745(3000)	**0.57%**	0.8636(6000)	**1.84**
Bugzilla	0.8958(4000)	0.7708(3500)	**16.22%**	0.8271(5000)	**8.31**
Firefox	0.7617(5000)	0.5586(3000)	**36.36%**	0.7195(2000)	**5.86**
Average	0.8826	0.7733	**14.13%**	0.8114	**8.78**

(as the Formula 11 shows). In detail, the critical difference (CD) is a critical value which is related to the significance level α, the sample size N and the number of compared methods k. The q_α is a value which depends on α and k by referring the value table in Demšar [49] (q_α is 2.343 in this case).

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \tag{11}$$

Similar with the visualization method used by Li [50], Fig. 3 visualizes the results of the post-hoc test by the Nemenyi test. Fig. 3(a) shows the results in terms of recall @1. Since the results of recall @3 and recall @5 are the same, we use Fig. 3(b) to show the results in terms of recall @3 and recall @5. The value of the vertical axis which corresponds to the arrow denotes the index of the methods. The highest index of the vertical axis denotes the CD. The length of the CD line denotes the CD value. The value of the horizontal axis denotes the average rank of each method.

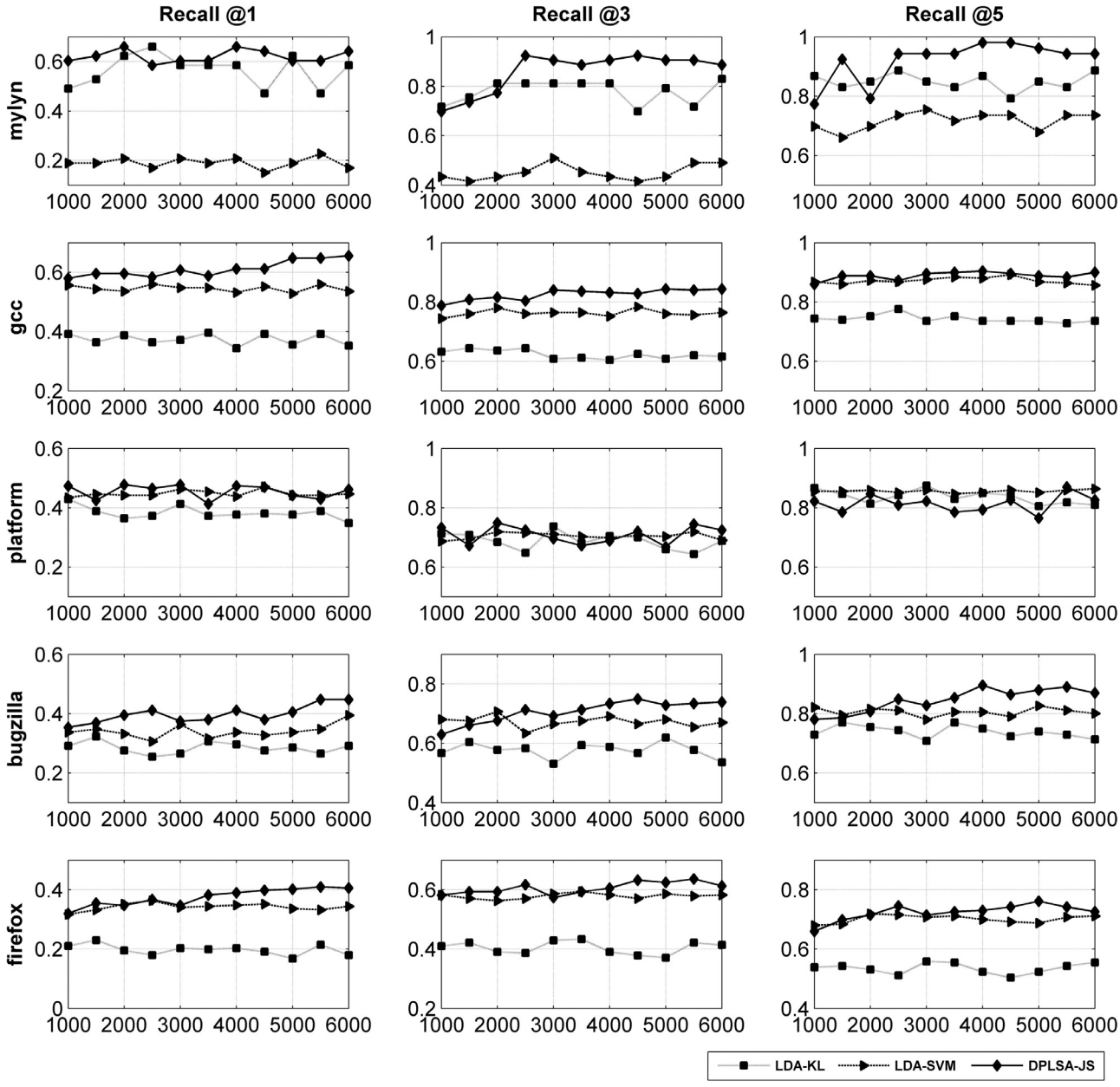


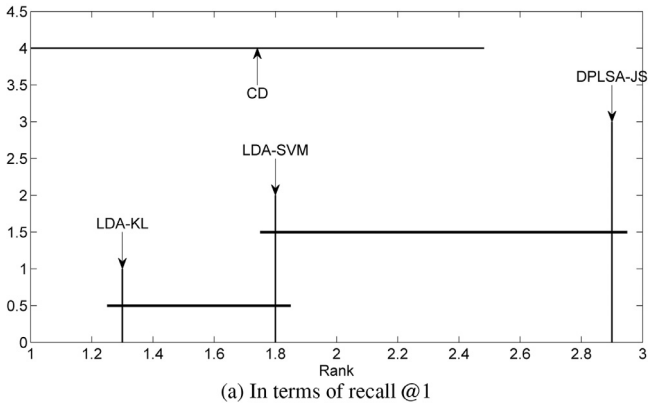
Fig. 2. Performance comparison of LDA-KL, LDA-SVM and DPLSA-JS over different vocabulary sizes.

