Improving Automated Bug Triaging with Specialized Topic Model

Xin Xia, *Member, IEEE*, David Lo, *Member, IEEE*, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, *Member, IEEE*, and Xinyu Wang, *Member, IEEE*

Abstract—Bug triaging refers to the process of assigning a bug to the most appropriate developer to fix. It becomes more and more difficult and complicated as the size of software and the number of developers increase. In this paper, we propose a new framework for bug triaging, which maps the words in the bug reports (i.e., the term space) to their corresponding topics (i.e., the topic space). We propose a specialized topic modeling algorithm named *multi-feature topic model (MTM)* which extends Latent Dirichlet Allocation (LDA) for bug triaging. *MTM* considers product and component information of bug reports to map the term space to the topic space. Finally, we propose an incremental learning method named *TopicMiner* which considers the topic distribution of a new bug report to assign an appropriate fixer based on the affinity of the fixer to the topics. We pair *TopicMiner* with MTM (*TopicMiner*^{MTM}). We have evaluated our solution on 5 large bug report datasets including GCC, OpenOffice, Mozilla, Netbeans, and Eclipse containing a total of 227,278 bug reports. We show that *TopicMiner*^{MTM} can achieve top-1 and top-5 prediction accuracies of 0.4831-0.6868, and 0.7686-0.9084, respectively. We also compare *TopicMiner*^{MTM} with Bugzie, LDA-KL, SVM-LDA, LDA-Activity, and Yang et al.'s approach. The results show that *TopicMiner*^{MTM} on average improves top-1 and top-5 prediction accuracies of Bugzie by 128.48 and 53.22 percent, LDA-KL by 262.91 and 105.97 percent, SVM-LDA by 205.89 and 110.48 percent, LDA-Activity by 377.60 and 176.32 percent, and Yang et al.'s approach by 59.88 and 13.70 percent, respectively.

Index Terms—Developer, bug triaging, feature information, topic model

1 Introduction

Bugs appear during software development and maintenance, and bug fixing is a time-consuming and costly task. Many software projects use bug tracking systems (e.g., Bugzilla and JIRA) to manage bug reporting, bug resolution, and bug archiving processes [9]. Aside from bug description and summary information, a typical bug report records other kinds of useful information, e.g., product and component. We refer to this information as *features* of a bug report. Fig. 1 presents a bug report from Eclipse with BugID=212000.[1] In the figure, we notice that the bug report belongs to product `CDT` and component `cdt-core`.

Once a bug report is received, assigning it to a suitable developer within a short time interval can reduce the time and cost of the bug fixing process. This assignment process is known as *bug triaging* (e.g., in Fig. 1, the bug is assigned to `Oleg Krasilnikov`[2]). Bug triaging is a time-consuming process since often many developers are involved in software development and maintenance. For Eclipse and Mozilla, more than 1,800 developers participated in the bug fixing process (see Table 2). If all of the bug reports need to be manually assigned to the most appropriate developers, the bug triaging tasks would take a lot of time and effort.

To aid in finding appropriate developers, automatic bug triaging approaches have been proposed [7], [10], [20], [38]. Many of these approaches use the vector space model (VSM) to represent a bug report, i.e., a bug report is treated as a vector of terms (words) and their counts. However, developers often use various terms to express the same meaning. The same term can also carry different meanings depending on the context. These synonymous and polysemous words cannot be captured by VSM.

In the information retrieval community, topic modeling [36], which can infer the inherent latent topics of a textual document, has been used as a way to deal with synonyms and polysemy problems. A topic model converts terms in a document to topics. Two terms that are different can now be deemed similar if they are of the same topic which addresses the synonym and polysemy problems. Various topic modeling algorithms are proposed in the literature including Latent Semantic Indexing/Analysis (LSA) [16], probabilistic LSA (pLSA) [18], and Latent

1. https://bugs.eclipse.org/bugs/show_bug.cgi?id=212000

- *X. Xia and X. Wang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China. E-mail: {xxia, wangxinyu}@zju.edu.cn.*
- *D. Lo and Y. Ding are with the School of Information Systems, Singapore Management University, Singapore 17890, Singapore. E-mail: {davidlo, ying.ding.2011}@smu.edu.sg.*
- *J.M. Al-Kofahi and T.N. Nguyen are with the Electrical and Computer Engineering Department, Iowa State University, Ames, IA 50011, USA. E-mail: {jafar, tien}@iastate.edu.*

Manuscript received 18 Feb. 2015; revised 27 Apr. 2016; accepted 8 May 2016. Date of publication 6 June 2016; date of current version 21 Mar. 2017. Recommended for acceptance by M. Di Penta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2576454

2. We checked the bug assignment history and commit logs to identify `Oleg Krasilnikov` as the bug fixer.

Product: CDT **Component**: cdt-core
Version: 4.0.1 **Platform**: PC All
Severity: normal **Priority**: P3
Assigned to: Oleg Krasilnikov
Summary: ClassCastException from MBSWizardHandler.getMainPageData() if extending CDTCommonProjectWizard
Description: We have derived a Wizard from CDTCommonProjectWizard and inseted a page at the first position (before the original pages) ie. the new page will be the starzting page. Unfortunately during project creation a ClassCastException is thrown from MBSWizardHandler.getMainPageData() because it assumes that the first page is CDTMainWizardPage. The main data should be returned with some other method to allow inserting pages at the beginning.
CDTMainWizardPage page = (CDTMainWizardPage)getStartingPage();

Fig. 1. Bug report #212000 of eclipse.

Dirichlet Allocation (LDA) [12]. Among the three, LDA is the most recently proposed and it addresses the limitations of LSA and pLSA [12]. LDA considers a document as a random mixture of latent topics, where a topic is a random mixture of terms.

We extend LDA and propose a new topic model named *multi-feature topic model (MTM)* for the bug triaging problem. Since a bug report has multiple features (e.g., product affected by the bug, component affected by the bug, etc.), *MTM* considers the features of a bug report when it converts terms in the textual description of the report (i.e., texts in the summary and description fields of the report) to their corresponding topics in the topic space. Given a bug report with a particular feature combination (i.e., product-component combination), MTM converts a word in the bug report, to a topic. Similar to standard topic modelling algorithm, like Latent Dirichlet Allocation [12], the word to topic transformation is done by looking at co-occurrences of words in documents (in our case: bug reports summaries and descriptions). However, different from LDA, when converting words to topics in a bug report with a particular feature combination, MTM puts a special emphasis on the appearances of words in bug reports with the same feature combination, without ignoring the word appearances in all other bug reports. Since the number of bug reports of a particular feature combination is often limited, to infer better topics, MTM needs to also consider terms that appear in bug reports belonging to other feature combinations. MTM considers each combination of features as a random mixture of latent topics, where a topic is a random mixture of terms. MTM is an extensible topic model, where one or more features can be taken into consideration.

We refer to a feature as a categorical field in a bug report that a bug reporter can fill when the reporter submits a bug report. These fields include the product, component, reporter, priority, severity, OS, version, and platform fields. We exclude the natural language descriptions in the bug reports, which includes the contents of the summary and description fields, as the features since they are not categorical in nature. In this paper, we use the product-component combination as the input feature combination, since product and component are two of the most important features that describe a bug. Given a bug report with a particular feature combination, MTM converts a term in the bug report to a topic by putting *special emphasis* on the appearances of the word in bug reports with the same feature combination, *without ignoring* the word appearances in all other bug reports.

We propose a new approach for bug triaging which leverages MTM. We take as input a training set of bug reports (whose fixers are known) and a new bug report whose fixer is to be predicted. Our approach, named $TopicMiner^{MTM}$ computes the *affinity* of a developer to a new bug report, based on the reports that the developer fixed before. To do this, we compare the topics that appear in the new bug report with those in the old reports that the developer has fixed before.

There are a number of recent studies that are related to ours [27], [35], [38]. Tamrawi et al. propose Bugzie which recommends a list of candidate fixers that are the most relevant to a bug report [38]. Somasundaram and Murphy merge LDA with Kullback Leibler divergence and Support Vector Machine (SVM) to form LDA-KL and SVM-LDA respectively which are then used to recommend a list of components that are most relevant to a bug report in the topic space [35]. Naguib et al. propose a method which leverages LDA to recommend bug reports to developers [27]. In their approach, LDA is used to convert a bug report into topics, and a developer into topics based on their activities (i.e., based on bug reports that the developer has assigned, resolved, or reviewed in the past). A bug report is then compared to various developers in the topic space. We refer to their approach as LDA-Activity in this paper. Yang et al. propose an approach which leverage the advantages of topic modelling and the features such as product, component, severity, and priority to recommend developers [46]. We use Bugzie, LDA-KL, SVM-LDA, LDA-Activity, Yang et al.'s approach as baselines that we compare our approach with.

We evaluate our approach on 5 datasets: GCC [2], OpenOffice [5], Netbeans [4], Eclipse [1], and Mozilla [3]. In total, we analyze 227,278 bug reports. We measure the effectiveness of $TopicMiner^{MTM}$ in terms of top-1 and top-5 prediction accuracies following [10], [20], [38]. For the five datasets, $TopicMiner^{MTM}$ can achieve top-1 and top-5 prediction accuracies of up to 0.6868, and 0.9084 respectively. We compare our approach with 5 state-of-the-art approaches namely Bugzie [38], LDA-KL [35], SVM-LDA [35], LDA-Activity [27], and Yang et al.'s approach [46]. $TopicMiner^{MTM}$ on average improves top-1 and top-5 prediction accuracies of Bugzie by 128.48 and 53.22 percent, LDA-KL by 262.91 and 105.97 percent, SVM-LDA by 205.89 and 110.48 percent, LDA-Activity by 377.60 and 176.32 percent, and Yang et al.'s approach by 59.88 and 13.70 percent, respectively.

The main contributions of the paper are:

1) We propose *multi-feature topic model*, which considers bug report feature information, and we use it to create a new bug triaging approach named *TopicMiner* that leverages topic model to recommend a list of candidate fixers that are the most relevant to a bug report.
2) We experiment on a large dataset containing a total of 227,278 bug reports to demonstrate the effectiveness of $TopicMiner^{MTM}$. We show that $TopicMiner^{MTM}$

Product: CDT **Component**: cdt-core
Assigned to: Oleg Krasilnikov
Summary: [Project Model UI] selected configuration is changed when moving from one property page to another
Description: Steps to reproduce:
1. Open some project C/C++ property page → active configuration is selected/displayed
2. Select some other configuration
3. go to some other C/C++ property page → active configuration gets selected/displayed again
This is very user-unfriendly since the user would typically assume that the selected configuration remains the same when going from one page to another thus he could make changes to the wrong configuration

Fig. 2. Bug report #190823 of eclipse.

can outperform Bugzie [38], LDA-KL [35], SVM-LDA [35], and LDA-Activity [27] by a substantial margin.

The remainder of the paper is organized as follows. We describe a motivating example in Section 2. We outline our overall framework in Section 3. We present LDA and our *multi-feature topic model* in Sections 4 and 5. We present our topic-based bug triaging approach *TopicMiner* in Section 6. Our experiment results are reported in Section 7. We describe related work in Section 9. We conclude and mention future work in Section 10.

2 Preminilaries

2.1 Motivation Example

Figs. 1, 2 and 3 show three Eclipse's bug reports; they all belong to product `CDT` and component `cdt-core`, and are assigned to the same fixer `Oleg Krasilnikov`. The bug report in Fig. 1 describes a page insertion error: when a user derives a wizard from CDTCommonProjectWizard and inserts a page to an object instance of this class, an exception would be thrown if the page is inserted at the first position. The bug report in Fig. 2 describes a property page error: selected configuration changes when a user moves from one property page to another. The bug report in Fig. 3 describes a button display error: edit and delete buttons are enabled even when nothing is selected.

Observations and Implications. From the three bug reports, we can observe the following:

1) These three bug reports share some latent (i.e., hidden) commonalities, e.g., they describe user interface operations and components.
2) The textual descriptions of these three bug reports are different (i.e., the terms used in the summary and description fields are different). This makes prior VSM-based bug triaging methods [7], [38] not perform well.
3) Terms in these three bug reports could be clustered into different categories, and each category represents an aspect (i.e., a topic) of the bug reports. For example, some terms such as open, create, set, edit, etc. are user interface operations; some terms such as page, bar, button, etc. are user interface components. These two (i.e., user interface operations, and user interface components) are two common topics that are shared by the three bug reports.

Product: CDT **Component**: cdt-core
Assigned to: Oleg Krasilnikov
Summary: edit/delete buttons enabled while nothing is selected
Description: Steps To Reproduce:
1. Create a gcc project, go to Settings and select Tools > C Compiler > Symbols
2. Create 2 new symbols FOO and BAR
3. Select BAR, the Edit/Delete buttons are enabled. Delete BAR
4. The Edit/Delete button are still enabled, but FOO isn't selected. Note that the Move up/down buttons are properly disabled.
In that state, nothing bad happens when clicking Delete/Edit. Of course the same problem also occurs for include paths, but there when you press Delete without having anything selected you get a confirmation dialog asking if you really want to delete the (non existing) selected files and directories.

Fig. 3. Bug report #226562 of eclipse.

4) The developer `Oleg Krasilnikov` seems to have the expertise to fix user interface bugs in product `CDT` and component `cdt-core`.

The above observations tell us that bug reports could share some latent commonalities, and these commonalities could help to decide the right developer to assign the bug reports to. We use topic model to recover topics which represent the latent information and use them to recommend bug fixers. A topic is expressed as a collection of terms.

To compare two bug reports, rather than using the similarities of terms used in the bug reports, we use their latent commonalities, by comparing their corresponding topic distributions. This is an effective way to compare bug reports since many similar bug reports use many different words but have similar topic distributions.

2.2 Topic Modelling

By making use of a topic model such as LDA [12], we map the terms in a bug report to their corresponding latent topics. A topic is a terminology used to describe a cluster of related words. The users do not need to input topics at all as topic modeling is unsupervised, i.e., we do not need to define the name of the topics (aka. clusters) in advance. Topic modeling does not generate topic names, but the names (if desired) can be manually inferred by looking at the words that are part of the topic. Most of these topic are meaningful to a human based on our observations and previous works [12]. Topic model assumes that words in a document come from some underlying topics. We need to discover these underlying topics and the topic assignments of words. Then, we can represent a document with this discovered information instead of only words appearing in documents.

Notice a term can be assigned to multiple topics. The topic assignment of a term that appears in a document is affected by the other words in the same document. Let us assume that term w is important to both topic A and topic B. If w appears in a document with many words about A, it is likely to be assigned to topic A. If it also appears in a document with many words related to B, it will be likely to be assigned to topic B.

Fig. 4 shows an example of topics learned using our MTM. Under topic one, terms page, bar, symbol, tool, etc.

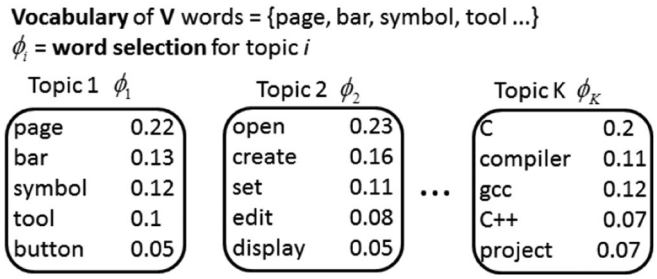


Fig. 4. Topic-word vectors.

represent user interface components. Under topic two, terms open, create, set, edit, etc. represent user interface operations. After we cluster terms into different topics, based on the proportion of terms in a bug report that belongs to different topics, we represent the bug report as a topic distribution. Fig. 5 presents an example topic distribution for the bug report shown in Fig. 3.

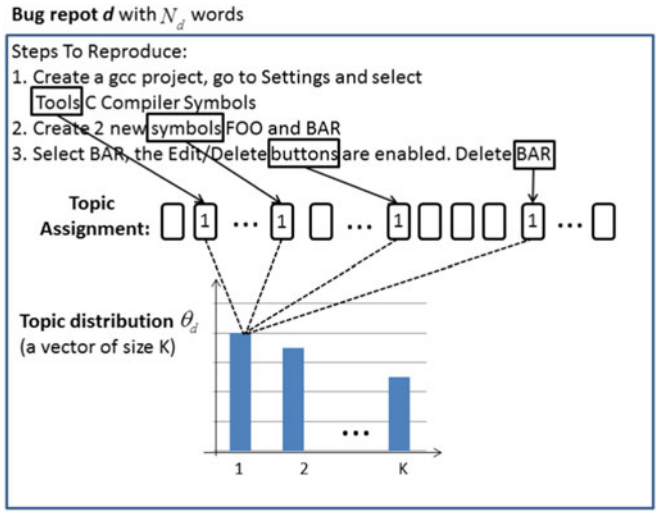


Fig. 5. Topic distribution of the bug report in Fig. 3.

3 Overall Framework

Fig. 6 presents our overall bug triaging framework that leverages topic modeling. The framework contains three phases: model construction phase, recommendation phase, and model update phase. In the model construction phase, a model is built from historical bug reports with known fixers. In the recommendation phase, the model is used to recommend a set of developers for a new unassigned bug report. In the model update phase, the model is updated by using additional bug reports with known fixers. To simulate real-life usage of our tool, we allow the model to be updated. In practice, new bug reports would be reported and assigned to fixers periodically; these new reports can be used to update the model.

