# Exploring Metadata in Bug Reports for Bug Localization

Xiaofei Zhang, Yuan Yao, Yaojing Wang, Feng Xu, Jian Lu
National Key Laboratory for Novel Software Technology, Nanjing University
Collaborative Innovation Center for Novel Software Technology and Industrialization
Nanjing 210023, China
xiaofeizhang@smail.nju.edu.cn, y.yao@nju.edu.cn, wyj@smail.nju.edu.cn, {xf, lj}@nju.edu.cn

***Abstract*—Information retrieval methods have been proposed to help developers locate related buggy source files for a given bug report. The basic assumption of these methods is that the bug description in a bug report should be textually similar to its buggy source files. However, the metadata (such as the component and version information) in bug reports is largely ignored by these methods. In this paper, we propose to explore the metadata for the bug localization task. In particular, we first apply a generative model to locate buggy source files based on the bug descriptions, and then propose to add the available metadata in bug reports into the localization process. Experimental evaluations on several software projects indicate that the metadata is useful to improve the localization accuracy and that the proposed bug localization method outperforms several existing methods.**

***Index Terms*—Bug localization, bug reports, bug metadata, supervised topic modeling**

## I. Introduction

In many software projects, the bug tracking system has been introduced to collect and manage bug reports. When a tester or developer discovers an abnormal behavior of the software, he or she will be asked to fill in a form provided by the bug tracking system. The form consists of a bug description and some bug metadata such as product, component, version, and hardware (see Fig. 1 as an example). Such a bug tracking system is helpful for developers to maintain software quality [1].

Even with a bug tracking system, it is still not an easy task to locate the buggy source files. For example, a developer who is assigned to deal with a bug report probably needs to analyze the information of the bug report, reproduce the abnormal behavior [2], and perform code review [3] to find the cause. To save the developers' effort, a better way is to automatically recommend a list of potential buggy source files for a given bug report. This task is referred to as bug localization.

For the bug localization task, information retrieval (IR) methods have been widely used. The IR methods treat each bug report as a query and the existing source files in the code repository as a collection of documents. Then, the bug localization task becomes a standard IR problem where the goal is to find textually similar source files for a given bug report. Under the IR framework, the key difference of the existing methods is on the similarity computation aspect. For example, Lukins et al. [4] applied Latent Dirichlet Allocation (LDA) [5] to learn the latent topics of bug reports and source files, based on which the similarity can be computed. Zhou et al. [6] adopted the vector space model (VSM) to compute similarities and further incorporated the similarities between bug reports to improve performance. Recently, Lam et al. [7] and Huo et al. [8] used deep learning techniques to extract features for bug reports and sources files, and compute similarities on these features.

One limitation of the existing IR methods is that they tend to ignore the metadata in the bug reports. On the one hand, most of the existing IR methods use only the bug descriptions in the bug reports (e.g., [6], [8]–[10]); even for the few methods that use metadata (e.g., [11]), they simply treat the metadata as part of the bug description. On the other hand, metadata in bug reports is potentially useful for bug localization as it contains valuable information (such as the component and version information) in addition to the bug description; for example, the component metadata may indicate that a subset of source files are related to the bug report; it has also been shown that metadata is helpful for other tasks such as bug report identification [12].

In this paper, we propose to systematically explore the metadata for the bug localization task. In particular, instead of following the existing IR methods, we first use a supervised topic modeling method to locate buggy source files based on the bug descriptions. The intuition is that compared to unsupervised IR methods, supervised methods may have better performance in terms of accurately identifying the relevant source files. The basic idea to formulate the problem as a supervised learning problem is by using existing bug fixing histories as supervision information and use the source filenames as the supervision labels. Further, we discover that the naming of source files often follow a hierarchy structure (e.g., the full name of a source file is the concatenation of the module name and the class name) and the substrings of filenames often appear multiple times in the content of bug report descriptions. For example, 'aspectj/weaver/bcel/LazyClassGen.java' is a source filename and its substrings like 'weaver' and 'lazyclass' may frequently appear in the descriptions of the related bug reports. We propose to incorporate this phenomenon into topic modeling, and we name this model as L2SS (Label-to-SubString).

Next, we propose to incorporate several types of metadata into L2SS. In particular, we consider seven types of metadata and add each of them into L2SS to see its usefulness. The

metadata consists of component, version, platform, operating system, priority, severity, and reporter. The basic idea to incorporate metadata is to model it as the prior probability for choosing topics. For example, if the component metadata is 'UI runtime', it is probable that the buggy source files are related to the UI. Such information can be used to lead the search direction of possible buggy source files for a given bug report. The model with metadata incorporated is named as L2SS+X where 'X' stands for the corresponding matadata.

To verify the effectiveness of the proposed method, we conduct experiments on four open source projects with a total of 36,417 bugs. The results show that the proposed method outperforms several existing methods in terms of localization accuracy, and that the metadata is useful to further improve the localization accuracy. For example, when evaluated on the JDT (Java development tools) dataset (which contains 5,211 fixed bug reports), one of the proposed method (i.e., L2SS+CM) is able to find the related buggy files within the top 10 recommendations for over 70% bug reports, while the BugLocator competitor [6] can help 55% bug reports within the top 10 recommendations. Meanwhile, the proposed method can be efficiently pre-trained and used at the prediction stage.