If the difference of the average ranks of two methods is smaller than CD, then the two methods are connected as Fig. 3 shows. This means that the difference of the two connected methods is not statistically significant. We observe that there are two groups which are connected, namely (DPLSA-JS, LDA-SVM) and (LDA-SVM, LDA-KL) in both Fig. 3(a) and (b). This represents that (1) DPLSA-JS does not show a significant difference compared to LDA-SVM, and (2) DPLSA-JS is better than LDA-KL with a statistical significance. Overall, though the difference is not significant between DPLSA-JS and LDA-SVM at the current significance level, the DPLSA-JS always shows the best ranks in terms of recall @1, recall @3 and recall @5. This is also consistent with the observations in Table 4.

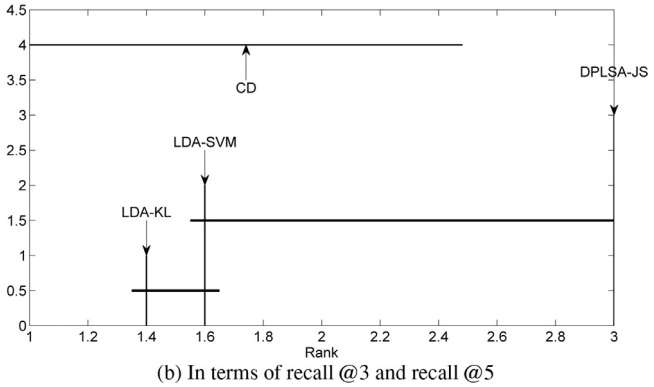
Besides, PLSA was reported to suffer from an over-fitting problem with a small amount of training data compared with LDA [46]. To evaluate the generalization performance of our DPLSA, we adopt the perplexity which is widely used in language models [14,41]. It is used to evaluate the generalization performance of the model on the unseen data set. A lower perplexity value implies a better generalization performance. Formally, it is defined as:

$$P = \exp\left[-\frac{\sum_{i,j} c(d_i, w_j) \log P(w_j|d_i)}{\sum_{i,j} c(d_i, w_j)}\right], \tag{12}$$

where $c(d_i, w_j)$ represents the w_j word counts on test set d_i. The perplexity of our DPLSA and LDA models on the test set over different vocabulary sizes is shown in Fig. 4. Since the perplexity is impacted by the number of topics, we keep the same number of topics in LDA and DPLSA, namely the number of topics is set to the number of components in each project (i.e., 18, 21, 14, 39, 55 for Platform, Bugzilla, Mylyn, Gcc. and Firefox, respectively). It shows that the generalization performance of DPLSA outperforms LDA over various vocabulary sizes in Bugzilla, Firefox and Gcc. In Mylyn and Platform, the generalization performance of DPLSA is better under a relatively large vocabulary size and is worse under a relatively small vocabulary size. It can also explain that LDA based methods are superior to DPLSA in some vocabulary size cases in the two projects. For example, in terms of recall @5 in Mylyn,



(a) In terms of recall @1



(b) In terms of recall @3 and recall @5

Fig. 3. The average ranks comparison of DPLSA-JS, LDA-KL and LDA-SVM by using the Nemenyi test. Methods that are not significantly different are connected. Fig. 3(a) shows the results in terms of recall @1. Fig. 3(b) shows the results in terms of recall @3 and recall @5.

Table 5
Friedman test for the performance comparison of DPLSA-JS, LDA-KL and LDA-SVM.

Recommendation cases	Average rank			*p*-value
	DPLSA-JS	LDA-KL	LDA-SVM	
Recall @1	2.9000	1.3000	1.8000	0.0294
Recall @3	3	1.4000	1.6000	0.0224
Recall @5	3	1.4000	1.6000	0.0224

the LDA-KL outperforms DPLSA-JS method when vocabulary size is 2000. The reason is the impact of different project's vocabulary on the topic modeling. The best settings of the vocabulary size for DPLSA and LDA are different in various projects. Therefore, we considered the recall attained the peak over different vocabulary sizes to fairly treat each method in the comparison as Table 4 shows. Overall, DPLSA shows comparable generalization performance to LDA although DPLSA may show worse at a relatively small vocabulary size.

4.5.2. Exploring the impact of comments (RQ2)

In this section, we explore the impact of comments on the recall of the recommender. To make a comparison, we create two DPLSA based component recommenders which were conducted using the same configuration as Section 3 except we change the training feature. One only uses the title and description as features while the other uses the title, description and comments as features. Fig. 5 shows the recall of the two recommenders. It is seen that the recommender with comments turns out to perform better than the recommender without comments in Mylyn, Gcc and Platform. However, the comments in Bugzilla and Firefox do not contribute to the recommendation, but also negatively impact performance in some cases.