Our framework first collects various information from a set of training bug reports with known fixers (Steps 1 and 2). It collects two important features from bug reports which are the components and products of the reports (Step 1).[3] Next, it extracts the description and summary texts from the reports (Step 2). We ignore any developer discussion since it is not available at the time an assignment is made. Moreover, previous studies also only use description and summary texts from the reports to recommend bug fixers [7], [10], [20], [27], [35], [38]. For each description and summary text, our framework tokenizes the text, removes stop words, stems them by using Porter stemmer [31] (i.e., reduces them to their root forms, e.g., "reads" and "reading" are reduced to "read"), and represents them in the form of a "bag of words" [25] (Step 2). These terms are in the same order as in the original bug report. Notice that the term order does not influence our model as we treat a document as a bag of words. Then, the processed text and feature information are inputted into a topic model which outputs a topic distribution for each bug report in our training data (Steps 3 and 4).[4] Next, the topic distributions and feature information are fed to *TopicMiner*.[5]

In the recommendation phase, *TopicMiner* is used to recommend a ranked list of developers to a new unassigned bug report. Our framework first extracts the features along with the summary and description text from the new bug report (Steps 6 and 7). Then, these are inputted into a topic model which outputs the topic distribution of the new report (Steps 8 and 9). Next, the topic distribution and feature information are inputted into *TopicMiner* to produce a list of top-k candidate fixers (Steps 10 and 11). In practice, a bug triager will check the list of potential fixers, and eventually assign the new bug report to a fixer. In the model update phase, we update *TopicMiner* by using the newly assigned bug report (Step 12).

4 Topic Extraction with LDA

Here, we describe how we use LDA to extract topics from bug reports.

4.1 Modeling a Bug Report

Using LDA, all unique terms (i.e., words) in bug reports are collected into a common *vocabulary Voc* of size V. A *topic* k is expressed as a collection of terms from *Voc*. LDA uses a *topic-word vector* ϕ_k of size V to represent a topic k. Each element of the vector ϕ_k represents the probability of the corresponding term in *Voc* to describe the topic.

For a bug report m containing L_m terms, LDA considers it as a textual document with K technical aspects (i.e., topics). LDA would infer the values of the following two key parameters/variables for m:

1. Topic Assignment Vector z_m. Each term in m belongs to one topic. Thus, a topic assignment vector z_m is of length L_m, and each element of z_m is an index to one topic (i.e., 1 to K).

2. Topic Distribution Vector θ_m. A bug report m could have multiple topics, and different topics have different weights in describing m. Thus, LDA assigns a bug report m a topic

3. In this paper, by default, we use product and component as the features. Other bug report fields (e.g., severity, priority, reporter, etc.) could potentially be included as features, but it is unclear if they could improve the performance further. We leave investigations of these other features as future work. Product and component are different but related fields in a bug report, thus we combine these features as a feature combination.

4. More description is available in Section 5.
5. More description is available in Section 6.

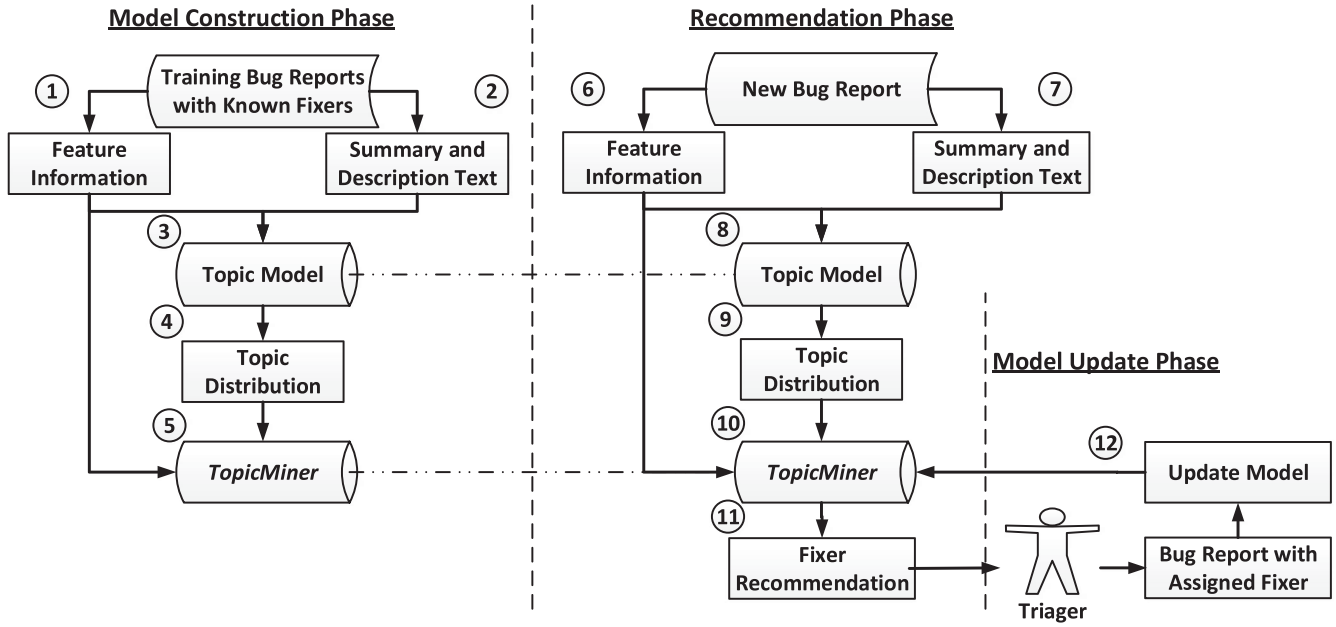


Fig. 6. Proposed bug triaging framework.

distribution vector θ_m to represent the weights of the K topics. θ_m is of length K, and each element of θ_m represents the weight of the corresponding topic. We denote the weight of topic k in θ_m as $\theta_m[k]$, and the higher $\theta_m[k]$ is, the more terms in the bug report m are assigned to topic k.

Notice a term can be assigned to multiple topics. The topic assignment of a term is affected by the other terms in the same document. Assume that word w is important to both topic `UI` and `interface operation`. If w appears in a document which is more about `UI`, it is more likely to be assigned to `UI` topic. If it appears in a document which is more about `interface operation`, it will be more likely to be assigned to `interface operation` topic.

4.2 Graphical Model and Generative Process

LDA can be represented as a graphical model, which is shown in Fig. 7. A circle represents a variable in the graphical model and a rectangle represents a variable that repeats a certain number of times. The arrows represent dependencies between variables. K refers to the number of topics which needs to be input by end users. M refers to the number of documents in the corpus. L_m refers to the number of words in the mth document. The shaded circles are observed variables, which are the words in a bug report. The other circles refer to latent variables. $w_m[n]$ refers to the nth word in the mth document. $z_m[n]$ refers to the topic of the nth word in the mth document. ϕ_k refers to topic-word vector for each topic k – there are in total K such ϕ_k. θ_m is the topic distribution vector – there are in total M such θ_m. α and β are the parameters of the Dirichlet prior for the topic distribution vector, and topic-word vector, respectively [12].

LDA is a generative probabilistic model of a textual corpus. A generative probabilistic model assumes that the words (i.e., bug reports) are generated based on a certain statistical process or model with the aforementioned sets of variables: θ_m, z_m, and ϕ_k, for each report m and each topic k. Given a bug report m of size L_m, LDA first generates its topic distribution vector θ_m according to a specific distribution (i.e., Dirichlet distribution [17]). Next, LDA randomly generates a topic for each position in m based on θ_m, i.e., it generates the topic assignment vector z_m to capture the topic of each of the L_m positions in m. Finally, for each position, a topic is randomly chosen honoring the topic distribution vector θ_m. Next, after the topic is chosen, a term (word) is chosen honoring the relevant entry in the topic-word vector ϕ_k for the chosen topic. In this way, a bug report is generated following the generative process of LDA. If one performs an experiment of generating a bag of words lots of times using the above process, on average, one would produce a bag of words where the probability of each word matches the probability of the word in the bug report.

LDA works in two phases: training and inference. In the training phase, the terms in the training bug reports are used to learn the values of the sets of variables, θ_m, z_m, and ϕ_k, which best fit the training bug reports. In the inference phase, given a new bug report new, based on the values of the sets of variables that have been learned from training bug reports, LDA infers the topics that are assigned to terms in new (i.e., z_{new}), and the topic distribution vector of new (i.e., θ_{new}). In our framework, during the model construction phase, we employ the training step of LDA; during the recommendation phase, we employ the inference step of LDA.

4.3 Algorithms

Here we describe the training phase and prediction phase of LDA in detail.

Training Phase: In the training phase, we estimate the values of the variables: z_m (topic assignment vector),

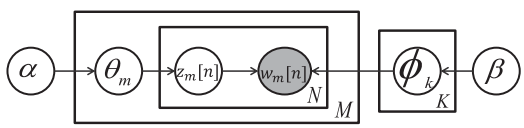


Fig. 7. The graphical model of LDA.

ϕ_k (topic-word vector), and θ_m (topic distribution vector), for each bug report m and each topic k, that best fit the bug reports in the training data. Notice that LDA only observes the words in the bug reports; thus, the optimal estimated values of these variables will be the ones that have the largest posterior probability, conditioned on the observed data (i.e., words in bug reports). Gibbs sampling is one of the solutions to estimate z_m, ϕ and θ_m [17]. Gibbs sampling is a generic procedure used to infer values of variables of a statistical model. It consists of many iterations where the estimated values of the variables are refined progressively. In each iteration, the value of each variable is estimated, one at a time, conditioned on the values of the other variables. We describe how the values of LDA's sets of variables are inferred using Gibbs sampling in the following paragraphs.

Step 1. Estimating the topic assignment vector z_m for each bug report m in the training data.

Initially, each vector z_m of a bug report m, is assigned random values. Next, the algorithm iterates many times. In each iteration, it estimates every element of z_m based on the current values of the other elements of z_m and other vectors of other bug reports in the training data. The iteration process would terminate after a large number of iterations. In this work, following [12], we set the number of iterations to 500. The number needs to be large enough so that the topic distributions are likely to converge. Similar to prior work (e.g., [12]), we do not use convergence as the stopping criteria, since the runtime may be too long if we wait for the topic distributions to fully converge (i.e., they change no further with additional iterations). We also find that there are little difference when we set the number of iterations to be more than 500 (See Section 8.7).

For each iteration, for each bug report m and each topic k, LDA estimates the probability of k being assigned to the ith position of m (i.e., $z_m[i]$). This probability (i.e., $p(z_m[i] = k)$) is computed as follows:

$$p(z_m[i] = k) = \frac{(N_m^M[-i,k] + \alpha)}{(N_m^M - 1 + K\alpha)} \times \frac{(N_k^V[-i,w_i] + \beta)}{(N_k^V - 1 + V\beta)} \quad (1)$$

In the above equation, $N_m^M[-i,k]$ is the number of words (excluding the ith word) in bug report m that are assigned to topic k; N_m^M is the number of words in bug report m; w_i is the ith word of bug report m; $N_k^V[-i,w_i]$ is the number of times the word w_i (excluding its appearance in the ith position of m) being assigned to topic k in all bug reports; N_k^V is the number of words assigned to topic k in all bug reports.

After the probability of each topic k is estimated using the above equation, the algorithm randomly chooses a topic, from the K topics, based on the estimated probabilities. The chosen topic is assigned as the topic of the ith position of m. This assignment is refined in the subsequent iterations. At the end of this step, we have a topic assignment vector z_m for every bug report m in the training data.

Step 2. Estimating the topic distribution vector θ_m for each bug report m in the training data.

Once the topic assignment vector z_m of bug report m has been computed, considering K topics, we compute its topic distribution vector θ_m based on the topics assigned to its constituent words as:

$$\begin{aligned} \theta_m &= \langle t_1, \ldots, t_K \rangle, \ where \\ t_i &= \frac{\#\ words\ assigned\ the\ i^{th}\ topic\ in\ m}{\#\ words\ in\ m} \end{aligned} \quad (2)$$

In this way, we map the words in the original bug reports into topics.

Step 3. Estimating topic-word vector ϕ_k for all K topics.

In the final step, the topic-word vector ϕ_k would be estimated for each topic k. We denote $\phi_k[i]$ as the probability of the term in the position i of *Voc* to represent topic k. $\phi_k[i]$ is estimated by computing the ratio between the number of times the ith term of *Voc* is assigned to topic k, and the total number of times terms in *Voc* are assigned to topic k.

Prediction Phase: To infer the topic distribution of a new bug report new, we input its terms, make use of the values of the sets of variables (i.e., z_m, ϕ_k, and θ_m, for each bug report m and each topic k) estimated in the training phase, and employ Gibbs sampling to iterate through the terms in the new bug report enough number of times to infer their corresponding topics. At the end of prediction phase, we get the topic distribution vector θ_{new} of the new bug report which would be processed by *TopicMiner* (see Section 6). In the next paragraphs, we first describe how z_{new} is estimated for a new bug report. Next, we describe how θ_{new} is estimated.

Step 1. Estimating the topic assignment vector z_{new} for a new bug report new.

Initially, our approach randomly assigns topics to z_{new}. Our approach then performs many (i.e., 500) iterations to refine the topics in z_{new}. For each iteration and each position i in z_{new}, given the topic assignments of bug reports in the training set and the current assignments of topics to words in the new bug reports, except for the ith word, we can compute the probability of assigning topic k to the ith word of the new bug report new as follows:

$$p(z_{new}[i] = k) = \frac{(N_{new}[-i,k] + \alpha)}{(N_{new} - 1 + K\alpha)} \times \frac{(N_k^V[-i,w_i] + \beta)}{(N_k^V - 1 + V\beta)} \quad (3)$$

In the above equation, $N_{new}[-i,k]$ is the number of words (excluding the current position i) in the new bug report new that are assigned to topic k; N_{new} is the number of words in the new bug report; w_i is the word at position i in bug report m; $N_k^V[-i,w_i]$ is the number of times the word w_i (excluding its appearance in the ith position of m) is being assigned to topic k in all bug reports; N_k^V is the number of words in all bug reports which are assigned to topic k.

After the probability of each topic k is estimated using the above equation, the algorithm randomly chooses a topic, from the K topics, based on the estimated probabilities. The chosen topic is assigned as the topic of the ith position of new. This assignment is refined in the subsequent iterations. At the end of this step, we have a topic assignment vector z_{new} for new.

Step 2. Estimating the topic distribution vector θ_{new} for the new bug report new.

To infer the topic distribution vector θ_{new}, we use Equation (2) to calculate θ_{new}, and the detailed step is the same as step 2 of the training phase.

5 Topic Extraction with MTM

Here, we describe our proposed topic model, named Multi-feature Topic Model, and how we use it to extract topics from bug reports. Notice LDA is a general topic model, which does not consider the characteristics of bug reports. Our MTM leverages multiple features of bug reports (i.e., product and component) to better generate the topics from the bug reports.

5.1 Modeling a Bug Report

All unique terms (i.e., words) in bug reports are collected into a common *vocabulary Voc* of size V. In addition to a textual description, a typical bug report contains many different fields (e.g., product and component). We refer to them as features of a bug report, and denote the features of a bug report m as F_m, i.e., $F_m = (f_{m1}, \ldots, f_{me})$. In this work, we use an instance of *MTM* with two features: product and component – we set f_{m1} to be the product, and f_{m2} to be the component.

The reason we choose these two features is that developers often specialize in some products and components. Moreover, bug reporters have to assign values to these two features when they submit a bug report (i.e., the values of these two fields are not null). Furthermore, our previous study shows that the values of these two features are stable (i.e., only a small proportion of bug reports have the values of their product and component fields reassigned before the final bug fixer is assigned) [44]. In our collected bug report dataset, we do not observe any missing values for product and component fields. In the uncommon cases where the values of the product and component fields get updated, developers can simply rerun our approach with the updated values of these fields.

A *topic* k is expressed as a collection of terms from *Voc*. MTM uses a *topic-word vector* ϕ_k of size V to represent a topic k. Each element of the vector ϕ_k represents the probability of the corresponding term in *Voc* to describe the topic. For example, in Fig. 4, for topic one, $\phi_1 = [0.22, 0.13, 0.12 \ldots]$, i.e., the probability of the term `page` to describe topic one is 22 percent while that for `bar` is 13 percent.

For a bug report m containing L_m terms, MTM considers it as a textual document with K technical aspects (i.e., topics). MTM will infer a *topic assignment vector* z_m corresponding to bug report m, i.e., for each term in m, MTM will infer its topic. Thus, a topic assignment vector z_m is of length L_m (i.e., the number of words in m), and each element of z_m is an index to one topic (i.e., 1 to K). Given a document, for each pair of term and topic, our model would output a probability that the term is related to a topic. Following [12], [17], we then assign the term to the topic which has the highest probability.

Assume that the set of all possible feature combinations together form set $F = \{F_1, F_2, \ldots, F_I\}$, where each entry in it is a feature combination and I is the total number of feature combinations. In MTM, a specific feature combination F_i (e.g., F_{i1} = `CDT` and F_{i2} = `cdt-core`) is associated to one or more topics, and each of the topics could be assigned a weight to describe the strength of its relationship with F_i. MTM assigns to a specific feature combination F_i a *feature-topic vector* θ_{F_i} to represent the weights of all of the K topics. θ_{F_i} has the length of K, and each element of θ_{F_i} represents the weight of the corresponding topic at the element's position. We denote the value of topic k in θ_{F_i} as $\theta_{F_i}[k]$, and the higher $\theta_{F_i}[k]$ is, the more terms in bug reports, whose feature combination is F_i, are assigned topic k. For example, considering the 3 bug reports in Figs. 1, 2 and 3, if the feature-topic vector is $\theta_{F_i=\{CDT,cdt-core\}} = [0.4, 0.3, \ldots 0.1]$, it means that among the 3 bug reports, which have the same product and component, 40 percent of the terms are assigned to the first topic, and 30 percent of the terms are assigned to the second topic, etc.

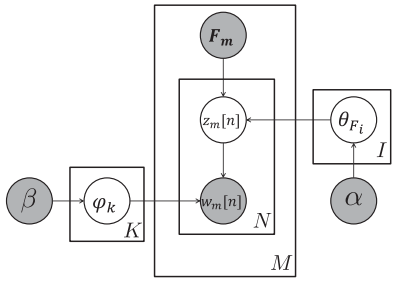


Fig. 8. The graphical model of multi-feature topic model (MTM).