The main contributions of this paper include:

1) We propose a supervised topic modeling method L2SS for the bug localization task. L2SS uses existing bug fixings and treats the source filenames as the supervision information. By doing so, L2SS can be applied when developers only have access to the source filenames.
2) We study the usefulness of several types of metadata in the bug report, and revise L2SS to incorporate these metadata (denoted as L2SS+). For example, when the component metadata is incorporated, we have the L2SS+CM method.
3) We evaluate the performance of L2SS and L2SS+ on several real datasets, and the results show the effectiveness and efficiency of the proposed methods. For example, L2SS+CM can achieve up to 16.9% improvement compared to its best existing competitors.

The remainder of this paper is organized as follows. Section II provides some background information. Section III presents the proposed bug localization method. Section IV describes the experimental setup and the research questions. Section V presents the experimental results. Section VI discusses the threats to validity. Section VII covers related work, and Section VIII concludes.

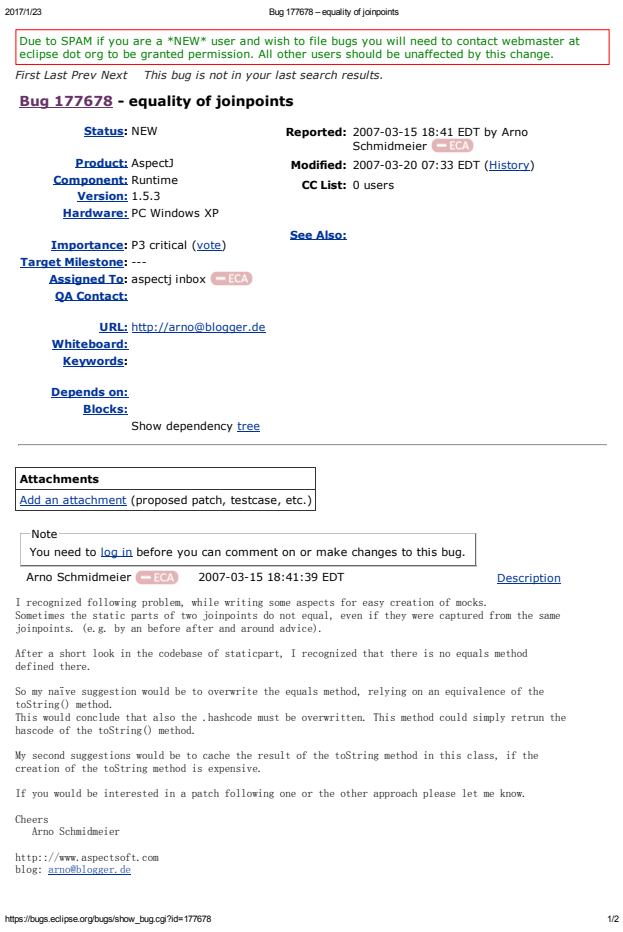
## II. Background

In this section, we introduce some background information.

### A. Bug Reports and Metadata

Bug tracking system has been widely used in software projects. Take the Bugzilla System in Eclipse project as an example. Fig. 1 shows a real bug report[1] (ID:177678) for AspectJ in Eclipse.

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=177678


2017/1/23 Bug 177678 – equality of joinpoints

Due to SPAM if you are a *NEW* user and wish to file bugs you will need to contact webmaster at eclipse dot org to be granted permission. All other users should be unaffected by this change.

First Last Prev Next This bug is not in your last search results.

**Bug 177678 - equality of joinpoints**

Status: NEW
Reported: 2007-03-15 18:41 EDT by Arno Schmidmeier
Product: AspectJ
Component: Runtime
Version: 1.5.3
Hardware: PC Windows XP
Modified: 2007-03-20 07:33 EDT (History)
CC List: 0 users
See Also:
Importance: P3 critical (vote)
Target Milestone: ---
Assigned To: aspectj inbox
QA Contact:
URL: http://arno@blogger.de
Whiteboard:
Keywords:
Depends on:
Blocks:
Show dependency tree

Attachments
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Arno Schmidmeier 2007-03-15 18:41:39 EDT Description

I recognized following problem, while writing some aspects for easy creation of mocks.
Sometimes the static parts of two joinpoints do not equal, even if they were captured from the same joinpoints. (e.g. by an before after and around advice).

After a short look in the codebase of staticpart, I recognized that there is no equals method defined there.

So my naïve suggestion would be to overwrite the equals method, relying on an equivalence of the toString() method.
This would conclude that also the .hashcode must be overwritten. This method could simply retrun the hascode of the toString() method.

My second suggestions would be to cache the result of the toString method in this class, if the creation of the toString method is expensive.

If you would be interested in a patch following one or the other approach please let me know.

Cheers
Arno Schmidmeier

http://www.aspectsoft.com
blog: arno@blogger.de

https://bugs.eclipse.org/bugs/show_bug.cgi?id=177678 1/2

Fig. 1. A bug report example in the AspectJ project.