For this statistical test, we conduct the Wilcoxon signed-rank test to analyze the significant difference when using comments or not as Demšar suggested [49]. We translate the question into the null hypothesis H_{null}: There is no significant difference between the performance using comments and not. And the alternative hypothesis H_{alt} is that there is a significant difference between the performance using comments and not. The statistical test contains 15 cases (five projects in terms of recall @1, recall @3 and recall @5) and 11 data points (vocabulary size varying from 1000 to 6000, stepping by 500) in each case. Table 6 reports the results in five projects. It shows that the computed *p*-value for Mylyn, Gcc, Platform and Bugzilla is less than 0.05. This enables us to reject the null hypothesis and accept the alternative hypothesis. It indicates that there is a statistical significance when using comments or not in these cases. In Firefox, the *p*-value indicates that we cannot reject the null hypothesis. In this case, the difference of using comments or not is not statistically significant. In terms of the performance, it is seen that the recommender using comments outperforms the recommender not using comments with statistical significance in Mylyn, Gcc and Platform. However, the same claim cannot be drawn on the other two projects. In Bugzilla, the recommender not using comments outperforms the recommender using comments with statistically significance. In Firefox, the difference is not statistically significant. Overall, our finding suggests that using comments in the DPLSA-JS recommender does not always make a contribution to the performance.

4.5.3. Exploring the impact of vocabulary size (RQ3)

The vocabulary consists of a list of unique words in the bug reports corpus. Furthermore it was normalized by removing the stop words and put into descending ordering by word frequency. This section examines the effect of varying the size of the vocabulary to the performance of DPLSA-JS. We translate the research question into the hypothesis H_{null}: There is no significant correlation between vocabulary size and the performance. And the alternative hypothesis H_{alt} is that there is a significant correlation between vocabulary size and the performance. The correlation analysis for the hypothesis contains 11 (vocabulary size varying from 1000 to 6000, stepping by 500) data points for each case.

Correlation tests were conducted to test the above hypothesis. We adopt the Spearman's rank-correlation coefficient [51] to analyze the correlation between vocabulary size and performance. The main advantage of the Spearman metric is that it does not require the variables to meet a particular distribution [52,53]. In detail, we list the Spearman correlation coefficient ρ and the *p*-value of all the cases in Table 7. To indicate the effect size of the correlations, we follow Cohen's guideline [54] that the correlation coefficient $\rho = 0.1$, 0.3, and 0.5 represent having small, medium and large effect sizes, respectively. It is seen that the cases whose correlation coefficient is smaller than 0.3 are highlighted in bold. In these cases, there is not a significant correlation between vocabulary size and performance. In four of the remaining cases, the correlations have a medium effect size. This indicates that the performance is weakly correlated with vocabulary size at a weak confidence level (less than 95%). In these cases, no significant correlation was found and the hypothesis H_{null} cannot be rejected. However, in most of the cases in Table 7, the correlations have a large effect size ($\rho > 0.5$). In these cases, the correlations are significant with at least 95% confidence (p-value < 0.05) level. This enables us to reject the H_{null} and maintain the alternative hypothesis H_{alt} in these cases. Overall, our finding suggests that the vocabulary size does matter in DPLSA-JS. Different projects need to adaptively set the vocabulary size according to an experimental method.