5.2 Graphical Model and Generative Process

MTM can be represented as a graphical model, which is shown in Fig. 8. A circle represents a variable in the graphical model and a rectangle represents a variable that repeats a certain number of times. The arrows represent dependencies between variables. K refers to the number of topics which needs to be input by end users. M refers to the number of documents in the corpus. I refers to the number of different feature combinations. Feature combination refers to the combination of multiple features. In this paper, by default, we use the product and component features, and combine them as the feature combination. The shaded circles are observed variables, which are the words and features of a bug report. The other circles refer to latent variables. $w_m[n]$ refers to the nth word in the mth document. $z_m[n]$ refers to the topic of the nth word in the mth document. ϕ_k refers to topic-word vector for each topic k – there are in total K such ϕ_k. θ_{F_i} is the feature-topic vector – there are in total I such θ_{F_i}. α and β are the hyperparameters for the feature-topic vector, and topic-word vector, respectively. Table 1 presents the symbols associated with MTM. We use the notation of θ, ϕ, and K as the feature-topic vector, topic-word vector, and number of topics since they are conventionally used in NLP and IR literature [12].

MTM is a generative probabilistic model of a textual corpus. A generative probabilistic model assumes that the data (i.e., bug reports) is generated based on a certain process/model with the aforementioned sets of variables: θ_{F_i}, z_m, and ϕ_k. For each feature combination F_i, MTM first generates its feature-topic vector θ_{F_i} according to the Dirichlet distribution [17]. Then for each bug report m, MTM generates the topic assignment vector z_m to describe the topic of each position in m according to its feature combination. This is done by first finding the feature combination in F

TABLE 1
Symbols Associated with Multi-Feature Topic Model

Nota.	Type	Description
M	scalar	Numbers of documents (i.e., bug reports) in the document collection.
K	scalar	Numbers of topics.
V	scalar	Number of unique terms in the bug reports.
N	scalar	Number of bug reports.
I	scalar	Number of different feature combinations.
α	scalar	Dirichlet prior, hyperparameter for the topic distribution for each feature combination.
β	scalar	Dirichlet prior, hyperparameter for the word distribution for each topic.
F_m	vector	Vector representation of a feature combination for the mth bug report.
W	vector	All words in the bug reports in the document collection.
$w_m[n]$	scalar	nth word in the mth bug report.
Z	vector	Topic assignment of all words.
$z_m[n]$	scalar	Topic assignment of the nth word in the mth bug report.
ϕ_k	vector	Word distribution for topic k.
θ_{F_i}	vector	Topic distribution for feature combination F_i.

that is equal to F_m. Assume this combination is F_j, then the corresponding feature-topic vector θ_{F_j}, where $F_j = F_m$, is used to sample topics to fill topic assignment vector z_m. Finally, in each position of the bug report, MTM generates a term (word) according to the topic k assigned to this position, and the topic-word vector θ_k corresponding to topic k. In this way, a bug report is generated by leveraging MTM.

The above paragraph describes the generative process of MTM, i.e., how to generate a bug report by leveraging MTM. This generation process is simulated to infer a topic distribution vector from a bug report.

In our MTM model, we have the θ and ϕ matrices. The vector θ_{F_i} corresponds to the topic distribution for a feature combination F_i. Traditional LDA does not compute such topic distribution, rather it computes a topic distribution for each document (in our case: bug report). The vector ϕ_k of MTM contains the same information as the corresponding vector in LDAs graphical model, that is, the word distribution for the kth topic.

Our model can be regarded as a simulation of how a developer writes a bug report. For example, suppose a developer finds a bug in the `user interface` component of the product `firefox`. To create a report to describe the bug, he first picks some topics according to the component user interface and the product firefox. These topics can be composed of a topic about user interface (with words page, bar, symbol, etc.), a topic about interface operation (with words click, open, close, etc.), a topic about browser with words (Internet, website, connect, etc.), and some other topics. To write down a word, the developer needs to first determine which topic he is describing using the word, then he picks a word from this determined topic and writes it down. This process continues until he finishes the report.

These topics are what we intend to learn by using our model. Instead of assuming that each topic contains only a group of words, we assume that each topic is a distribution

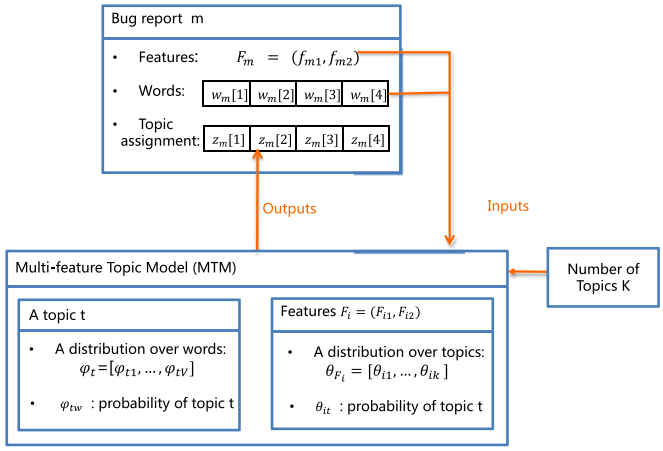


Fig. 9. Relationships among the variables in MTM.

over words. Those words with high probabilities can represent a topic better than others. Our model tries to learn these topics automatically. Intuitively, they can also be regarded as clusters of words.

In reality, the way a developer writes a bug report differs from what we assume. However, previous works on topic models have shown that they can learn meaningful grouping of words that correspond to inherent topics in documents. So, in our paper, we design a model under the context of bug triaging and apply it to real bug report dataset. Both its performance (for bug triaging) and the learned topics show that it is useful.

Fig. 9 presents the relationships of variables in our model and how our model works. For each bug report m, it is associated with a feature combination F_m and a list of words w_m. A set of bug reports is input into the Multi-feature Topic Model. There are two sets of parameters in MTM, which are topic-word vectors $\{\phi_1, \phi_2, \ldots, \phi_K\}$ and feature-topic vectors $\{\theta_{F_1}, \theta_{F_2}, \ldots, \theta_{F_M}\}$. They are all unknown and will be learned by MTM based on the input bug reports. After learning these variables, MTM is then able to assign a topic to each word, this assignment vector for bug report m is denoted as z_m.

5.3 Algorithms

MTM follows a two phase process: training phase and inference phase. The training phase and inference phase correspond to the model construction phase and recommendation phase in Fig. 6. In the training phase, the terms in the training bug reports are used to learn the values of the sets of variables, z_m, ϕ_k, and θ_{F_i}, which best fit the training bug reports. In the inference phase, given a new bug report new, based on the values of the sets of variables that have been learned from training bug reports, MTM infers the topics that are assigned to terms in new (i.e., z_{new}). In our framework, during the model construction phase, we employ the training step of MTM; during the recommendation phase, we employ the inference step of MTM.

Given a new bug report, our approach infers its topic distribution by using the topic model trained in the model construction phase. Admittedly, this strategy cannot cope with new topics that emerge over time, and we assume that the topics do not change much over time. If developers are making major changes (by implementing totally new

requirements) or the accuracy of our approach is not good, the topic model can be retrained from scratch. By doing so, the newly emerging topics would be learned. Here, we describe the training phase and prediction phase of MTM in detail.

TopicMiner takes as input a topic distribution vector. To obtain this vector θ_m, given a new bug report m, we first run MTM on m and obtain z_m by assigning the most probable topic to each word in m by using Equation (4). Similarly, we use LDA to infer z_m by assigning the most probable topic to each word in m by using Equation (1). Next, we derive θ_m from z_m and inputs it to TopicMiner. A topic distribution vector for a bug report m θ_m is of length K, and each element of θ_m is the proportion of words in m of the corresponding topic. We denote the weight of topic k (i.e., the proportion of terms in the bug report m that are assigned to topic k) in θ_m as $\theta_m[k]$. For example, in Fig. 5, if $\theta_m = [0.3, 0.2, \cdots 0.15]$, it means that 30 percent of all terms in m are about user interface components, 20 percent are about user interface operations, etc.topic assignment vector

Training Phase: In the training phase, we aim to estimate the values of the sets of variables: z_m, ϕ_k, and θ_{F_i} for each bug report m, each topic k and each feature combination F_i, that best fit bug reports in a training set. The optimal estimated values of these variables are the ones that have the largest posterior probabilities conditioned on the observed data (i.e., words and features of bug reports). We use Gibbs sampling [17] to estimate z_m, ϕ_k and θ_{F_i}. Gibbs sampling is a generic procedure used to infer values of variables of a statistical model. It consists of many iterations where the estimated values of the variables are refined progressively. In each iteration, the value of each variable is estimated, one at a time, conditioned on the values of the other variables. We describe how the values of MTM's sets of variables are inferred using Gibbs sampling in the following paragraphs.

Since the detailed derivation steps are long and complicated, they appear in a technical report [42]. In this paper, we simply present how the resultant formulas are used in the Gibbs sampling iterations.

Step 1. Estimating the topic assignment vector z_m for each bug report m with feature combination f in the training data.

Initially, the variable z_m for each bug report m and ϕ_k for each topic k, and θ_{F_i} for each feature combination F_i are all assigned with random values. Next, the algorithm iterates many times. In each iteration, it estimates every element of z_m based on the current values of the other elements of z_m and other vectors of other bug reports in the training data. The iteration process will terminate after many iterations. In this work, following [12], to ensure the convergence of topic distributions, we set the number of iterations to 500. We also find that there are little difference when we set the number of iterations more than 500 (See Section 8.7).

MTM estimates the probability of topic k being assigned to the ith position of bug report m (i.e., $z_m[i]$) with feature combination f, in each iteration, using the following equation:

$$p(z_m[i] = k) = \frac{(N_f^F[-i,k] + \alpha)}{(N_f^F - 1 + K\alpha)} \times \frac{(N_k^V[-i,w_i] + \beta)}{(N_k^V - 1 + V\beta)} \quad (4)$$

In the above equation, $N_f^F[-i,k]$ is the number of words that are assigned to topic k in bug reports with feature combination f (excluding the ith word of m); N_f^F is the number of words in bug reports with feature combination f; w_i is the ith word of m; $N_k^V[-i,w_i]$ is the number of times the word w_i (excluding its appearance in the ith position of m) is assigned to topic k; N_k^V is the number of words assigned to topic k.

After the probability of each topic k is estimated using the above equation, the algorithm randomly chooses a topic, from the K topics, based on the estimated probabilities. The chosen topic is assigned as the topic of the ith position of m. This assignment is refined in the subsequent iterations. At the end of this step, we have a topic assignment vector z_m for every bug report m in the training data.

Step 2. Estimating the feature-topic vector θ_{F_i} for each feature combination F_i.

Our approach first separates bug reports into different groups according to their feature combinations. Bug reports are in the same group if their feature combinations are the same. Under each group corresponding to a feature combination F_i, for a topic k, we denote $\theta_{F_i}[k]$ as the probability that topic k represents fi. $\theta_{F_i}[k]$ is approximately computed by computing the ratio between the number of terms which are assigned to topic k in bug reports with feature combination F_i, and the total number of terms in bug reports with feature combination F_i (i.e., the maximum likelihood estimate).

Step 3. Estimating topic-word vector ϕ_k for all K topics.

In the final step, the topic-word vector ϕ_k would be estimated for each topic k. We denote $\phi_k[i]$ as the probability of the term in position i of a common *vocabulary Voc* to represent topic k. $\phi_k[i]$ is estimated by computing the ratio between the number of times the ith term of *Voc* is assigned to topic k, and the total number of times terms in *Voc* are assigned to topic k.

Step 4. Estimating the topic distribution vector θ_m for each bug report m in the training data.

Once the topic assignment vector z_m of bug report m has been computed, considering K topics, we compute its topic distribution vector θ_m based on the topics assigned to its constituent words by using Equation (2). In this way, we map the words in the original bug reports into topics. This topic distribution vector would be processed by *TopicMiner* – as will be described in Section 6. Notice since step one of our approach produces all necessary data for Step 2 and Step 4, it is possible that these two steps can be done in parallel.

Inference Phase: To infer the topic distribution of a new bug report, we input its terms and multiple features, make use of the values of the sets of variables (i.e., z_m, ϕ_k, and θ_{F_i}) estimated in the training phase, and employ Gibbs sampling to iterate through the terms in the new bug report enough number of times to get their corresponding topics. At the end of the prediction phase, we get the topic distribution vector θ_{new} which would be processed by *TopicMiner*.

Step 1. Estimating the topic assignment vector z_{new} for a new bug report new.

Initially, we randomly assign topics to z_{new}. We then perform a number of (i.e., 500) iterations to refine the topics in z_{new}. For each iteration and each position i in z_{new}, given the topic assignments of bug reports in the training set and the current assignments of topics to words in the new bug reports, except for the ith word, we can compute the

probability of assigning topic k to the ith word of the new bug report new with feature combination f as follows:

$$p(z_{new}[i]=k)=\frac{(N_f^{F+}[-i,k]+\alpha)}{(N_f^{F+}-1+K\alpha)}\times\frac{(N_k^V[i,w_i]+\beta)}{(N_k^V-1+V\beta)} \quad (5)$$

In the above equation, $N_f^{F+}[-i,k]$ is the number of words that are assigned to topic k in the new bug report (excluding its ith word) and bug reports with feature combination f in the training set; N_f^{F+} is the number of words in the new bug report and bug reports with feature combination f in the training set; w_i is the ith word of new; $N_k^V[-i,w_i]$ is the number of times the word w_i (excluding its appearance in the ith position of new) is assigned topic k in all bug reports; N_k^V is the number of words that are assigned topic k in all bug reports.

Equations (4) and (6) are used to assign a topic to word according to the topics assignment of all other words in our dataset. Each of them is a probability of assigning a certain topic to the target word. Intuitively, the calculation of this probability is based on how likely this topic appears in the current document and how likely this topic generates the current word. Separated by the multiplication sign, there are two components in the formula. The first component is the probability of topic k appearing in the current document. The second component is the probability that word w is generated from topic k. The two parts together determine the probability of assigning topic k to the current word.

Step 2. Estimating the topic distribution vector θ_{new} for the new bug report new.

Our approach infers the topic distribution vector θ_{new} from the topic assignment vector z_{new} using Equation (2). The detailed step is the same as step 4 of the training phase of MTM.

5.4 Differences with LDA

LDA can be represented as a graphical model, which is shown in Fig. 7. Its structure is simpler than the graphical model of MTM. It has fewer observed variables which exclude features of bug reports such as the product and component of the reports. Also, rather than having a topic distribution per feature combination, it has a topic distribution per bug report θ_m.

Similar to MTM, LDA is a generative probabilistic model of a textual corpus. The generative process of MTM uses three sets of variables: topic distribution θ_m, topic assignment vector z_m, and topic-word vector ϕ_k, for each report m and each topic k. Given a bug report m of size L_m, LDA first generates its topic distribution vector θ_m according to Dirichlet distribution. Next, LDA generates the topic assignment vector z_m to describe the topic of each of the L_m positions in m according to its topic distribution vector θ_m. Finally, for each position in m, LDA generates a term (word) according to the topic k, which is assigned to the position, and the topic-word vector ϕ_k corresponding to topic k. In this way, a bug report is generated following the generative process of LDA.

Since the graphical model and generative process of LDA and MTM are different, to estimate the values of z_m, ϕ_k, and θ_{F_i} of MTM using Gibbs sampling, we need to re-derive a number of equations that eventually translate to Equation (4). A comprehensive comparison of LDA and MTM detailed derivation steps is available in the supplemental materials, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2016.2576454.

One major difference between the Gibbs sampling process of LDA and MTM is the equation that is used to update the probability of assigning a topic to a word, that is recomputed in each Gibbs sampling iteration. Rather than using Equation (4), the probability of assigning topic k to the ith word of a new bug report new is computed as follows:

$$p(z_{new}[i]=k)=\frac{(N_{new}[-i,k]+\alpha)}{(N_{new}-1+K\alpha)}\times\frac{(N_k^V[-i,w_i]+\beta)}{(N_k^V-1+V\beta)} \quad (6)$$

In the above equation, $N_{new}[-i,k]$ is the number of words (excluding the current position i) in the new bug report new that are assigned to topic k; N_{new} is the number of words in the new bug report; w_i is the word at position i in bug report m; $N_k^V[-i,w_i]$ is the number of times the word w_i (excluding its appearance in the ith position of m) is being assigned to topic k in all bug reports; N_k^V is the number of words in all bug reports which are assigned to topic k. Notice that Equation (1) above is different from Equation (4) (for MTM). For LDA, $N_m^M[-i,k]$ is used; while for MTM, $N_f^F[-i,k]$ is used. $N_f^F[-i,k]$ is used to take the feature combination into consideration when estimating the probability of a topic assignment.

Also, for a new product-component combination which is never seen before in the model building phase, our model will ignore the product-component information and behave like LDA.

6 TopicMiner: An Incremental Learning Method

In this section, we present *TopicMiner* that takes as input a set of topic distribution vectors of a set of bug reports and outputs a list of developers. Before we describe the process of how *TopicMiner* produces its outputs, we need to define a few terms, as follows:

Definition 1 (Affinity Score of a Developer to a Topic for a Particular Feature Combination). *Consider a set of topic distribution vectors T^f for a set of bug reports B with feature combination f. Let T_d^f refer to the topic distribution vectors corresponding to bug reports in B that are assigned to a fixer d. Also, given a topic distribution vector θ, let $\theta[t]$ denote an entry in the topic distribution vector θ corresponding to topic t. The affinity score $\mu_t^f(d)$ of a fixer d towards a topic t considering a feature combination f is given by:*

$$\mu_t^f(d)=\frac{\sum_{\theta\in T_d^f}\theta[t]}{\sum_{\theta'\in T^f}\theta'[t]} \quad (7)$$

Definition 2 (Affinity Score of a Developer to a Bug Report of a Particular Feature Combination). *Consider a bug report b of feature combination f with its topic distribution vector θ, and let $\theta[t]$ correspond to the entry in θ corresponding to topic t. The affinity of developer d to a bug report b*

TABLE 2
Statistics of Collected Bug Reports

Project	Time	# Reports	# Fixers	# Terms	% Generic	# Final	# Prod.	# Comp.	# Comb	# Ave.	# Ov.
GCC	2001-12-03 – 2013-04-01	27,632	280	12,895	51.23%	**13,475**	2	49	49	275	13
OpenOffice	2002-05-17 – 2013-04-07	42,243	740	17,598	11.66%	**37,318**	33	54	173	215	351
Netbeans	2008-01-01 – 2013-03-13	46,346	405	21,564	6.42%	**43,371**	38	381	459	94	200
Eclipse	2008-01-01 – 2013-03-12	82,978	1,898	32,498	26.30%	**61,156**	165	721	880	69	422
Mozilla	2009-06-23 – 2012-02-23	86,183	1,813	28,356	16.51%	**71,958**	69	690	777	92	750

of feature combination f, denoted as $\mu_b^f(d)$, is derived from the affinity of developer d to the various topics that appear in b considering feature combination f. It is given by the following equation:

$$\mu_b^f(d) = 1 - \prod_{t \in b}(1 - \mu_t^f(d) \times \theta[t]) \quad (8)$$

Here, $t \in b$ denotes that topic t is contained in bug report b (i.e., $\theta[t] > 0$). Informally put, the above formula would be very small if the bug reports that developer d fixed before share very few topics with the topics contained in bug report b. It would be large if they share a lot of common topics.

Based on the above definitions, *TopicMiner* proceeds in the following steps:

1) In the model construction phase (see Section 3), it extracts a set of topic distribution vectors of all training (historical) bug reports by using MTM. Next, it splits the bug reports into different disjoint sets according to their feature (i.e., product and component) combinations. For each feature combination f, each developer d, and each topic t, we use Equation (7) to compute the affinity score of d towards t considering feature combination f (i.e., $\mu_t^f(d)$). In this way, we have multiple $\mu_t^f(d)$ scores for the different combinations.
2) In the recommendation phase, for a new unassigned bug report b, it extracts its product and component combination f, and for each developer d, it uses Equation (8) to compute the affinity score of d towards b considering feature combination f (i.e., $\mu_b^f(d)$). Next, it sorts the developers based on their affinity scores, and recommends a list of top-k developers with the highest affinity scores.
3) In the model update phase, after a bug report of feature combination f is assigned to a developer d_{fixer}, for each topic t, it updates the affinity score of d_{fixer} towards t considering feature combination f (i.e., $\mu_t^f(d_{fixer})$) by using Equation (7).

When using TopicMiner^{MTM}, if the product and component features of a bug report changes, we can re-run the algorithm to get a better estimate of the bug reporter to be assigned to the bug report.

7 Experiments and Results

7.1 Dataset

We collect five datasets from different open source software projects: GCC, OpenOffice, Netbeans, Eclipse, and Mozilla. Table 2 shows the statistics of the five datasets that we collected. The columns correspond to the project name (Project), the time period of collected bug reports (Time), the number of collected reports (# Reports), the number of unique bug fixers (# Fixers), the number of unique terms (i.e., words) in the bug reports after we remove terms appearing less than 10 times (# Terms), the percentage of bug reports assigned to fixers with generic names (e.g., nobody, issue) (percent Generic), the number of reports without generic fixers (# Final), the number of different products (# Prod.), the number of different components (# Comp.), the number of product and component combination (# Comb.), the average number of bug reports per product- component combination (# Ave.), and the number of fixers that fix bugs across multiple product-component combinations (# Ov.), respectively. All bug reports are downloaded from the bug tracking systems of the corresponding projects. We collected bug reports with status "closed" and "fixed" following previous studies [7], [10], [20], [38].

Following prior approaches, e.g., [38], we identify bug fixers by looking at the "assigned to" fields in the bug reports. However, we notice that for many bug reports the "assigned to" fields are set to generic names which do not specify particular developers. In GCC, 51.23 percent of the bug reports are assigned to "unassigned"; In OpenOffice, 11.66 percent of the bug reports are assigned to generic names such as "issues", "needsconfirm", and "swneedsconfirm"; In Netbeans, 6.42 percent of the bug reports are assigned to "issues"; In Eclipse, 26.30 percent of the bug reports are assigned to generic names like "platform-runtime-inbox", "webmaster", "platform-text-inbox", and "AJDT-inbox"; In Mozilla, 16.51 percent of the bug reports are assigned to "nobody". Since these generic names are not actual developers, we do not recommend them, and thus they are excluded from our datasets. We record the percentage of bug reports assigned to generic names in column percent Generic in Table 2, and in column # Final Report, we record the final number of bug reports after we exclude those assigned to fixers with generic names.

In Table 2, we also list the number of different products and components in columns # Prod. and # Comp. respectively. In a bug report, the product and component fields store the product and component that are affected by the reported bug. A software system contains many products, and each product may contain many components. Eclipse has 165 products and 721 components in total as shown in Table 2.

7.2 Experiment Setup

For each bug report, we extract its bug ID, bug fixer, summary text, description text, product and component. We extract the stemmed non-stop terms (i.e., words) from the summary and description text. We exclude bug fixers who appear less than 10 times to reduce noise [7], [20], since the

expertise of these developers is hard to predict. We also remove terms which appear less than 10 times to reduce noise, and speed up the bug triaging process. These terms are put in the vectors in the same order as their appearances in the original bug report collection. Notice that the term order does not influence the outcome of our model.

To simulate the usage of our approach in practice, we use the same longitudinal data setup described in [10], [38]. The bug reports extracted from each bug repository in Table 2 are first sorted in chronological order of creation time, and then divided into 11 non-overlapping windows of equal sizes. The process proceeds as follows: First, in fold 0, we train using bug reports in frame 0, and test the trained model using the first bug report in frame 1, then we update the bug triaging model by using the first bug report, and then test using the second bug report, and update the model using the second bug report, and so on for all bug reports in frame 1. Then, in fold 1, we train using bug reports in frame 0 and frame 1, and proceed in a similar way (like frame 1) to test using bug reports in frame 2, and so on. In the final fold (fold 9), we train using bug reports in frame 0-9, and test using bug reports in frame 10. We then compute the average accuracy, which is defined as the ratio between the total number of predicted hits and the total number of test cases (i.e., the number of bug reports from frame 1-10). Following [38], we consider two criteria for a prediction hit: the fixer is identified in the top-1 list of developers (top-1 accuracy), and the fixer is identified in the top-5 list of developers (top-5 accuracy).

We use the fixers recorded in the bug repositories as the ground truth. For the training phase of MTM, following a previous study [12], we set the maximum number of iterations to 500, and the parameters α and β to $50/T$ (where T is the number of topics) and 0.1, respectively. By default, we set the number of topics T to 11 percent of the number of distinct terms in the training data, since we empirically find that TopicMiner^{MTM} achieves the best performance under this setting (see Section 7.4.2). We use percentages rather than a fixed number as the amount of training data varies for different datasets and different test frames (following the longitudinal study setup [10], [38] described above). If there are more distinct terms, there are more topics. Moreover, since both MTM and LDA use Gibbs Sampling to generate the topics which introduces randomness, we run MTM and LDA 10 times, and we compute the average performance across the 10 times.

We compare TopicMiner^{MTM} with a number of baseline approaches, i.e., Bugzie [38], LDA-KL [35], LDA-SVM [35], LDA-Activity [27], and Yang et al.'s approach [46]. For Bugzie, there are two parameters: the developer cache size and the number of descriptive terms. We use 100 percent developer cache size and set the number of descriptive terms to 10. These have been shown to result in the best performance [38]. For LDA-KL [35], SVM-LDA [35], LDA-Activity [27], and Yang et al.'s approach [46], we use JGibbsLDA (a popular implementation of LDA), and use the same settings for weights α and β, and number of topics T as *MTM*. The settings of the weights follow the settings described in the papers that introduce LDA-KL, SVM-LDA, and LDA-Activity

7.3 Research Questions

In this paper, we are interested in the following research questions:

RQ1: How accurate is TopicMiner^{MTM} as compared with other baselines?

Bugzie and LDA-Activity have been used to recommend fixers to bug reports in prior studies. LDA-KL and SVM-LDA can also be used to recommend fixers to bug reports. Yang et al. propose a approach which leverage the advantages of topic modelling and the features such as product, component, severity, and priority to recommend developers [46]. In this research question, we investigate the extent our approach (*TopicMiner^{MTM}*) outperforms these baselines. To answer this research question, we compare the top-1 and top-5 accuracies of *TopicMiner^{MTM}* with those of Bugzie, LDA-KL, SVM-LDA, LDA-Activity, and Yang et al.'s approach for the five datasets.

RQ2: What is the effect of varying the number of topics to the performance of TopicMiner^{MTM}?

MTM generates topics from a bug report collection; the number of topics needs to be manually specified. A previous study by Panichella et al. shows that different numbers of topics might affect the performance of topic models in several software engineering tasks [30]. In this research question, we investigate whether the performance of *TopicMiner^{MTM}* varies for various numbers of topics. For the other research questions, by default, the number of topics is set to be 11 percent of the number of distinct terms in a training dataset (i.e., a bug report collection).

Moreover, for other LDA-based approaches (i.e., LDA-KL, SVM-KL, LDA-Activity, and Yang et al.'s approach), by default, we also set the number of topics as 11 percent of the number of distinct terms in a training dataset. Since the number of topics may also affect the performance of these approaches, we also experiment with other numbers of topics. We vary the number of topics to be 1–15 percent of the number of distinct terms in a training dataset, and compare the performance of TopicMiner^{MTM} with LDA-KL, SVM-KL, LDA-Activity, and Yang et al.'s approach.

RQ3: What is the effect of varying the amount of training data to the performance of TopicMiner^{MTM}?

To evaluate the performance of TopicMiner^{MTM}, we use the longitudinal data setup. With the number of folds increase, the amount of the training data increase. In this research question, we investigate whether the performance of TopicMiner^{MTM} increases with the amount of training data increase. To answer this research question, we present the top-1 and top-5 accuracies for the 10 folds as shown in the experiment setup section.

RQ4: How much time does it take for TopicMiner^{MTM} to run?

The efficiency of TopicMiner^{MTM} would affect its usability. In this question, we investigate whether the runtime of *TopicMiner^{MTM}* is reasonable. To answer this research question, we investigate the average amount of time that is needed by *TopicMiner^{MTM}* and the baseline approaches to process a bug report during the model construction phase, and the average time they need to process a bug report during the recommendation and model update phases.

7.4 Results

7.4.1 RQ1: Accuracy of TopicMiner^{MTM}

Table 3 compares the performance of TopicMiner^{MTM} with the baselines in terms of top-1 and top-5 accuracies,

TABLE 3
Top-1 and Top-5 Accuracies for *TopicMiner*^{MTM}(T^M) versus Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang), Respectively (Mean±Standard Deviation)

Projects	Top-1 Accuracy					
	T^M	BZ	KL	SVM	AC	Yang
GCC	0.5048±0.0056	0.2454±0.0000	0.1334±0.0078	0.1939±0.0055	0.1585±0.0087	0.3012±0.0087
OpenOffice	0.5161±0.0023	0.2706±0.0000	0.1811±0.0039	0.2514±0.0055	0.1248±0.0102	0.3514±0.0121
Netbeans	0.6868±0.0034	0.2819±0.0000	0.1882±0.0056	0.2213±0.0013	0.1583±0.0029	0.4128±0.0045
Eclipse	0.6127±0.0051	0.2352±0.0000	0.1320±0.0012	0.1136±0.0256	0.0602±0.0210	0.4065±0.0045
Mozilla	0.4831±0.0078	0.1940±0.0000	0.1376±0.0048	0.1365±0.0033	0.0852±0.0120	0.2817±0.0145
Average.	**0.5607**	**0.2454**	**0.1545**	**0.1833**	**0.1174**	**0.3507**
Projects	**Top-5 Accuracy**					
	T^M	BZ	KL	SVM	AC	Yang
GCC	0.7864±0.0049	0.5713±0.0000	0.4071±0.0023	0.4647±0.0056	0.3947±0.0056	0.6824±0.0045
OpenOffice	0.7757±0.0072	0.5723±0.0000	0.4651±0.0038	0.4762±0.0045	0.3558±0.0125	0.6438±0.0015
Netbeans	0.9084±0.0039	0.5861±0.0000	0.4478±0.0044	0.4594±0.0035	0.3613±0.0078	0.8456±0.0231
Eclipse	0.8865±0.0043	0.5072±0.0000	0.3385±0.0018	0.2633±0.0016	0.1798±0.0023	0.8039±0.0038
Mozilla	0.7686±0.0078	0.4555±0.0000	0.3443±0.0089	0.2964±0.0056	0.2013±0.0210	0.6528±0.0067
Average.	**0.8251**	**0.5385**	**0.4006**	**0.3920**	**0.2986**	**0.7257**

respectively. From the table, we notice the improvement of our method over Bugzie, LDA-KL, SVM-LDA, LDA-Activity, and Yang et al.'s approach are substantial. Across the five projects, TopicMiner^{MTM} on average improves top-1 and top-5 prediction accuracies of Bugzie by 128.48 and 53.22 percent, LDA-KL by 262.91 and 105.97 percent, SVM-LDA by 205.89 and 110.48 percent, LDA-Activity by 377.60 and 176.32 percent, and Yang et al.'s approach by 59.88 and 13.70 percent, respectively. Notice Bugzie's result shown in Table 3 is different from the result presented in [38] since we drop bug reports assigned to generic names, e.g., nobody, issues, unassigned, etc. These generic names do not identify particular developers and must be removed to measure the effectiveness of an automated bug triaging solution.

To check if the differences in the performance of TopicMiner^{MTM} and the baseline approaches are statistically significant, for the each dataset, we apply Wilcoxon Rank Sum test [40] on the top-1 and top-5 accuracies of each pair of competing approaches. Since we run the test multiple times (twice for each dataset), we also use Bonferroni correction [6] to counteract the results of multiple comparisons. Moreover, we also compute Cliffs delta [14],[6] which is a non-parametric effect size measure that quantifies the amount of difference between the results of a pair of competing approaches. We find that in terms of top-1 and top-5 accuracies, the improvements of TopicMiner^{MTM} over the baseline approaches are all statistically significant for all of the five projects at the confidence level of 99 percent (i.e., the p-values are less than 0.001), and the effect sizes are large for all of the five projects. Thus, the improvements of TopicMiner^{MTM} over the baseline approaches are statistically significant and substantial.

We notice that the performance of TopicMiner^{MTM} is worse for the OpenOffice and Mozilla datasets than for GCC, Netbeans, and Eclipse datasets. We manually check the datasets, and find that for OpenOffice and Mozilla, more developers leave and join the communities over the period of time considered, which increases the difficulty to recommend fixers to bug reports for those datasets.

Compared with Bugzie, our TopicMiner^{MTM} recommends bug fixers by using topic distributions of bug reports instead of term (i.e., word) distribution of bug reports. Techniques that rely on term distributions suffer from synonym and polysemy problems C many words may share the same meaning, and the same word may have different meanings. Techniques that rely on topic distributions address these problems by clustering similar terms into topics, and many topics can share the same word. Thus, our TopicMiner^{MTM} can achieve a better performance than some baseline approaches, e.g., Bugzie.

Some of the baseline approaches (i.e., LDA-KL, LDA-SVM, LDA-Activity, and Yang et al.s approach) leverage LDA which is a general purpose topic modelling technique. However, bug reports are semi-structured, and they contain not only the natural language description of the bugs, but also some additional structured features such as the product and component information. By leveraging the features, one can better capture the topic distributions for bug reports by capturing both global (learned from all bug reports) and local information (learned from bug reports that share a particular product and component combination). Thus, our TopicMiner^{MTM} can achieve a better performance than the bug triaging approaches which use LDA.

Furthermore, our TopicMiner is an incremental learning approach. We update the model whenever a new bug report is assigned to developers. In this way, our model can adapt to the real-time changes from the open source community, and further improve the performance of bug trigaing.

We notice LDA-Activity does not work as well as other baseline approaches. LDA-Activity creates an activity profile for each developer, and the activity profile includes many different kinds of activities, e.g., bug reviewing, bug assignment, and bug resolution. Since our task is specific to one of these activities, considering more activities introduces noise by unnecessarily increasing the size of candidate bug fixers.

Figs. 10 and 11 present two bug reports from OpenOffice. Both of these two bug reports are in the product `porting`

6. Cliff defines a delta of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large effect size respectively.

Product: porting
Component: code
Assigned to: foskey
Summary: cannot build dmake, cannot find conf.h
Description: For many platforms, the file conf.h is called config.h. On IRIX, it is sysvr4 that is used
eg
dmake/unix/sysvr4/config.h⟹ dmake/unix/sysvr4/conf.h
This will also occur on *bsd, sysvr[1,3] and maybe some others

Fig. 10. Bug report #8108 of OpenOffice.

and component `code`, and assigned to `foskey`. Although the terms in these two bug reports are different, they both describe a configuration bug. A topic modeling based approach such as TopicMiner^{MTM} achieves a better performance than a term based approach such as Bugzie. Moreover, since LDA does not consider the specific topic distribution under different product-component combinations, these LDA based approaches do not work well for these two bug reports. We manually checked the topic distribution for these bug reports and find that their probabilities for a topic, which we manually label as `configuration`, are small. For our MTM, we consider the topic distribution for different product-component combinations. And for these two bug reports, we find the probabilities of these two bug reports for the configuration topic are the largest compared to other topics. Thus, our TopicMiner^{MTM} can recommend bug fixers better by leveraging product-component combinations.