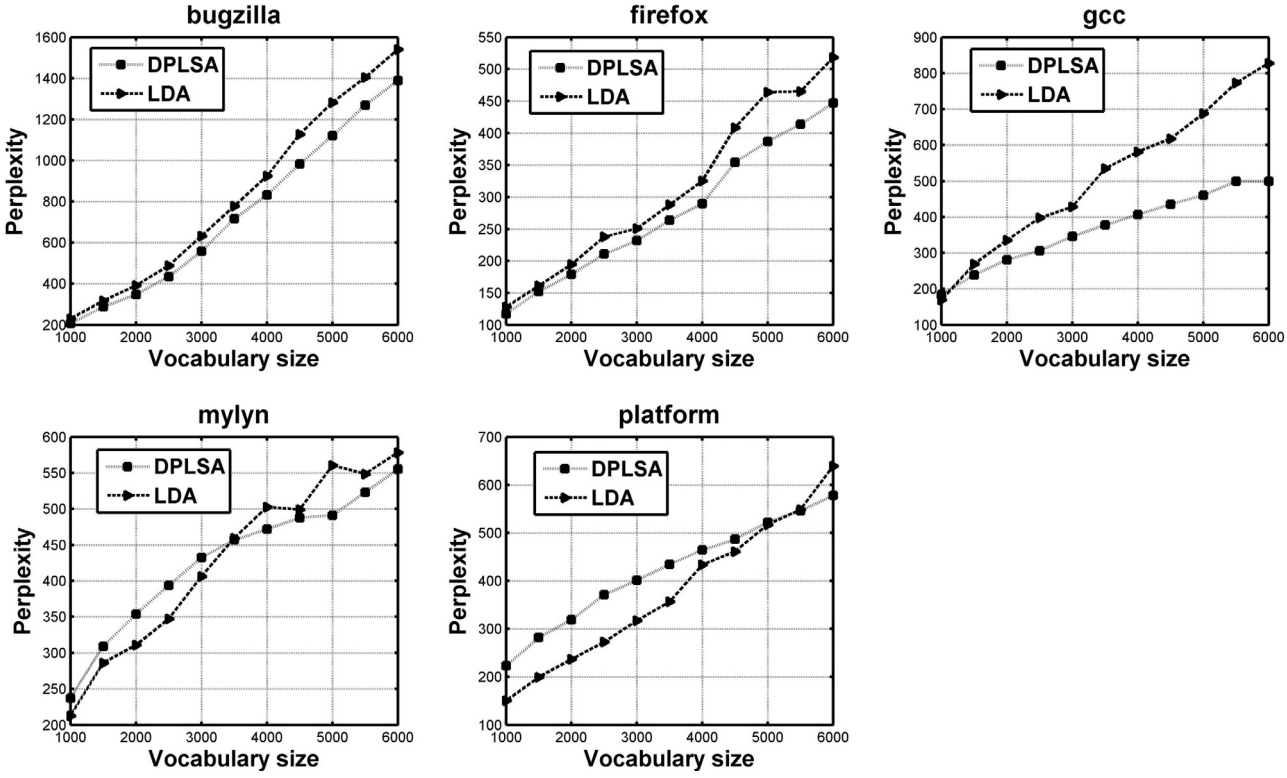


Fig. 4. Perplexity comparison of DPLSA and LDA over different vocabulary sizes.

Table 6
Wilcoxon singed-rank test for the performance comparison of recommenders without comments and with comments in five projects.

Project	*p*-value of Recall @1	*p*-value of Recall @3	*p*-value of Recall @5
Mylyn	0.0010	0.0029	0.0039
Gcc	0.0010	0.0010	0.0010
Platform	0.0010	0.0049	0.0137
Bugzilla	0.0010	0.0010	0.0010
Firefox	0.0508	0.8955	0.8135

Table 7
The Spearman coefficients with the *p*-value in different cases (*medium effect size; **large effect size).

Project	Spearman correlations (ρ(*p*-value))					
	Without comments			With comments		
	@1	@3	@5	@1	@3	@5
Mylyn	0.687**(0.020)	**0.145(0.672)**	−0.765**(0.006)	**0.120(0.726)**	0.496*(0.121)	0.683**(0.020)
Gcc	0.760**(0.007)	0.781**(0.005)	0.897**(0.000)	0.893**(0.000)	0.845**(0.001)	0.452*(0.163)
Platform	0.705**(0.015)	0.838**(0.001)	0.927**(0.000)	−0.301*(0.368)	**−0.105(0.759)**	**0.197(0.562)**
Bugzilla	0.751**(0.008)	0.900**(0.000)	0.873**(0.001)	**−0.101(0.768)**	0.879**(0.000)	0.877**(0.000)
Firefox	0.938**(0.000)	0.707**(0.015)	0.655**(0.029)	0.800**(0.005)	0.934**(0.000)	0.443*(0.173)

4.5.4. Exploring the correspondence between topics and components (RQ4)

The advantage of DPLSA is that it interprets each topic with a correspondence to components. The most expected correspondence is a one-to-one correspondence between topics and components, such that the recommendation comes down to finding the maximum topic in the topic-document distribution of a new report. However, due to the existence of semantic similarities among bug reports in different components, it is difficult to find a distinct boundary between components. A tradeoff in bug triaging, similar to this work, is to transform the one-to-one classification problem into a one-to-more (e.g., 1-3, 1-5) recommendation problem [4,34].

Fig. 6 shows the estimated $\theta_j^{d_{mean}}$ (i.e., the average document topic probability of all the training reports in a component) in each project. The horizontal axis denotes the component index for each project while the vertical axis denotes each topic. The component index and topic index are identical with the order in the initialization step. The value represents the probabilistic relevance of each component on each topic in terms of $\theta_j^{d_{mean}}$. For each topic, we fill the top 3 corresponding components with a black square. The black squares represent which components the topic can be interpreted with. It is seen that most of the entries in the diagonal are black. This can explain the discriminative power: most of the components are correlated with the identical index topic in the initialization step with a significantly high probability (within the top 3). Therefore, we conclude that there is a diagonal correspondence between the learned topics and components, namely each topic correlates with the identical index

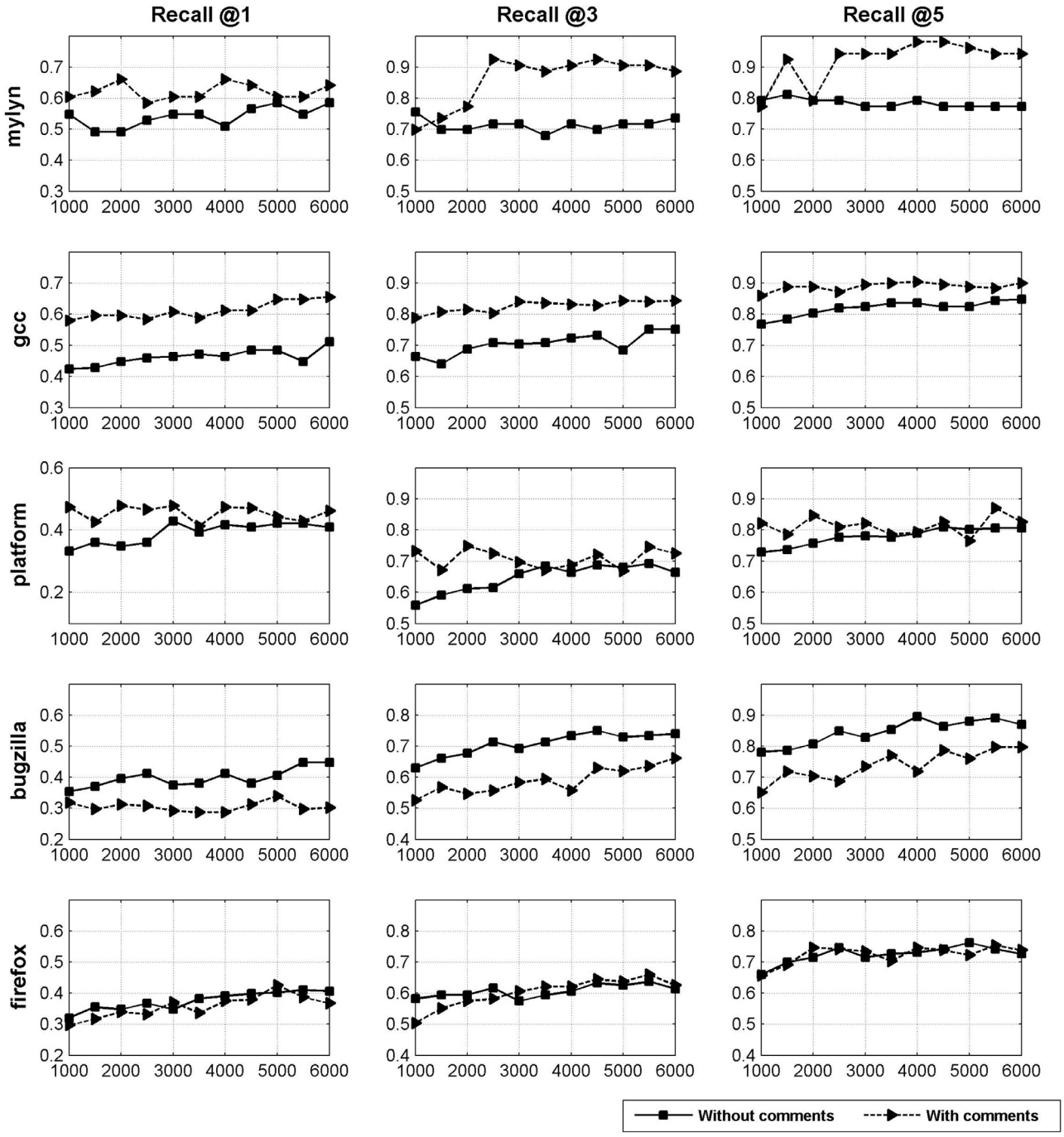


Fig. 5. The recall @1, @3 and @5 of five projects among different vocabulary sizes in a recommender without the comments and a recommender with the comments.

component. This is also the significant difference between DPLSA and LDA.

To explain the correspondence, we explore the word-topic distributions. Take the Mylyn dataset as an example, Table 8 lists the correlated words among all the topics. The number of topics is set to 14 which is the same as the number of components. The top 20 words are listed in Table 8 when the vocabulary size is equal to 5000 using DPLSA, in descending order by the relevant probability. Some informative words connected with only one topic and correlated with one component obviously. For example, 'bugzilla', 'jira', 'trac' and 'attach' (bold in Table 8) are informative words. Obviously, they are correlated with 'bugzilla', 'jira', 'trac' and 'XML' component separately. It is difficult to say there is a one-to-one correspondence between learned topics and components because of the existence of uninformative words. These words are connected with several topics with similar probability and are shared between several components. For example, 'eclipse', 'java', 'org' and 'mylyn' (underlined in Table 8) are uninformative words.

5. Threats to validity

5.1. Internal validity

Threats to internal validity result from the potential limitations in our experiments.

Impact of comments. Previous works have stated that the comments in bug reports reflect several aspects of the bug: correlations with other bugs, solutions to fix the bug, and indicators to be a duplicate bug report. In this work, experiments show that