We also notice that our TopicMiner^{MTM} does not work as well as other baseline approaches for bug reports whose product-component combinations appear fewer than five times. For example, in the fold 0 of Mozilla, there is only one bug report with product `Directory` and component `LDAP C SDK`. Our TopicMiner^{MTM} cannot recommend a suitable bug fixer for this bug report, while other approaches such as Bugzilla and LDA-SVM can recommend the fixer. Thus, we recommend users to use a general model when the number of bug reports in a specific product-component combination is small.

7.4.2 RQ2: Varying the Number of Topics.

Figs. 12, 13, 14, 15, and 16 present the top-1 and top-5 accuracies of TopicMiner^{MTM} compared with LDA-KL (KL), SVM-KL (SVM), LDA-Activity (AC), and Yang et al.'s approach (Yang) with various numbers of topics for the five datasets. We notice our TopicMiner^{MTM} shows the best

Product: porting
Component: code
Assigned to: foskey
Summary: configure cannot detect Xaw.h
Description:For FreeBSD, configure fails at:
checking for security/pam_appl.h... yes
checking whether we are using the GNU C++ compiler... yes
. . .
configure:8091: result: no
. . .
apparently, -I/usr/X11R6/include or other settings are missing which x_includes holds.

Fig. 11. Bug report #27021 of OpenOffice.

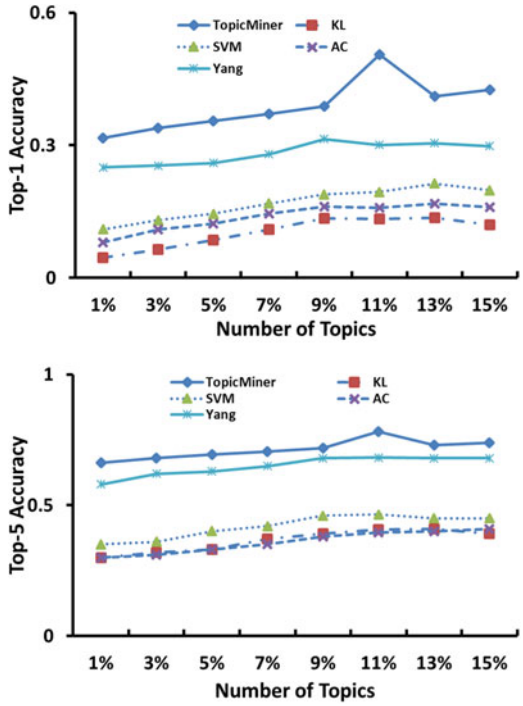


Fig. 12. Top-1 (top) and top-5 (bottom) accuracy for different numbers of topics in GCC (5 percent to 15 percent of the number of distinct terms in the training data).

performance for each number of topics. Furthermore, the performance of the best setting of each baseline approach does not outperform the best setting of TopicMiner^{MTM}.

In general, up to a certain point, the performance of TopicMiner^{MTM} increases as the number of topics increases, after that point, the performance then either remains stable or decreases. In our experiment, the number of topics

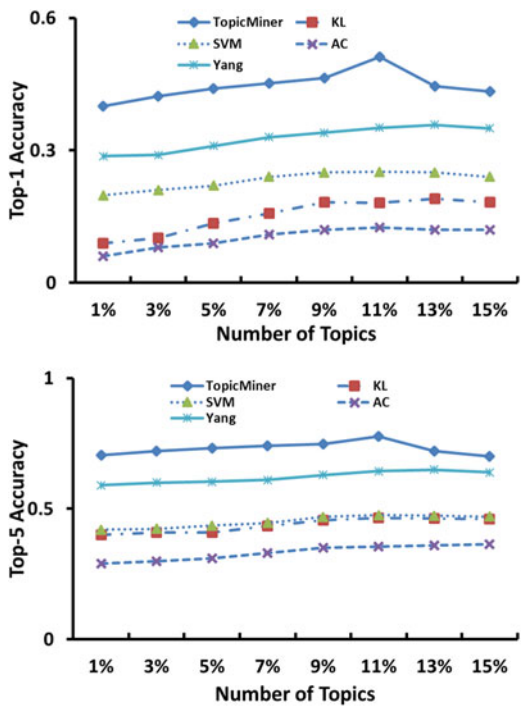


Fig. 13. Top-1 (top) and top-5 (bottom) accuracy for different numbers of topics in OpenOffice (5 percent to 15 percent of the number of distinct terms in the training data).

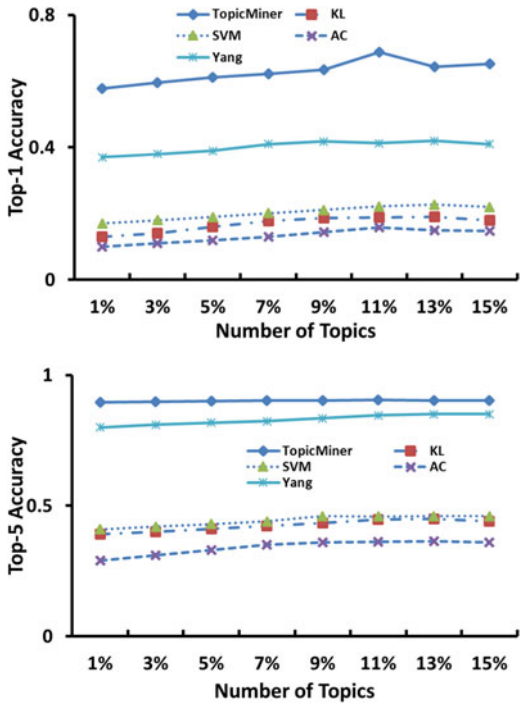


Fig. 14. Top-1 (top) and top-5 (bottom) accuracy for different numbers of topics in Netbeans (5 percent to 15 percent of the number of distinct terms in the training data).

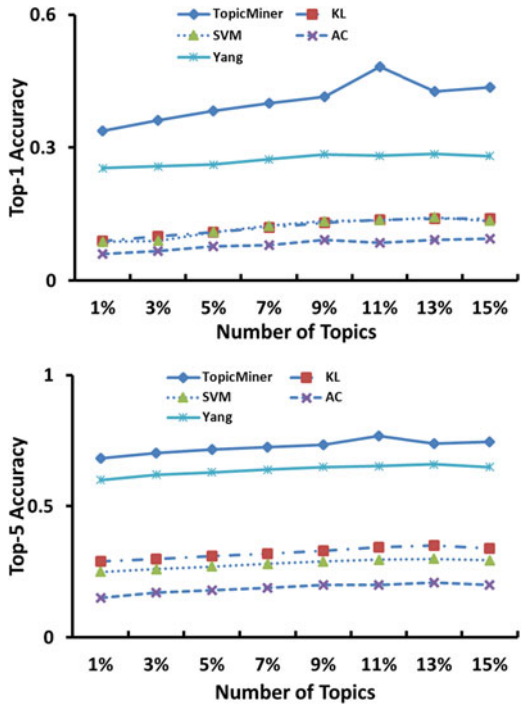


Fig. 16. Top-1 (top) and top-5 (bottom) accuracy for different numbers of topics in Mozilla (5 percent to 15 percent of the number of distinct terms in the training data).

corresponding to 11 percent of the number of distinct terms achieves the best performance. LDA-KL, SVM-KL, LDA-Activity, and Yang et al.'s approach show similar trends as TopicMiner^{MTM} when we increase the number of topics, and several of these baseline approaches achieve the best performance when the number of topics is around 11 percent of the number of distinct terms.

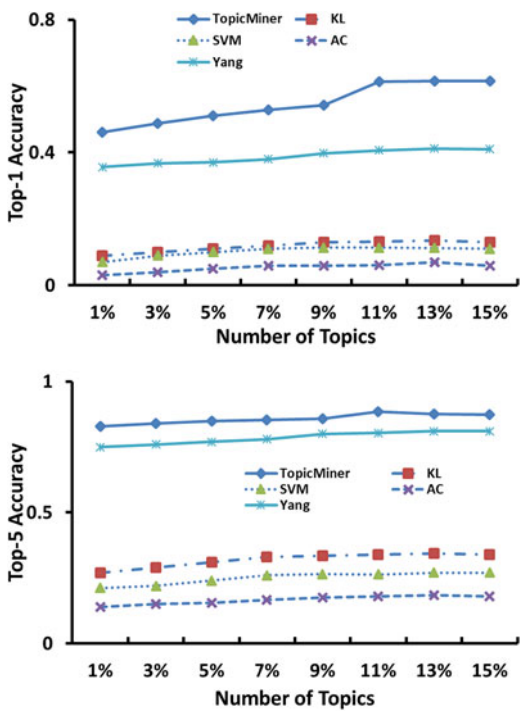


Fig. 15. Top-1 (top) and top-5 (bottom) accuracy for different numbers of topics in Eclipse (5 percent to 15 percent of the number of distinct terms in the training data).

7.4.3 RQ3: Amount of Training Data

Figs. 17, 18, 19, 20, and 21 present the top-1 and top-5 accuracies for TopicMiner^{MTM} with different amounts of training data (fold 0 -fold 9). Note that in our longitudinal data

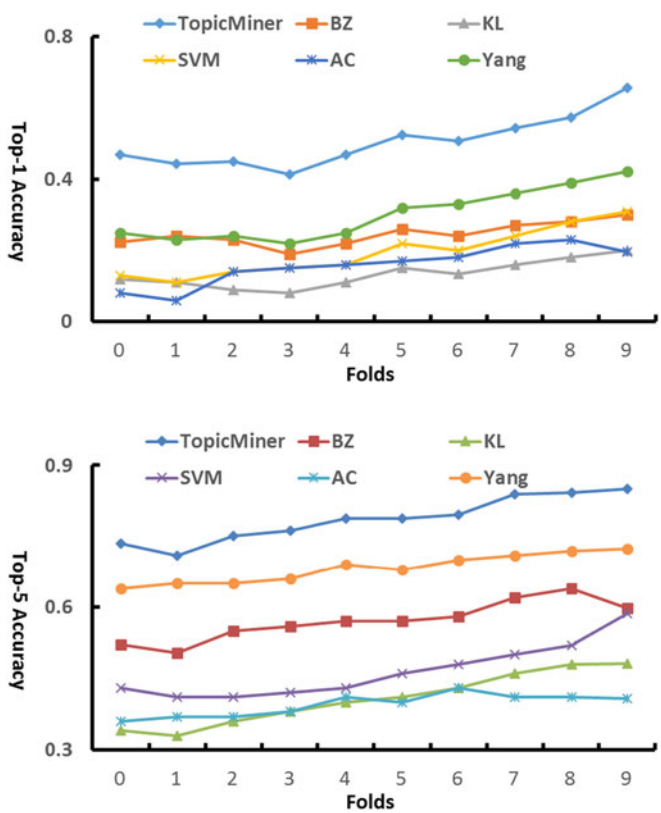


Fig. 17. Top-1 (top) and top-5 (bottom) accuracy for different folds in GCC.

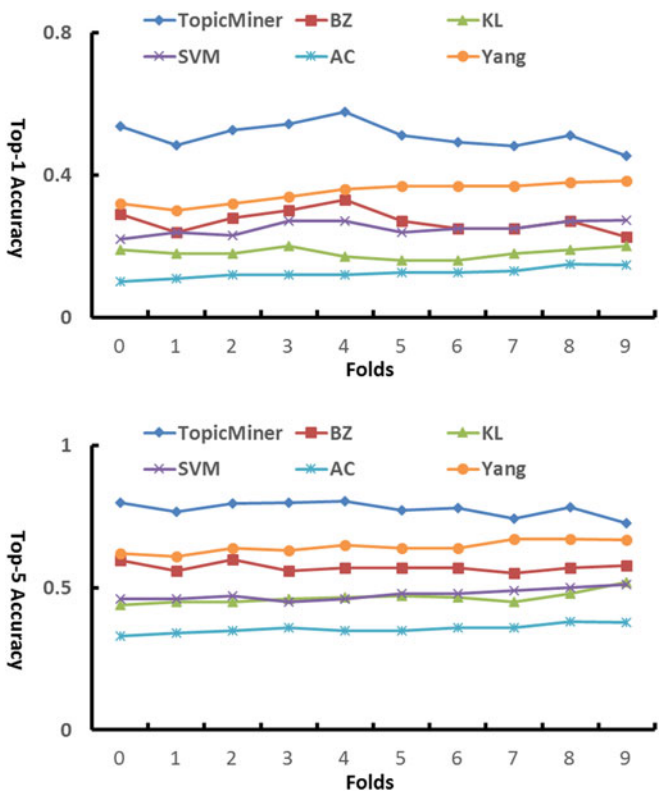


Fig. 18. Top-1 (top) and top-5 (bottom) accuracy for different folds in OpenOffice.

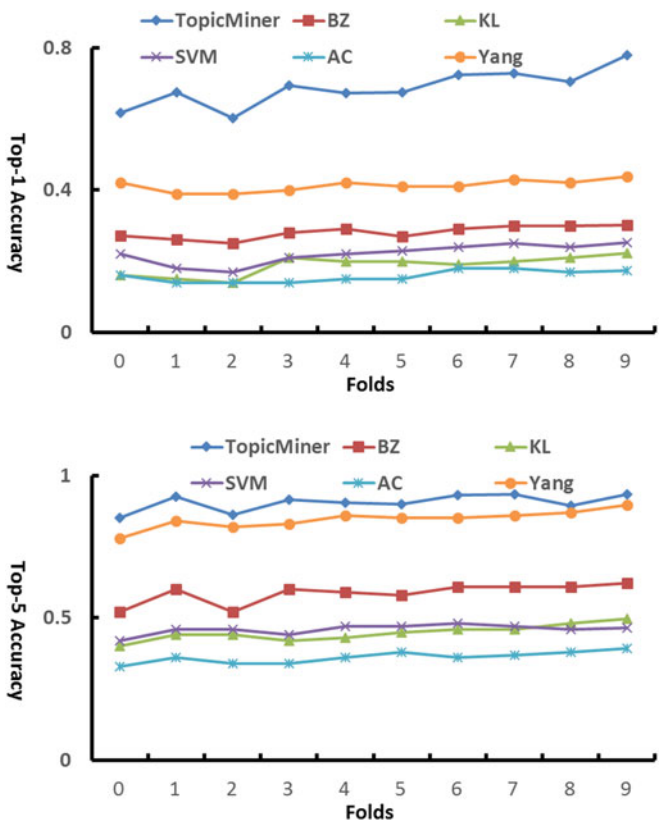


Fig. 19. Top-1 (top) and top-5 (bottom) accuracy for different folds in Netbeans.

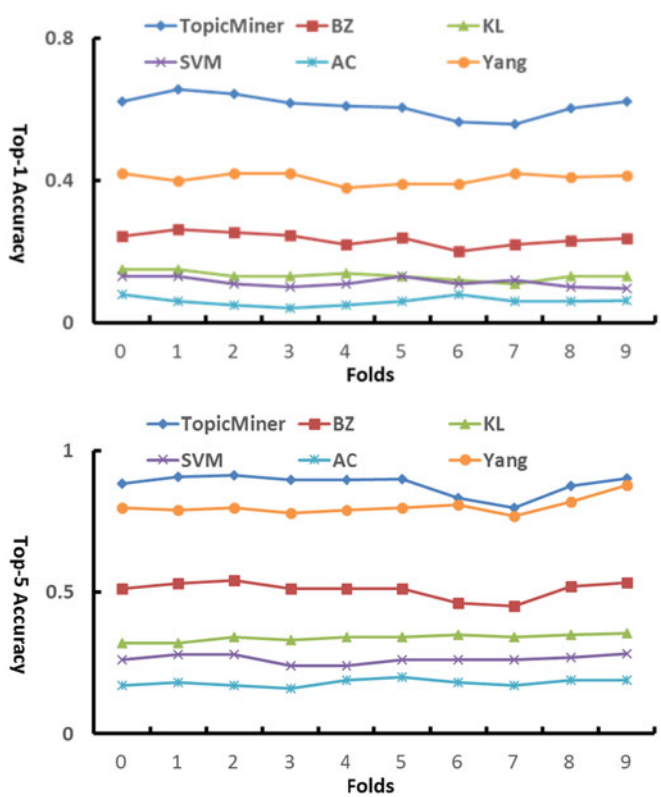


Fig. 20. Top-1 (top) and top-5 (bottom) accuracy for different folds in Eclipse.

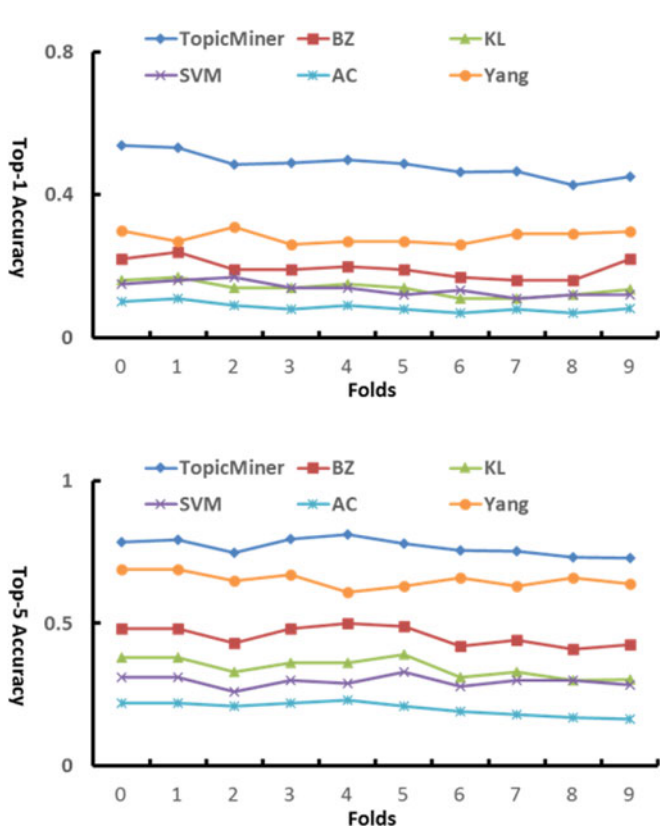


Fig. 21. Top-1 (top) and top-5 (bottom) accuracy for different folds in Mozilla.