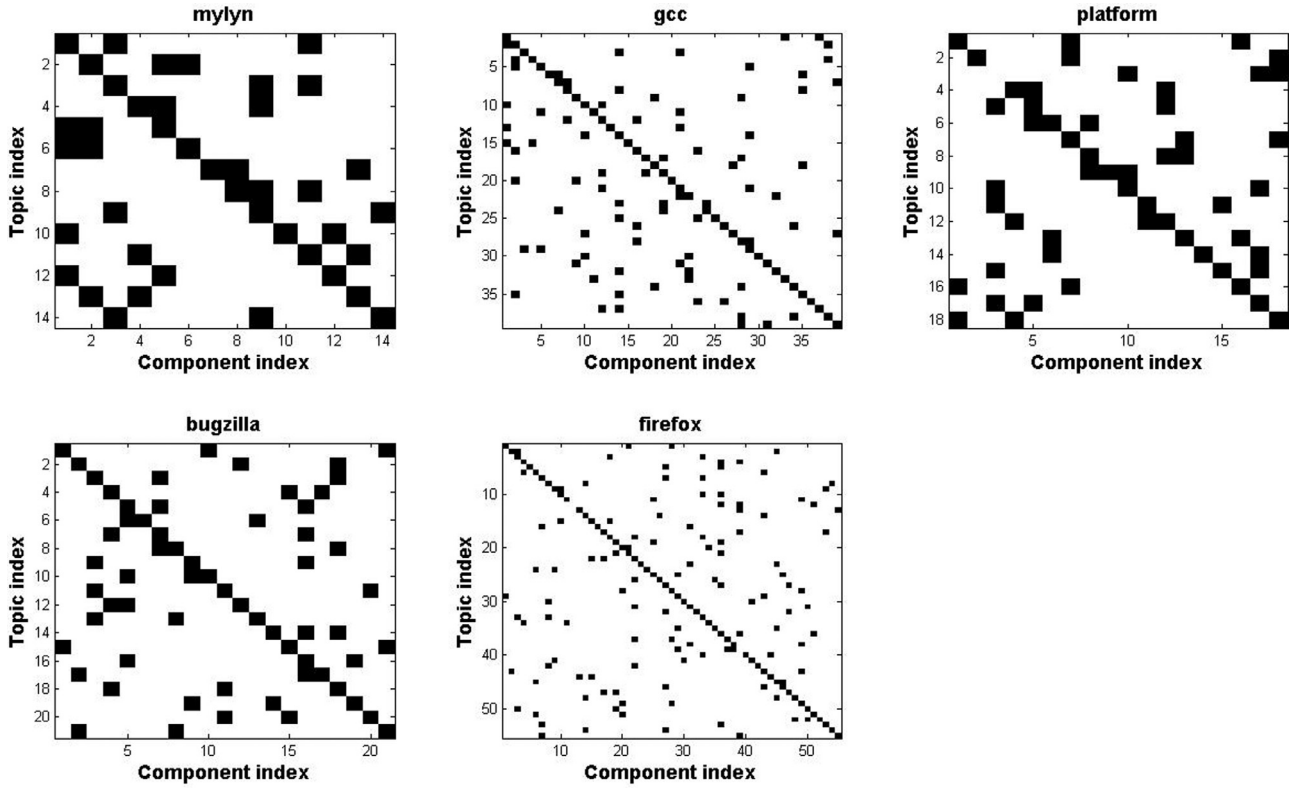


Fig. 6. The correspondence between learned topics and components. The horizontal axis denotes the component index for each project while the vertical axis denotes each topic. The value represents the probabilistic relevance of each component on each topic in terms of $\theta_j^{d_{mean}}$. For each topic, three filled black square represents the top 3 corresponding components ordered by relevance probability.

Table 8
Top 20 words in each topic in Mylyn.

Topic index	Top 20 words
1	**attach** create patch context zip *mylyn* appli fix test file update add ad code steffen check good review bug great
2	*java eclipse org* intern core osgi framework bundleload ui start defaultclassload abstractbundle lang loadclass source baseadaptor findlocalclass runtime run classload
3	editor task **trac** api comment bug method attribute connector hyperlink implement text descript id code field support repli string key
4	task queri list time synchron work date data categori bug set local change show schedul mark fix tasklist issue active
5	*eclipse org java* mylar core intern run error ui test junit framework assert task monitor log tasklist message source unknown
6	*java org eclipse* intern core run lang jface ui viewer util object actioncontributionitem wait abstracttreeview equinox thread wizarddialog method lock
7	line section screenshot comment text color expand node image layout attach fix size button viewer icon font label style space
8	**bugzilla** bug test comment field submit version error frank repli report work cgi rob fix update problem set change id
9	**jira** *mylyn* apache *java org* core service http server issue httpclient common axi client log *eclipse* connect soap intern atlassian
10	*mylyn eclipse org* update build feature http site version install project file download plug plugin release tool week dev connector
11	*java eclipse org* ui intern swt widget workbench main run display core runtime invoke reflect eclipsestart launcher adaptor eclipseapplaunch *mylyn*
12	comment make ui bug support point repli extens user project page good add code dont implement move current api report
13	task editor bug work open view select list action click show context menu make comment button set mark filter focus
14	repositori queri url dialog set connector web page wizard mylar search create task error user open properti work issue browser

using comments in some projects offer no contribution and even bring negative impacts. There are different factors which may impact the usefulness of comments, such as the person who writes the comment, the corresponded bug report and the informative degree of the content. However, we did not draw the conclusion about the detailed guideline to identify the usefulness. This is a threat for generalizing a well-supported result. We perform a statistical test and draw a converse conclusion that using comments in the DPLSA-JS component recommender does not always make contribution to the performance to mitigate this problem. A more refined work is needed on larger datasets to identify what factors and how they impact the usefulness of comments.

Impact of vocabulary. Vocabulary plays a significant role in topic modeling. In this work, the vocabulary consists of a list of unique words in the bug reports corpus. Furthermore, it was normalized by removing the stop words and put into descending order by word frequency. However, some words which possess a high frequency do not provide the discriminative information for bug assignment, such as some key words in java code (e.g., "java" and "org"). The vocabulary selection method based on the word frequencies in this work may include these words which are not discriminative enough. This is a threat to the recommendation performance. We vary the vocabulary size in a substantial range to mitigate this problem. Also, a more refined work is to take into account the effects of searching and removing these words from the vocabulary.

5.2. External validity

Threats to external validity correlate with the generalizability of our approach. When generalizing our approach outside of this study, two aspects should be noticed.

Impact of labeled bug reports. The proposed recommender relies on learning from prior bug triaging knowledge in past bug reports. Attention needs to be paid to the problem of the effectiveness of the assigned bug reports. The correctness of their provided prior knowledge is correlated with the state of each bug report. In our work, we take the same policy as [6]. Reports with a state of UNCONFIRMED or NEW are ignored. However, due to the diversity of the triaging policy in different projects, the data set needs to be further refined because of these diverse policies. Besides, the time-sequential policy used in this work determines that the selected test reports are created in the time period following the time period of the training reports. This is a threat to apply this approach to a practical environment. The recommender needs to be updated every time period to mitigate this problem.

Impact of using a supervised model. The results and conclusions are drawn through a supervised model. Using a supervised model determines that the model needs to be trained on the reports of the new system when generalizing the approach to other systems. Moreover, when a new component is added to the system, it indicates that the number of topics is changing. This is a threat for generalizing our approach. The model needs to be updated by adding sufficient reports for the new component to mitigate this problem. In the evaluation, choosing how many components to recommend is a variable. This may be a threat for the comparison, since we did not consider all the choices. However, we consider the typical choices (making 1, 3, 5 recommendations) at the number of recommendations to mitigate this problem.

6. Conclusion and future work

This paper has empirically evaluated the capability of the proposed DPLSA in creating a component recommender of bug reports and compared its performance against the state-of-art methods in the context of five open source projects. In summary, the conclusions are drawn as follows: first, using comments in the DPLSA-JS recommender does not always make a contribution to the performance. Second, the vocabulary size does matter in DPLSA-JS. Different projects need to adaptively set the vocabulary size according to the experimental method. Third, the correspondence between learned topics and components in DPLSA increases the discriminative power of topics which is useful for the recommendation task. Also, this overcomes difficulties in determining an appropriate number of topics in the LDA method. The experiments show that the DPLSA-JS outperforms the LDA-KL and LDA-SVM methods. In detail, the proposed DPLSA-JS approach outperforms the LDA-KL method by 30.08%, 19.60% and 14.13% for average recall @1, recall @3 and recall @5, respectively. Also, DPLSA-JS outperforms the LDA-SVM method by 31.56%, 17.80% and 8.78% for average recall @1, recall @3 and recall @5, respectively.

In the future, we plan to enhance the effectiveness of our method further. For example, we plan to employ various text mining techniques to adaptively choose the valid comments and terms in the vocabulary. Also, we plan to investigate whether our method outperforms other LDA based methods, such as LDA-GA [55] and sLDA [43]. Besides, we plan to test our method on more projects and bug reports and conduct a field study by interviewing developers in both industrial and open source projects.

Acknowledgments

The work described in this paper was partially supported by the National Natural Science Foundation of China (Grant nos. 91118005, 61173131), Changjiang Scholars and Innovative Research Team in University (Grant no. IRT1196), Chongqing Graduate Student Research Innovation Project (Grant no. CYS15022), and the Fundamental Research Funds for the Central Universities (Grant nos. CDJZR12098801 and CDJZR11095501).

References

[1] C.R.B. de Souza, D. Redmiles, G. Mark, J. Penix, M. Sierhuis, Management of interdependencies in collaborative software development, in: Proceedings of International Symposium on Empirical Software Engineering (ISESE), 2003, pp. 294–303.
[2] J. Anvik, G.C. Murphy, Determining implementation expertise from bug reports, in: Proceedings of Fourth International Workshop on Mining Software Repositories (MSR '07), 2007, p. 2.
[3] K. Crowston, J. Howison, H. Annabi, Information systems success in free and open source software development: theory and measures, Softw. Process: Improv. Pract. 11 (2006) 123–148.
[4] K. Somasundaram, G.C. Murphy, Automatic categorization of bug reports using latent Dirichlet allocation, in: Proceedings of the 5th India Software Engineering Conference, ACM, 2012, pp. 125–130.
[5] A. Sureka, Learning to classify bug reports into components, in: C. Furia, S. Nanz (Eds.), Objects, Models, Components, Patterns, Springer, Berlin Heidelberg, 2012, pp. 288–303.
[6] J. Anvik, G.C. Murphy, Reducing the effort of bug report triage: recommenders for development-oriented decisions, ACM Trans. Softw. Eng. Methodol. 20 (2011) 1–35.
[7] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug? in: Proceedings of the 28th International Conference on Software Engineering, ACM, 2006, pp. 361–370.
[8] C. Sun, D. Lo, X. Wang, J. Jiang, S.-C. Khoo, A discriminative model approach for accurate duplicate bug report retrieval, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ACM, 2010, pp. 45–54.
[9] S. Rastkar, G.C. Murphy, G. Murray, Summarizing software artifacts: a case study of bug reports, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ACM, 2010, pp. 505–514.
[10] I. Chawla, S.K. Singh, Performance evaluation of VSM and LSI models to determine bug reports similarity, in: Proceedings of the Sixth International Conference on Contemporary Computing, 2013, pp. 375–380.
[11] N.K. Nagwani, S. Verma, K.K. Mehta, Generating taxonomic terms for software bug classification by utilizing topic models based on Latent Dirichlet Allocation, in: Proceedings of the 11th International Conference on ICT and Knowledge Engineering (ICT&KE), 2013, pp. 1–5.
[12] X. Xia, D. Lo, X. Wang, B. Zhou, Dual analysis for recommending developers to resolve bugs, J. Softw.: Evol. Process 27 (2015) 195–220.
[13] Y. Geunseok, Z. Tao, L. Byungjeong, Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports, in: Proceedings of the IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), 2014, pp. 97–106.
[14] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.
[15] T.L. Griffiths, M. Steyvers, Finding scientific topics, in: Proceedings of the National Academy of Sciences of the United States of America, 101, 2004, pp. 5228–5235.
[16] H. Naguib, N. Narayan, B. Brugge, D. Helal, Bug report assignee recommendation using activity profiles, in: Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013, pp. 22–30.
[17] R. Shokripour, J. Anvik, Z.M. Kasirun, S. Zamani, 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, IEEE Press, 2013, pp. 2–11.
[18] X. Jifeng, J. He, R. Zhilei, Z. Weiqin, Developer prioritization in bug repositories, in: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 25–35.
[19] T. Zhang, B. Lee, A hybrid bug triage algorithm for developer recommendation, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, pp. 1088–1094.
[20] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, T.N. Nguyen, Fuzzy set and cache-based approach for bug triaging, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011, pp. 365–375.
[21] G. Salton, Automatic Text Processing, Addison-Wesley, 1989.
[22] J. Anvik, L. Hiew, G.C. Murphy, Coping with an open bug repository, in: Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange, ACM, 2005, pp. 35–39.
[23] X. Xie, W. Zhang, Y. Yang, Q. Wang, DRETOM: developer recommendation based on topic models for bug resolution, in: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, ACM, 2012, pp. 19–28.
[24] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2009, pp. 111–120.
[25] X. Xin, D. Lo, W. Xinyu, Z. Bo, Accurate developer recommendation for bug resolution, in: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 72–81.