TABLE 4
Model Construction Time, and Recommendation and Model Update Time, Across the 10 Folds, for *TopicMiner*^{MTM}(T^M), Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang) in GCC (in Minutes)

Approach	Model Construction Time										
	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total Time
T^M	2.39	4.78	7.17	9.56	11.94	14.33	16.72	19.11	21.50	23.89	131.38
BZ	0.03	0.07	0.10	0.13	0.16	0.20	0.23	0.26	0.29	0.33	1.80
KL	1.35	2.71	4.06	5.41	6.77	8.12	9.48	10.83	12.18	13.54	74.45
SVM	1.81	3.62	5.43	7.24	9.04	10.85	12.66	14.47	16.28	18.09	99.49
AC	1.35	2.70	4.05	5.40	6.75	8.10	9.45	10.80	12.15	13.50	74.22
Yang	1.63	3.27	4.90	6.53	8.17	9.80	11.43	13.07	14.70	16.33	89.83
Approach	**Model Construction Time**										
	Fold 0	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Fold 6**	**Fold 7**	**Fold 8**	**Fold 9**	**Total Time**
T^M	2.44	2.46	2.46	2.45	2.43	2.48	2.48	2.46	2.44	2.48	24.58
BZ	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.90
KL	0.35	0.33	0.33	0.33	0.32	0.37	0.37	0.37	0.33	0.38	3.48
SVM	0.39	0.38	0.39	0.41	0.39	0.39	0.41	0.39	0.39	0.39	3.93
AC	0.31	0.30	0.32	0.30	0.31	0.30	0.32	0.30	0.30	0.38	3.14
Yang	0.33	0.32	0.35	0.34	0.33	0.33	0.33	0.33	0.37	0.37	3.40

setup, we divide our data into 11 non-overlapping frames, thus one frame corresponds to 9.09 percent (1/11) of the total number of bug reports. In fold 0, the amount of training data is 9.09 percent of the total number of bug reports, and in fold 9, the amount of training data is 90.09 percent of the total number of bug reports.

From the five figures, we notice for GCC and Netbeans, in general, the performance of TopicMiner^{MTM} increases as the amount of training data increases. For OpenOffice, Eclipse and Mozilla, in general, the performance of TopicMiner^{MTM} decreases as the amount of training data increases. Also, we notice for all of the folds, the top-1 and top-5 accuracies of TopicMiner^{MTM} are much better than those of the baseline approaches.

Our collected Eclipse and Mozilla datasets are much larger than the other three datasets, which contain 82,978 and 86,183 bug reports, and 1,898 and 1,813 candidate fixers, respectively. As the amount of training data increases, the number of the candidate fixers also increases, and some fixers may leave the community, thus, the performance of TopicMiner^{MTM} decreases as the amount of training data increases for Eclipse and Mozilla. Still, the performance of TopicMiner^{MTM} are acceptable, the top-5 accuracy is above 0.7 for Eclipse and Mozilla in the 10 folds.

For OpenOffice, we notice that for several folds (such as folds 2, 5, and 9), the performance of TopicMiner^{MTM} decreases as compared with the previous folds. We manually checked the dataset, and find that in these folds, a number of new feature combinations are introduced. For example, the product-component combination "TestProduct-other" are introduced in the frame 3 (fold 2), which makes TopicMiner^{MTM} wrongly recommend fixers to bug reports belonging to this new product-component combination.

Moreover, from the two tables, we notice that the number of features (i.e product-component combinations) do not have direct impact to the performance of TopicMiner^{MTM}. For example, in Mozilla, the number of product-component combination is 777, but it achieves the lowest top-1 and top-5 accuracies compared with the other 4 datasets. And in GCC, the number of product-component combination is only 49, but its top-1 accuracy is ranked 4th and its top-5 accuracy is ranked 3rd among the 5 datasets.

7.4.4 RQ4: Time Efficiency of TopicMiner^{MTM}

Tables 4, 5, 6, 7, and 8 present the total time it takes for the 6 algorithms, i.e., TopicMiner^{MTM}, Bugzie, LDA-KL, SVM-LDA, LDA-Activity, and Yang et al.'s approach to complete the model construction phase, and the recommendation and model update phase in each of the 10 folds. We notice that the model construction time, and the prediction and model update time of TopicMiner^{MTM} are more expensive than those of the baseline approaches. However, they are still reasonable. On average for a fold, we need about 47.57 minutes and 10.26 minutes to process 18,596 bug reports during the model construction phase, and 4,132 bug reports during the prediction and model update phase, respectively. Note that the training phase can be done offline (e.g., overnight). Also, a learned model can be used to recommend fixers to many new bug reports, and updated incrementally.

8 Discussion

8.1 Stableness of TopicMiner^{MTM}

Notice our TopicMiner^{MTM} is run 10 times, and the average top-1 and top-5 accuracy scores are computed across the 10 times. Here, we would like to investigate whether the performance of TopicMiner^{MTM} would be substantially different when we run it a fewer number of times. Figs. 22, 23, 24, 25, and 26 present the top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of runs for GCC, OpenOffice, Netbeans, Eclipse, and Mozilla dataset, respectively. We notice that across the five figures, the performance of TopicMiner^{MTM} is stable, and for various number of runs, the difference in performance is small. Thus, we believe that our TopicMiner^{MTM} is a stable approach, and the randomness introduced due to Gibbs Sampling has little impact to its performance.

We also manually investigate why the performance of TopicMiner^{MTM} slightly varies (i.e., by at most 3 percent

TABLE 5
Model Construction Time, and Recommendation and Model Update Time, Across the 10 Folds, for *TopicMiner*^{MTM}(T^M), Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang) in OpenOffice (in Minutes)

Approach	Model Construction Time										
	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total Time
T^M	4.18	8.37	12.55	16.74	20.92	25.10	29.29	33.47	37.66	41.84	230.13
BZ	0.10	0.19	0.29	0.38	0.48	0.58	0.67	0.77	0.87	0.96	5.29
KL	2.43	4.85	7.28	9.70	12.13	14.55	16.98	19.41	21.83	24.26	133.41
SVM	3.43	6.86	10.30	13.73	17.16	20.59	24.02	27.46	30.89	34.32	188.77
AC	2.36	4.73	7.09	9.45	11.82	14.18	16.54	18.91	21.27	23.63	129.99
Yang	2.83	5.65	8.48	11.31	14.14	16.96	19.79	22.62	25.44	28.27	155.49
Approach	**Model Construction Time**										
	Fold 0	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Fold 6**	**Fold 7**	**Fold 8**	**Fold 9**	**Total Time**
T^M	4.83	4.84	4.83	4.85	4.85	4.86	4.88	4.89	4.89	4.89	48.61
BZ	0.36	0.38	0.37	0.36	0.38	0.36	0.38	0.37	0.37	0.38	3.71
KL	0.71	0.7	0.71	0.71	0.72	0.72	0.71	0.73	0.73	0.73	7.17
SVM	1.04	1.04	1.04	1.06	1.05	1.06	1.06	1.06	1.08	1.08	10.57
AC	0.58	0.60	0.60	0.61	0.61	0.61	0.62	0.62	0.62	0.63	6.07
Yang	0.82	0.82	0.84	0.85	0.85	0.85	0.86	0.86	0.85	0.88	8.45

from the mean) when we run it multiple times with random selections of the initial seed. We find that the topics inferred using MTM are slightly different for the multiple runs, which causes the performance of TopicMiner^{MTM} to slightly fluctuate. The topics slightly fluctuate with different seeds because of the random process involved in Gibbs sampling. To reduce the fluctuation further, it is possible to either increase the number of iterations in Gibbs sampling or to consider the generation of multiple topic models which are used to vote on the recommendation (cf., [11]). We leave the exploration of these options as future work.

Moreover, we notice that the top-5 accuracies of TopicMiner^{MTM} are higher than the corresponding top-1 accuracies; this is the case since top-1 accuracy (i.e., the proportion of bug reports for which the first recommended fixer is the actual fixer) is a stricter evaluation metric, than top-5 accuracy (i.e., the proportion of bug reports for which *one of the first five* recommended fixers is the actual fixer).

8.2 Effect of Training Using the Last Frame

In our previous section, we use the same longitudinal data setup described in Section 7.2. Shokripour et al. find that training a prediction model by using the whole history of a software system may cause a loss of performance [34]. Here, we would like to investigate whether it is the same case for bug triaging. To perform such investigation, we adapt the experiment setting described in Section 7.2, such that in each fold, we reduce the training set to include only the last frame. For example, in fold 9, we only use the bug reports in frame 9 to build the prediction model, and test using bug reports in frame 10.

Table 9 compares the performance of TopicMiner^{MTM} with the baselines in terms of top-1 and top-5 accuracies, respectively. We notice still our TopicMiner^{MTM} shows substantial improvement over the baseline approaches. Moreover, we notice that TopicMiner^{MTM} using only the last frame achieves a slightly better performance than TopicMiner^{MTM} using all of the historical bug reports. On average across the 5 projects,

TABLE 6
Model Construction Time, and Recommendation and Model Update Time, Across the 10 Folds, for *TopicMiner*^{MTM}(T^M), Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang) in Netbeans (in Minutes)

Approach	Model Construction Time										
	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total Time
T^M	7.01	14.01	21.02	28.02	35.03	42.03	49.04	56.04	63.05	70.05	385.28
BZ	0.10	0.20	0.30	0.39	0.49	0.59	0.69	0.79	0.89	0.99	5.42
KL	3.73	7.45	11.18	14.90	18.63	22.36	26.08	29.81	33.53	37.26	204.93
SVM	4.63	9.25	13.88	18.50	23.13	27.76	32.38	37.01	41.64	46.26	254.44
AC	3.87	7.74	11.61	15.48	19.35	23.22	27.09	30.96	34.83	38.71	212.88
Yang	4.27	8.54	12.81	17.09	21.36	25.63	29.90	34.17	38.44	42.71	234.93
Approach	**Model Construction Time**										
	Fold 0	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Fold 6**	**Fold 7**	**Fold 8**	**Fold 9**	**Total Time**
T^M	7.20	7.20	7.20	7.21	7.21	7.20	7.22	7.22	7.22	7.22	72.10
BZ	0.69	0.69	0.70	0.70	0.70	0.70	0.71	0.71	0.71	0.71	7.02
KL	0.89	0.89	0.89	0.89	0.90	0.90	0.90	0.90	0.91	0.91	8.98
SVM	1.09	1.09	1.10	1.10	1.10	1.10	1.11	1.11	1.11	1.12	11.03
AC	0.82	0.82	0.82	0.82	0.84	0.84	0.84	0.84	0.84	0.84	8.32
Yang	0.99	1.00	1.00	1.00	1.02	1.02	1.02	1.02	1.02	1.02	10.11

TABLE 7
Model Construction Time, and Recommendation and Model Update Time, Across the 10 Folds, for *TopicMiner*^{MTM}(T^M), Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang) in Eclipse (in Minutes)

Approach	Model Construction Time										
	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total Time
T^M	18.53	37.06	55.60	74.13	92.66	111.19	129.72	148.26	166.79	185.32	1,019.27
BZ	0.21	0.43	0.64	0.85	1.07	1.28	1.49	1.70	1.92	2.13	11.72
KL	9.65	19.29	28.94	38.58	48.23	57.88	67.52	77.17	86.81	96.46	530.53
SVM	14.70	29.39	44.09	58.78	73.48	88.18	102.87	117.57	132.26	146.96	808.28
AC	9.40	18.79	28.19	37.58	46.98	56.37	65.77	75.17	84.56	93.96	516.77
Yang	11.58	23.17	34.75	46.33	57.91	69.50	81.08	92.66	104.24	115.83	637.04
Approach	**Model Construction Time**										
	Fold 0	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Fold 6**	**Fold 7**	**Fold 8**	**Fold 9**	**Total Time**
T^M	21.78	21.78	21.78	21.81	21.81	21.81	21.81	21.81	21.82	21.82	218.03
BZ	2.87	2.87	2.87	2.87	2.87	2.90	2.90	2.90	2.90	2.90	28.85
KL	2.58	2.58	2.58	2.58	2.58	2.60	2.60	2.60	2.60	2.60	25.90
SVM	3.02	3.02	3.02	3.03	3.03	3.03	3.03	3.04	3.04	3.04	30.30
AC	2.37	2.37	2.37	2.37	2.39	2.39	2.39	2.39	2.41	2.41	23.86
Yang	2.78	2.78	2.78	2.80	2.80	2.80	2.80	2.80	2.83	2.83	28.00

the top-1 and top-5 accuracies for TopicMiner^{MTM} trained using only the last frame are 0.5864 and 0.8313, while these scores for TopicMiner^{MTM} using all of the historical bug reports are 0.5607 and 0.8251, respectively.

8.3 Impact of Different Product-Component Combinations

Considering some product-component combinations have more bug reports, while some product-component combinations have less, we also check whether there is any fall off in performance for product-component combinations with fewer bug reports. Tables 10 and 11 present the first 5 product-component combinations which appear at least 10 times, and the top 5 product-component combinations which appear the most in OpenOffice and Netbeans datasets, respectively. The columns correspond to the name of the product (Product), name of the component (Component), number of the times that the product-component combination appears in our collected data (# Comb.), top-1 accuracy for TopicMiner^{MTM} for bug reports that fall under the product-component combination (Top-1), and top-5 accuracy for TopicMiner^{MTM} (Top-5).

From Tables 10 and 11, we notice in general, as the number of bugs in the product-component combination increases, the top-1 and top-5 accuracies also increase. For example, for NetBeans dataset, the top-1 and top-5 accuracies for the product-component combination "serverpluginCode" are 0.20 and 0.50, while the top-1 and top-5 accuracies for the product-component combination "projects-maven" are 0.73 and 0.98.

8.4 TopicMiner^{MTM} versus TopicMiner^{LDA}

TopicMiner can be paired with various topic models. To further validate the benefit of our new topic model MTM, we pair TopicMiner with LDA (*TopicMiner*^{LDA}) and compare its performance with *TopicMiner*^{MTM}. To pair TopicMiner with LDA, we simply modify the first step of TopicMiner, described in Section 6, to use LDA instead of MTM. Notice

TABLE 8
Model Construction Time, and Recommendation and Model Update Time, Across the 10 Folds, for *TopicMiner*^{MTM}(T^M), Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang) in Mozilla (in Minutes)

Approach	Model Construction Time										
	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total Time
T^M	11.30	22.59	33.89	45.18	56.48	67.77	79.07	90.36	101.66	112.95	621.24
BZ	0.24	0.48	0.72	0.96	1.20	1.44	1.68	1.92	2.16	2.40	13.19
KL	6.33	12.67	19.00	25.34	31.67	38.01	44.34	50.68	57.01	63.34	348.40
SVM	8.55	17.10	25.64	34.19	42.74	51.29	59.83	68.38	76.93	85.48	470.13
AC	6.06	12.12	18.19	24.25	30.31	36.37	42.43	48.50	54.56	60.62	333.41
Yang	7.41	14.83	22.24	29.66	37.07	44.48	51.90	59.31	66.72	74.14	407.76
Approach	**Model Construction Time**										
	Fold 0	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Fold 6**	**Fold 7**	**Fold 8**	**Fold 9**	**Total Time**
T^M	14.94	14.94	14.94	14.97	14.97	14.97	14.97	14.99	14.99	14.99	149.67
BZ	2.66	2.66	2.67	2.67	2.67	2.69	2.69	2.69	2.69	2.69	26.78
KL	1.92	1.92	1.92	1.94	1.94	1.94	1.94	1.94	1.94	1.94	19.34
SVM	2.34	2.34	2.34	2.35	2.35	2.35	2.35	2.38	2.38	2.38	23.56
AC	1.56	1.56	1.56	1.58	1.58	1.58	1.58	1.58	1.58	1.58	15.74
Yang	1.96	1.96	1.96	1.96	1.97	1.97	1.97	1.99	1.99	1.99	19.72

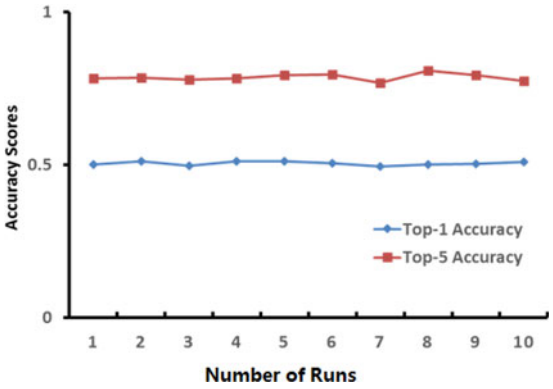


Fig. 22. Average top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of runs applied to GCC dataset.

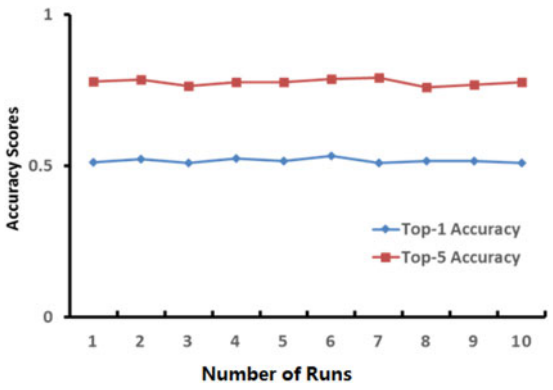


Fig. 23. Average top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of runs applied to OpenOffice dataset.

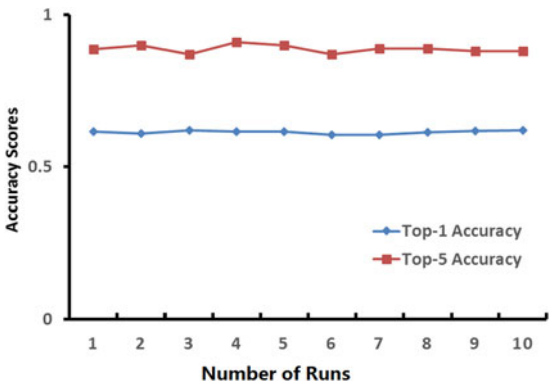


Fig. 24. Average top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of runs applied to Netbeans dataset.

LDA does not use the product-component combination information, so we leave them out when training and using LDA. After we get the topic distributions by leveraging LDA, we then input the topic distributions with the product-component combination information into TopicMiner. TopicMiner considers both the topic distributions and product-component combination to recommend developers.

Table 12 presents the top-1 and top-5 accuracies for *TopicMiner*^{MTM} and *TopicMiner*^{LDA}. We notice that *TopicMiner*^{MTM} achieves better performance than *TopicMiner*^{LDA}. Across the five projects, *TopicMiner*^{MTM} on average improves top-1 and top-5 prediction accuracies of *TopicMiner*^{LDA} by 62.43 and 21.46 percent, respectively.

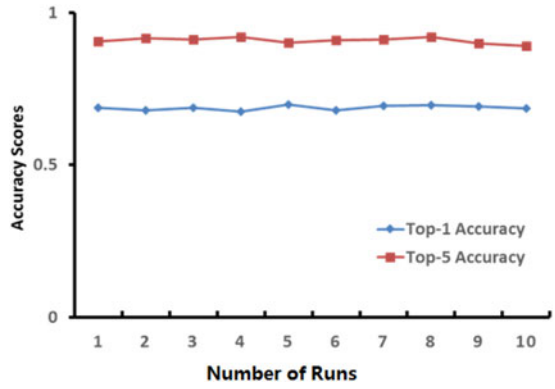


Fig. 25. Average top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of runs applied to Eclipse dataset.

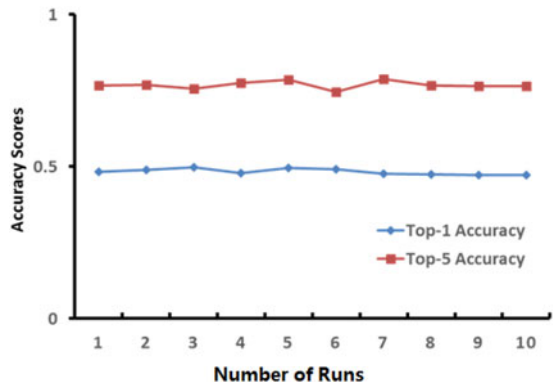


Fig. 26. Average top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of runs applied to Mozilla dataset.

8.5 TopicMiner^{MTM} versus TopicMiner$^{LDA}_{Local}$

Here, we create a new baseline named *TopicMiner*$^{LDA}_{Local}$. This baseline first groups bug reports according to their product-component combination. Next, for each group, we use LDA to extract the topic distributions for the bug reports, and build a TopicMiner model. For a new bug report, we first get its product-component combination, and use the corresponding TopicMiner model for the particular feature combination to recommend fixers.

Table 13 presents the top-1 and top-5 accuracies for TopicMiner^{MTM}, and *TopicMiner*$^{LDA}_{Local}$. We notice that TopicMiner^{MTM} achieves better performance than *TopicMiner*$^{LDA}_{Local}$. Across the five projects, TopicMiner^{MTM} on average improves top-1 and top-5 prediction accuracies of *TopicMiner*$^{LDA}_{Local}$ by 53.87 and 19.67 percent, respectively. Notice TopicMiner^{MTM} uses both global (i.e., from all feature combinations) and local information (i.e., from a specific feature combination) to identify the topics of a bug report, while *TopicMiner*$^{LDA}_{Local}$ only uses local information.

8.6 Adding Product and Component Information to the Baselines

Here, we also incorporate product and component information into the 4 baseline approaches (Bugzie, LDA-KL, SVM-LDA, and LDA-Activity).[7] We first divide the datasets into many small datasets, one for each feature

7. Notice Yang et al.'s approach incorporate the product and component information into their recommendation model.

TABLE 9
TopicMiner^{MTM} (T^M) versus Bugzie (BZ), LDA-KL (KL), SVM-LDA (SVM), LDA-Activity (AC), and Yang et al.'s Approach (Yang), Respectively

Projects	Top-1 Accuracy					
	T^M	BZ	KL	SVM	AC	Yang
GCC	0.5117	0.2654	0.1546	0.2138	0.1765	0.3234
OpenOffice	0.5339	0.2945	0.2011	0.2765	0.1468	0.3814
Netbeans	0.7196	0.3119	0.2234	0.2432	0.1728	0.4321
Eclipse	0.6423	0.2552	0.1541	0.1325	0.0865	0.4214
Mozilla	0.5247	0.2134	0.1543	0.1537	0.1014	0.3009
Average.	**0.5864**	**0.2681**	**0.1775**	**0.2039**	**0.1368**	**0.3718**
Projects	**Top-5 Accuracy**					
	T^M	BZ	KL	SVM	AC	Yang
GCC	0.8041	0.5934	0.4270	0.4867	0.4123	0.7076
OpenOffice	0.7811	0.6006	0.5015	0.5062	0.3786	0.6542
Netbeans	0.8912	0.6012	0.4676	0.4812	0.3815	0.8614
Eclipse	0.9052	0.5372	0.3543	0.2865	0.1890	0.8234
Mozilla	0.7751	0.4756	0.3543	0.3156	0.2236	0.6785
Average.	**0.8313**	**0.5616**	**0.4209**	**0.4152**	**0.3170**	**0.7450**

TABLE 10
The First Five Product-Component Combinations Which Appear at Least 10 Times, and the Top Five Product-Component Combinations Which Appear the Most in OpenOffice Dataset

Product	Component	# Comb.	Top-1	Top-5
performance	code	10	0.00	0.00
udk	documentation	12	0.08	0.17
Infrastructure	documentation	12	0.25	0.58
App Dev	vba	12	0.50	0.50
Base	MySQL Conn	12	0.33	0.33
General	ui	1334	0.43	0.68
gsl	code	1634	0.43	0.73
Writer	code	1668	0.55	0.83
Base	code	2464	0.68	0.92
General	code	2509	0.40	0.69

TABLE 11
The First 5 Product-Component Combinations Which Appear at Least 10 Times, and the Top Five Product-Component Combinations Which Appear the Most in NetBeans Dataset

Product	Component	# Comb.	Top-1	Top-5
serverplugins	Code	10	0.20	0.50
cnd	ClassView	10	0.60	0.60
webservices	Editor	10	0.40	0.70
ide	Commit Validation	10	0.30	0.40
javascript	JSON	10	0.50	0.60
Java	Source	689	0.67	0.94
cnd	Code Model	783	0.61	0.97
debugger	Java	806	0.88	0.99
php	Editor	1087	0.66	0.97
projects	Maven	1132	0.73	0.98

combination (i.e., product-component combination). Next, we train a model for each feature combination, and use the model to recommend fixers for bug reports of the same feature combination. The goal is to check whether the baseline approaches work better than *TopicMiner*^{MTM} if they also consider product and component information. Table 14 presents the top-1 and top-5 accuracies. From the table, we notice the

TABLE 12
TopicMiner^{MTM} (T^M) versus *TopicMiner*^{LDA} (T^L)

Projects	Top-1 Accuracy		Top-5 Accuracy	
	T^M	T^L	T^M	T^L
GCC	0.5048	0.2888	0.7864	0.6443
OpenOffice	0.5161	0.1573	0.7757	0.3687
Netbeans	0.6868	0.5383	0.9084	0.8881
Eclipse	0.6127	0.4278	0.8865	0.8265
Mozilla	0.4831	0.3136	0.7686	0.6690
Average.	**0.5607**	**0.3452**	**0.8251**	**0.6793**

TABLE 13
TopicMiner^{MTM} (T^M) versus *TopicMiner*$^{LDA}_{Local}$ (T^L_{Lo})

Projects	Top-1 Accuracy		Top-5 Accuracy	
	T^M	T^L_{Lo}	T^M	T^L_{Lo}
GCC	0.5048	0.3043	0.7864	0.6649
OpenOffice	0.5161	0.1854	0.7757	0.4032
Netbeans	0.6868	0.5621	0.9084	0.8654
Eclipse	0.6127	0.4456	0.8865	0.8321
Mozilla	0.4831	0.3245	0.7686	0.6821
Average.	**0.5607**	**0.3644**	**0.8251**	**0.6895**

TABLE 14
TopicMiner^{MTM}(T^M) versus Bugzie (BZ*), LDA-KL (KL*), SVM-LDA (SVM*), and LDA-Activity (AC*), Respectively

Projects	Top-1 Accuracy				
	T^M	BZ*	KL*	SVM*	AC*
GCC	0.5048	0.2808	0.1780	0.2470	0.0531
OpenOffice	0.5161	0.3314	0.1461	0.1602	0.0546
Netbeans	0.6868	0.4333	0.4356	0.4594	0.2362
Eclipse	0.6127	0.4186	0.3891	0.3915	0.2605
Mozilla	0.4831	0.2842	0.2552	0.2920	0.1137
Average.	**0.5607**	**0.3497**	**0.2808**	**0.3100**	**0.1436**
Projects	**Top-5 Accuracy**				
	T^M	BZ*	KL*	SVM*	AC*
GCC	0.7864	0.6638	0.5380	0.6021	0.2542
OpenOffice	0.7757	0.6739	0.3529	0.3519	0.2156
Netbeans	0.9084	0.8392	0.8422	0.8553	0.7638
Eclipse	0.8865	0.8230	0.7852	0.8131	0.7342
Mozilla	0.7686	0.6684	0.5992	0.6482	0.4309
Average.	**0.8251**	**0.7337**	**0.6235**	**0.6541**	**0.4797**

improvement of our method over Bugzie, LDA-KL, SVM-LDA, and LDA-Activity are substantial. Across the 5 projects, *TopicMiner*^{MTM} on average improves top-1 and top-5 prediction accuracies of Bugzie by 60.34 and 12.46 percent, LDA-KL by 99.68 and 32.33 percent, SVM-LDA by 80.87 and 26.14 percent, LDA-Activity by 290.46 and 72.00 percent, respectively.

8.7 Impact on Different Number of Iterations

By default, we set the number of iterations for MTM as 500. In this section, we also investigate the performance of TopicMiner^{MTM} with different number of iterations. We set the number of iterations as 100-1,000, and every time increase it by 100. Figs. 27, 28, 29, 30, and 31 present the top-1 and top-5 accuracies for TopicMiner^{MTM} with different

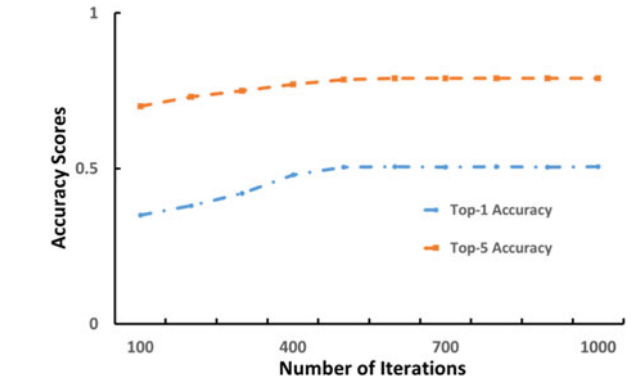


Fig. 27. Top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of iterations applied to the GCC dataset.

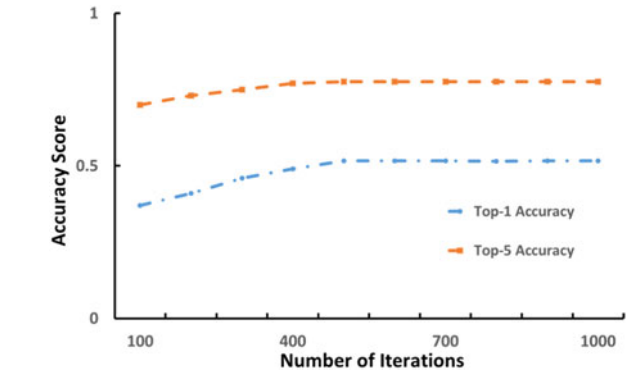


Fig. 28. Top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of iterations applied to the OpenOffice dataset.

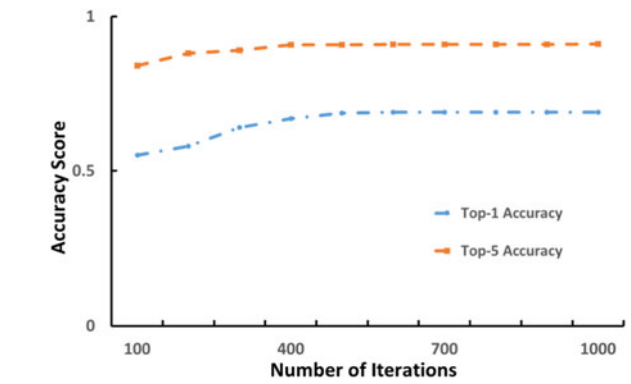


Fig. 29. Top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of iterations applied to the Netbeans dataset.

number of iterations for GCC, OpenOffice, Netbeans, Eclipse, and Mozilla datasets, respectively. We notice that when we increase the number of iterations from 100 to 500, the performance of TopicMiner^{MTM} is increased. However, when we increase the number of iterations from 500 to 1,000, the performance of TopicMiner^{MTM} remains more or less the same. In practice, we will need more time to run MTM if we set a high number of iterations. Thus, we recommend MTM users to set the number of iterations as 500.

8.8 Impact on the Preprocessing of Terms and Fixers

In the preprocessing of our datasets, we exclude bug fixers who appear less than 10 times to reduce noise (following [10], [38]), and we also remove terms which appear less

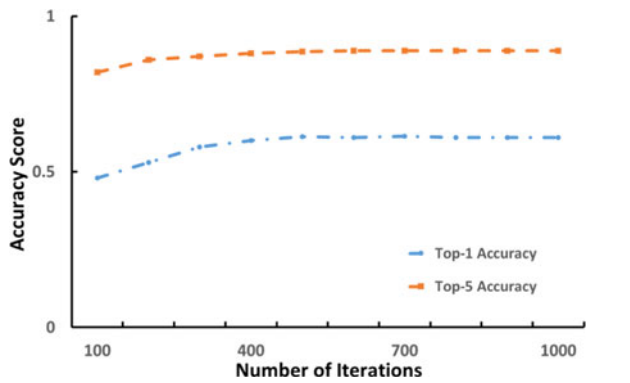


Fig. 30. Top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of iterations applied to the Eclipse dataset.

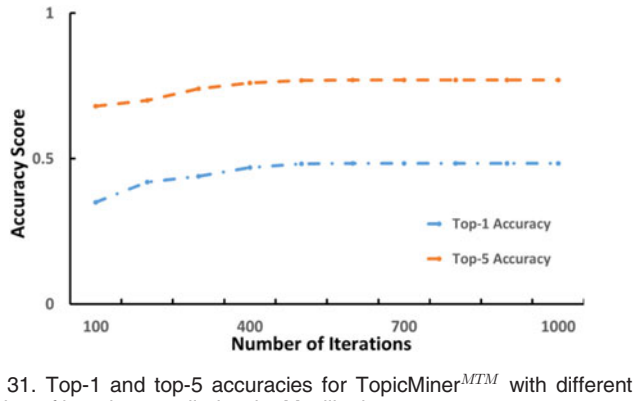


Fig. 31. Top-1 and top-5 accuracies for TopicMiner^{MTM} with different number of iterations applied to the Mozilla dataset.

TABLE 15
Top-1 and Top-5 Accuracies of TopicMiner^{MTM} Compared with the Baseline Approaches in the Noisy Setting

Projects	Top-1 Accuracy					
	T^M	BZ	KL	SVM	AC	Yang
GCC	0.5068	0.2250	0.1290	0.1739	0.1345	0.2654
OpenOffice	0.5077	0.2509	0.1727	0.2315	0.1189	0.3348
NetBeans	0.6432	0.2913	0.1789	0.2123	0.1456	0.4212
Eclipse	0.6220	0.2234	0.1123	0.1036	0.0712	0.3945
Mozilla	0.4724	0.1894	0.1432	0.1345	0.0783	0.2714
Average.	**0.5504**	**0.2360**	**0.1472**	**0.1711**	**0.1097**	**0.3375**
Projects	**Top-5 Accuracy**					
	T^M	BZ	KL	SVM	AC	Yang
GCC	0.7923	0.5513	0.3812	0.4432	0.3723	0.6645
OpenOffice	0.7623	0.5514	0.4652	0.4532	0.3334	0.6512
NetBeans	0.8543	0.5543	0.4234	0.4313	0.3323	0.8213
Eclipse	0.8923	0.4832	0.3234	0.2513	0.1704	0.7945
Mozilla	0.7603	0.4344	0.3234	0.2789	0.1923	0.6345
Average.	**0.8123**	**0.5149**	**0.3833**	**0.3716**	**0.2801**	**0.7132**

than 10 times to reduce noise and speed up the bug triaging process (following [10], [38]). In this section, we also investigate the performance of TopicMiner^{MTM} with all of the terms and fixers in the five datasets. Table 15 presents the top-1 and top-5 accuracies of TopicMiner^{MTM} compared with the baseline approaches in this noisy setting. On average, TopicMiner^{MTM} achieves top-1 and top-5 accuracies of 0.5321 and 0.7736 respectively. We notice that TopicMiner^{MTM} achieves a better performance in the default setting than the noisy setting – which is expected. The same

reduction in performance applies to all baselines. Furthermore, TopicMiner^{MTM} still outperforms the baseline approaches by substantial margins even in the noisy setting.