[26] M. Linares-Vasquez, K. Hossen, D. Hoang, H. Kagdi, M. Gethers, D. Poshyvanyk, Triaging incoming change requests: Bug or commit history, or code authorship? in: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 451–460.
[27] G. Bortis, A. van der Hoek, PorchLight: a tag-based approach to bug triaging, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 342–351.
[28] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, ACM press, New York, 1999.
[29] L. Hiew, Assisted Detection of Duplicate Bug Reports, The University Of British Columbia, 2006.
[30] D. Čubranić, Automatic bug triage using text categorization, in: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE), Citeseer, 2004.
[31] L. Chen, X. Wang, C. Liu, An approach to improving bug assignment with bug tossing graphs and bug similarities, J. Softw. 6 (3) (2011) 421–427.
[32] P. Bhattacharya, I. Neamtiu, Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–10.
[33] G.A. Di Lucca, M. Di Penta, S. Gradara, An approach to classify software maintenance requests, in: Proceedings of the International Conference on Software Maintenance (ICSM), 2002, pp. 93–102.
[34] J.K. Anvik, Assisting Bug Report Triage Through Recommendation, University of British Columbia, 2007.
[35] D. Poshyvanyk, Y.G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE Trans. Softw. Eng. 33 (2007) 420–432.
[36] A. Marcus, J.I. Maletic, A. Sergeyev, Recovery of traceability links between software documentation and source code, Int. J. Softw. Eng. Knowl. Eng. 15 (2005) 811–836.
[37] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, R.A. Harshman, Indexing by latent semantic analysis, JASIS 41 (1990) 391–407.
[38] B. Dit, D. Poshyvanyk, A. Marcus, Measuring the semantic similarity of comments in bug reports, in: Proceedings of the 1st STSM'08, Citeseer, 2008.
[39] S.N. Ahsan, J. Ferzund, F. Wotawa, Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine, in: Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA), 2009, pp. 216–221.
[40] S.K. Lukins, N.A. Kraft, L.H. Etzkorn, Bug localization using latent Dirichlet allocation, Inf. Softw. Technol. 52 (2010) 972–990.
[41] T. Hofmann, Unsupervised learning by probabilistic latent semantic analysis, Mach. Learn. 42 (2001) 177–196.
[42] H.U. Asuncion, A.U. Asuncion, R.N. Taylor, Software traceability with topic modeling, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ACM, 2010, pp. 95–104.
[43] J.D. Mcauliffe, D.M. Blei, Supervised topic models, Advances in Neural Information Processing Systems, 20, MIT press, 2008, pp. 121–128.
[44] D. Ramage, D. Hall, R. Nallapati, C.D. Manning, Labeled LDA: a supervised topic model for credit attribution in multi-labeled corpora, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2009, pp. 248–256.
[45] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, J.D. Kymer, Automated classification of software change messages by semi-supervised latent Dirichlet allocation, Inf. Softw. Technol. 57 (2015) 369–377.
[46] Y. Lu, Q. Mei, C. Zhai, Investigating task performance of probabilistic topic models: an empirical study of PLSA and LDA, Inf. Retr. 14 (2011) 178–203.
[47] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, J. R. Stat. Soc. Ser. B (Methodol.) 39 (1977) 1–38.
[48] A. Majtey, P. Lamberti, D. Prato, Jensen–Shannon divergence as a measure of distinguishability between mixed quantum states, Phys. Rev. A 72 (2005) 052310.
[49] J. Demsar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[50] Y. Li, Sparse machine learning models in bioinformatics, Electronic Theses and Dissertations, Paper 5023, University of Windsor, 2014.
[51] P. Sprent, N.C. Smeeton, Applied Nonparametric Statistical Methods, CRC Press, 2007.
[52] D. Athanasiou, A. Nugroho, J. Visser, A. Zaidman, Test code quality and its relation to issue handling performance, IEEE Trans. Softw. Eng. 40 (2014) 1100–1125.
[53] S.L. Pfleeger, Experimental design and analysis in software engineering, part 5: analyzing the data, SIGSOFT Softw. Eng. Notes 20 (1995) 14–17.
[54] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, Academic press, 2013.
[55] A. Panichella, B. Dit, R. Oliveto, M.D. Penta, D. Poshyvanyk, A.D. Lucia, How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms, in: Proceedings of the International Conference on Software Engineering, IEEE Press, 2013, pp. 522–531.