8.9 Threats to Validity

Threats to internal validity relates to errors and bias in our experiments. We have double checked our experiments and datasets. Still there could be errors that we did not notice. For the ground truth selection, to ensure the fixers in the bug reports are the true bug fixers, we follow the previous approaches such as [38]. We first collected the bug reports which have status of "closed" and "fixed" to ensure these bugs are real bugs and have been fixed. Next, we extracted the fixers in the "assigned to" field. The final fixer who fixed the bug would be recorded into the "assigned to" field. To further reduce the threats due to ground truth identification, we also remove the bug reports with the "assigned to" fields set to generic names.

Our settings for the parameters of LDA and MTM might not be optimal; to minimize this threat we have investigated various numbers of topics and used the same settings for LDA and MTM. Also, even with the suboptimal settings, our approach has outperformed Bugzie, which is one of the state-of-the-art approaches.

In the paper, we use the value of the product and component fields as inputs to MTM. Values of product and component fields have been used by a number of previous studies on bug report management [29], [37], [39], [43]. In the bug triaging process, the appropriate fixer is typically determined after the values of the product and component fields are determined. For example, in Mozilla[8] and Eclipse[9], when users report a bug, they are required to fill the product and component fields, and then the bug triager would find the appropriate fixer. The values of the product and component fields can be changed during the life time of a bug report. In this work, we use the final values of the product and component fields. Thus, we are also interested to check whether the final values of these fields are typically determined before the final bug fixer is determined. We analyze the bug reports of GCC, OpenOffice, Netbeans, Mozilla, and Eclipse, and we find that for 86.52, 85.22, 38.03, 79.83, and 85.33 percent of the bug reports of the respective software projects the product and component fields are finalized first before the final bug fixers are assigned. Notice that for Netbeans, the percentage is low; we asked some developers in the Netbeans development community and they told us that this is due to the community maintenance process – i.e., many products and components are renamed and the values of the product and component fields of older bug reports are changed to reflect the new names. Despite the difference between Netbeans and the other datasets, we include it for diversity purpose. Reclassifications of old products and components happen in practice, and we need to investigate if our approach works well for such situation. For the other four software projects, most of the bug reports have their product and component fields finalized before the fixer is determined. Thus, we believe the usage of the product and component fields is realistic and the values of these fields can be used to help find a suitable fixer in practice.

Threats to external validity relates to the generalizability of our results. We have analyzed 227,278 bug reports from five software systems. In the future, we plan to reduce this threat further by analyzing more bug reports from more software systems.

Threats to construct validity refers to the suitability of our evaluation measures. We use top-1 and top-5 accuracies which are also used by past bug triaging studies [10], [20], [38]. Thus, we believe there is little threat to construct validity.

9 Related Work

9.1 Automated Bug Triaging

There are a number of machine learning and information retrieval approaches for automatic bug triaging [7], [8], [15], [20], [22], [26], [27], [35], [38]. Anvik et al. and Cubranic et al. propose the bug triaging problem, and use machine learning methods such as Naive Bayes, SVM, and C4.8 to solve it [7], [15]. Jeong et al. propose to use a bug tossing graph to improve bug triaging prediction accuracy [20]. Bhattacharya et al. improve the accuracy of the approach by Jeong et al. further by proposing a multi-feature tossing graph [10]. Tamrawi et al. [38] propose a method called Bugzie, which uses a fuzzy set and cache-based approach to increase the accuracy of bug triaging. Naguib et al. propose a method that compares a bug report to developers in topic space by leveraging LDA [27]. They first categorize bug reports into topics by using LDA, and then create activity profiles for developers in a bug tracking system. A profile contains two parts: developer's role and topic associations. These profiles are then used to recommend developers for new bug reports. *TopicMiner*^{MTM} and LDA-Activity use different topic models and the formulas used to compute the suitability of a developer to a bug report also differ. To assign topics to words in bug reports, we build a specialized topic model, i.e., MTM, that takes special characteristics/features of bug reports into consideration. Also, to compute developer topic associations (i.e., $\mu_b^f(d)$, the similarity between a developer and a topic), different from LDA-Activity, *TopicMiner*^{MTM} considers feature combination and the rarity of a topic (i.e., the denumerator of Equation (8)). To predict bug fixers, LDA-Activity considers developer historical contributions not only as fixers but also in other roles (i.e., reviewers and assigners). *TopicMiner*^{MTM} focuses on developer contributions as fixers; it ignores developer contributions in other roles which might introduce noise. We have shown that *TopicMiner*^{MTM} outperforms LDA-Activity by a substantial margin.

Yang et al. also use LDA to extract topics from bug reports, and find bug reports related to each topic [46]. For a new bug report, their approach first decides the topics of the bug report. Then they utilize multiple features (i.e., component, product, priority and severity) to identify similar reports that have the same set of features as the new bug report, and recommend developers based on the similar reports. Our approach is different from Yang et al.'s approach. First, we design a specific topic model named

8. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Bug_writing_guidelines

9. https://bugs.eclipse.org/bugs/page.cgi?id=bug-writing.html

MTM which incorporates the multi-feature information into the topic model while Yang et al. only use LDA. Second, Yang et al. use severity and priority fields in the bug reports; however, in practice, most of the bug reports set their severity and priority values as the default value [44].

There have been a number of automatic bug triaging methods that use other information sources aside from bug reports, e.g., commits and source code comments. A number of approaches use feature location techniques to find program units (e.g., files or classes) that are related to a change request (i.e., bug report or feature request) and then mine commits in version control repositories or comments in source code files to recommend appropriate developers [21], [24], [33]. The success of these approaches depends on the accuracy of feature location techniques which are often still low [32]. Also, the quality of the code comments and commits information can be poor due to outdated comments [19], unavailability of authorship information for authors without commit rights in CVS and SVN repositories [23], etc. In this work, we focus on analyzing textual information available in bug reports to recommend appropriate fixers.

9.2 Other Studies on Bug Report Management

Somasundaram and Murphy propose LDA-KL and SVM-LDA to recommend appropriate components to a bug report [35]. They first extract topic distributions of bug reports by using LDA, and then use KL-divergence and support vector machine to recommend appropriate components. LDA-KL computes the similarity between the topic distribution of a new bug report with the average topic distribution of a collection of bug reports belonging to the same unit (i.e., component). The units with the least divergence are recommended. SVM-LDA works by inputting topic distributions of training (historical) bug reports and their labels (i.e., components they belong to) to a SVM to create a classifier. This classifier is then used to predict labels of a new bug report. In this work, we adapt LDA-KL and SVM-LDA for bug fixer recommendation; we do so by considering bug reports fixed by the same developer as a unit (for LDA-KL) and by considering bug fixers as labels (for SVM-LDA). We have compared $TopicMiner^{MTM}$ with LDA-KL and SVM-LDA, and shown that $TopicMiner^{MTM}$ outperforms these baselines.

Bortis and van der Hoek propose Porchlight which allows developers to tag bug reports and search bug reports of interest using a query language [13]. Our work, similar to past automated bug triaging solutions [7], [8], [15], [20], [26], [27], [35], [38] is complementary to Porchlight. Porchlight can be used along with automated bug triaging solutions to navigate through a large number of bug reports and assign appropriate developers to each one of them.

Related to bug triaging studies, a few studies recommend people that would participate in a bug resolution process [41], [43], [45]. These people include triagers, fixers, and other people that post one or more comments in a discussion thread, corresponding to a bug report, in a bug tracking system. For example, Xia et al. use LDA and multi-label learning to recommend participants in a bug resolution process [43]. Different from these studies, in this work, we focus on the bug fixers. For a bug report, since there is only one fixer but there are many participants to the bug resolution process, we address a more difficult problem. Different from the approach in [43], we propose a new topic model, named MTM, and show that it can outperform LDA for bug triaging.

9.3 Specialized Topic Model

Nguyen et al. propose a specialized topic model to find buggy source code files [28]. Nguyen et al. propose another topic model to detect duplicate bug reports [29]. Our work is orthogonal to the above studies and our topic model is also different from the models used in the above studies. *MTM* assigns topics to words in a bug report from the topic distribution of the corresponding feature combination of that report. We need to consider feature combination for bug triaging as many developers are more familiar with a particular set of product and component combinations than other combinations. However, for bug triaging, there is no need to consider the topic distribution of buggy source code files (considered in [28]) or the topic distribution of buggy concepts shared by bug reports that are duplicate of one another (considered in [29]).

10 Conclusion and Future Work

We propose a new topic model based bug triaging approach, named *TopicMiner*, and a new topic model, named multi-feature topic model, which takes into consideration the features of a bug report when assigning topics to words in the report. We have evaluated our solution on 227,278 bug reports from five software systems and demonstrate that $TopicMiner^{MTM}$ outperforms Bugzie, LDA-KL, SVM-LDA, LDA-Activity, and Yang et al.'s approach by substantial margins.

In the future, we plan to improve the effectiveness of our approach further, and investigate additional bug reports. Also, in this work, we merge the two features (i.e., product and component) as one composite feature (i.e., by creating a feature combination). Other ways of using the multiple features exist and we plan to explore them in a future work. We also plan to design a better topic model to predict fixers when the number of bug reports in a specific product-component combination is small (e.g., by using a mixture of models which includes a general model that the approach can back off to when the number of bug reports in a specific product-component combination is small).

Acknowledgments

This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, NSFC Program (No. 61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01. Xinyu Wang is the corresponding author.

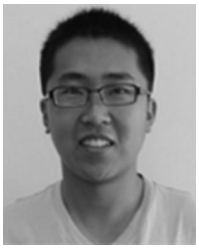
References

[1] Eclipse bug tracking system. (2016). [Online]. Available: https://bugs.eclipse.org/bugs/
[2] Gcc bug tracking system. (2016). [Online]. Available: http://gcc.gnu.org/bugzilla/

[3] Mozilla bug tracking system. (2016). [Online]: https://bugzilla.mozilla.org/.
[4] Netbeans bug tracking system. (2016). [Online]. Available: http://netbeans.org/bugzilla/
[5] Openoffice bug tracking system. (2016). [Online]. Available: https://issues.apache.org/ooo/
[6] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons" in *Encyclopedia of Measurement and Statistics*, N. J. Salkind, Ed. Thousand Oaks, CA, USA: Sage, 2007.
[7] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 361–370.
[8] J. Anvik and G. Murphy, "Determining implementation expertise from bug reports," presented at the *Proc. 4th Int. Workshop Min. Softw. Repositories*, Washington, DC, USA, 2007.
[9] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams," in *Proc. 2010 ACM Conf. Comput. Support. Coop. Work*, 2010, pp. 291–300.
[10] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *Proc. 2010 IEEE Int. Conf. Softw. Maint.*, 2010, pp. 1–10.
[11] D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt, "Understanding lda in source code analysis," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 26–36.
[12] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet, et al., location," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.
[13] G. Bortis and A. van der Hoek, "Porchlight: A tag-based approach to bug triaging," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 342–351.
[14] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. Hove, United Kingdom: Psychology Press, 2014.
[15] D. Čubranić, "Automatic bug triage using text categorization," in *Proc. 16th Int. Conf. Softw. Eng. Knowl. Eng.*, 2004, pp. 92–97.
[16] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Am. Soc. Inf. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.
[17] G. Heinrich, Parameter estimation for text analysis. [Online]. Available: http://www.arbylon.net/publications/text-est.pdf, 2005.
[18] T. Hofmann, "Probabilistic latent semantic analysis," in *Proc. 99th Conf. Uncertain. Artif. Intell.*, 1999, pp. 289–296.
[19] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2293–2304, 2012.
[20] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2009, pp. 111–120.
[21] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *J. Softw.: Evol. Process*, vol. 24, no. 1, pp. 3–33, 2012.
[22] H. H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *J. Softw. Maint.*, vol. 24, no. 1, pp. 3–33, 2012.
[23] C. Kolassa, D. Riehle, and M. A. Salim, "A model of the commit size distribution of open source," in *SOFSEM 2013: Theory and Practice of Computer Science*. Berlin, Germany: Springer, 2013.
[24] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *Proc. 28th IEEE Int. Conf. Softw. Maint.*, 2012, pp. 451–460.
[25] C. D. Manning, P. Raghavan, and H.Schütze, *Introduction to Information Retrieval*, vol. 1. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[26] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proc. 6th IEEE Int. Work. Conf. Min. Softw. Repositories*, 2009, pp. 131–140.
[27] H. Naguib, N. Narayan, B. Brugge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *Proc. 10th IEEE Work. Conf. Min. Softw. Repositories*, 2013, pp. 22–30.
[28] A. Nguyen, T. Nguyen, J. Al-Kofahi, H. Nguyen, and T. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 263–272.
[29] 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 *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 70–79.
[30] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 522–531.
[31] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.
[32] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. IEEE/ACM 28th Int. Conf. Autom. Softw. Eng.*, 2013, pp. 345–355.
[33] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *Proc. 10th Work. Conf. Min. Softw. Repositories*, 2013, pp. 2–11.
[34] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "A time-based approach to automatic bug report assignment," *J. Syst. Softw.*, vol. 102, pp. 109–122, 2015.
[35] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent Dirichlet allocation," in *Proc. 5th India Softw. Eng. Conf.*, 2012, pp. 125–130.
[36] M. Steyvers and T. Griffiths, "Probabilistic topic models," in *Latent Semantic Analysis: A Road to Meaning*, T. Landauer, D. McNamara, S. Dennis, and W. Kintsch, Eds., Hove, United Kingdom: Laurence Erlbaum, 2007.
[37] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 253–262.
[38] A. Tamrawi, T. Nguyen, J. Al-Kofahi, and T. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 365–375.
[39] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empir. Softw. Eng.*, pp. 1–30, 2014.
[40] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol 1, no. 6, pp. 80–83, 1945.
[41] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *Proc. 18th Asia Pac. Softw. Eng. Conf.*, 2011, pp. 389–396.
[42] X. Xia, Y. Ding, D. Lo, J. Al-Kofahi, T. Nguyen, and X. Wang, "Toward more accurate bug triaging with topic modeling," Tech. Report. [Online]. Available: http://pan.baidu.com/s/1eRkv4Dc, 2015.
[43] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proc. 20th Work. Conf. Reverse Eng.*, 2013, pp. 72–81.
[44] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maint. Reengineering Reverse Eng.*, 2014, pp. 174–183.
[45] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: Developer recommendation based on topic models for bug resolution," in *Proc. 8th Int. Conf. Predictive Models Softw. Eng.*, 2012, pp. 19–28.
[46] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," in *Proc. IEEE 38th Annu. Comput. Softw. Appl. Conf.*, 2014, pp. 97–106.

Xin Xia received the PhD degree from the College of Computer Science and Technology, Zhejiang University, China, in 2014. He is currently a research assistant professor at the College of Computer Science and Technology, Zhejiang University. His research interests include software analytic, empirical study, and mining software repository.

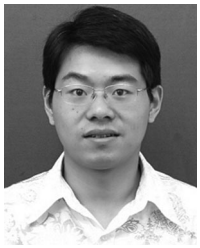
David Lo received the PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 130 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009. He has won a number of research awards including an ACM distinguished paper award for his work on bug report management. He has published in many top international conferences in software engineering, programming languages, data mining and databases, including the ICSE, FSE, ASE, PLDI, KDD, WSDM, the *IEEE Transactions on Knowledge and Data Engineering*, ICDE, and VLDB. He has also served on the program committees of ICSE, ASE, KDD, VLDB, and many others. He is a steering committee member of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) which is a merger of the two major conferences in software engineering, namely CSMR and WCRE. He will also serve as the general chair of ASE 2016. He is a leading researcher in the emerging field of software analytics and has been invited to give keynote speeches and lectures on the topic in many venues, such as the 2010 Workshop on Mining Unstructured Data, the 2013 Génie Logiciel Empirique Workshop, the 2014 International Summer School on Leading Edge Software Engineering, and the 2014 Estonian Summer School in Computer and Systems Science.



Ying Ding received the BS degree from Nanjing University of Posts and Telecommunications in 2011. He is currently working toward the PhD degree with the School of Information Systems, Singapore Management University. He works in the area of text mining and recommender systems. His primary research interest is using machine learning techniques, such as probabilistic graphical models and deep learning, to help computer understand natural language better.

Jafar M. Al-Kofahi received the BS degree in computer science from Jordan University of Science and Technology, Irbid, Jordan, in 2005. He is currently working toward the PhD degree in software engineering at Iowa State University. He is also a system developer engineer at Amazon Business. His research interests are in the area of software maintenance and evolution, and currently focusing on build code maintenance and evolution.

Tien N. Nguyen is currently an associate professor in both Electrical and Computer Engineering Department and Computer Science Department at Iowa State University. His research interests include program analysis, mining large-scale software repositories, and software maintenance and evolution. Since 2009, he has been awarded three ACM SIGSOFT Distinguished Paper Awards, one Best Paper Award, and one best ICSE Formal Research Demonstration Award at the top-tier, international software engineering conferences including ICSE, FSE, and ASE. His research has been supported by 14 external grants including eight US National Science Foundation grants from US National Science Foundation, and several grants from industry. He has served on Program Committees and Program Boards of top-tier software engineering conferences including ICSE, FSE, ASE, OOPSLA, and ECOOP.



Xinyu Wang received the graduate and PhD degrees in computer science from Zhejiang University of China, in 2002 and 2007, respectively. He was a research assistant at the Zhejiang University, from 2002 to 2007. He is currently an associate professor in the College of Computer Science, Zhejiang University. His research interests include software engineering, formal methods, and very large information systems.