We can see from Fig. 1 that a bug report typically contains description, status, product, component, version, hardware (platform and operating system), importance (priority and severity), etc. Except for the bug description, we refer to the rest information as metadata. The metadata is widely ignored by existing information retrieval methods, while it may be helpful for bug localization. For example, the component metadata may indicate a subset of source files that are related to the bug; the bug reporter metadata can also play similar roles as a reporter may be responsible for testing a specific part of the code repository.

### B. Supervised Topic Modeling

Topic modeling (e.g. LDA [5]) has been used to capture the topical similarities between bug reports and source codes to locate bugs [4]. To make use of the existing fixing histories between source files and bug reports, a natural tool is supervised topic modeling. In bug localization task, bug reports can be seen as documents and the related source filenames can be seen as the labels for the documents. Then, given a bug report, the goal is to predict its related labels (i.e., source filenames).

Fig. 2 shows an example (LLDA [13]) of supervised topic modeling. The basic idea of LLDA is as follows. First, each

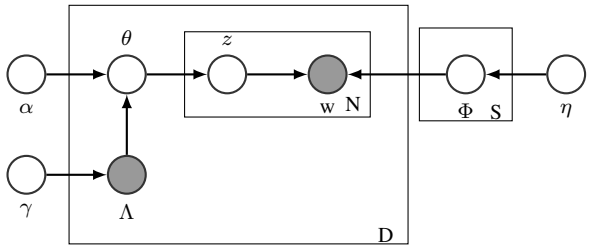



Fig. 2. Graphical representation for LLDA.

bug report contains several topics, and all the possible topics form the vector $\theta$. For a specific bug report, its topics are determined by the indicator vector $\Lambda$ which indicates the related source files for this bug report. In other words, $\Lambda$ contains the supervision labels, and we assume that each source file corresponds to a unique topic. Then, each word $w$ in a bug report is generated from one specific topic $z$ sampled from $\theta$ and the topic-word distribution $\Phi$. $\alpha, \eta, \gamma$ are hyper-parameters of the model.

## III. The Proposed Approach

In this section, we first present problem statement, and then describe the proposed methods (L2SS and L2SS+).

### A. Problem Statement

We first introduce the notations used in this paper. We use $D$ and $S$ to denote the collection of bug reports and source files, respectively. For each bug report, we denote its bug description as $d$ and its metadata as $m_d$. We assume $d$ has $N_d$ words which is drawn from the vocabulary $V$. Each bug report has a indicator vector $\Lambda^{(d)}$ of length $|S|$ where each element in $\Lambda^{(d)}$ is a binary value indicating whether or not the source file is related to this bug report.

With these notations, we define the bug localization problem as follows.

**Problem** *Bug Localization Problem*

**Given:** *(1) a set of bug reports $D$ where each bug report contains a bug description $d$ with $N_d$ words and a collection of metadata $m_d$, (2) the existing bug fixings where each bug report corresponds to a binary indicator vector $\Lambda^{(d)}$, and (3) a new bug report.*

**Find:** *the related buggy source files for the new bug report.*

As we can see, the input of our methods includes the bug reports (containing bug descriptions and metadata) and the existing fixing histories. Then, for a new bug report, we aim to output a ranked list of possible buggy source files.

### B. The L2SS Model

Next, we present the L2SS method when metadata is not used as input. In many bug reports, the source filenames may have appeared in the bug reports. For example, bug report may contain the stack trace which may provide direct clues for the possible buggy files [14]. We refer to this phenomenon as word co-occurrence, and aim to incorporate such phenomenon in our model. The graphical representation of L2SS is shown in Fig. 3.

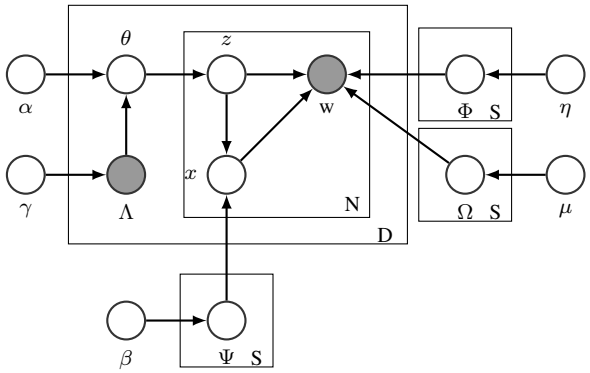



Fig. 3. Graphical representation for L2SS.

Compared with Fig. 2, there are several modifications. First, we introduce an additional topic-word distribution $\Omega$. $\Omega$ contains the substrings of the chosen topic $z$ (each topic corresponds to a source filename). Second, we have two ways for generating a word $w$. The first way is the same with Fig. 2, and the second way is for word co-occurrence. That is, when generating a word $w$, we directly choose it from $\Omega$. Third, to manage the two ways, we introduce a switching variable $x$ which obeys the binomial distribution $\Psi$ to decide which way to generate word $w$.

With the above topic model, we can train the parameters by feeding the existing data (bug reports, bug fixings, and source filenames) as input, and apply the trained model on the new bug report to output the possibly related source files. A complete description of the L2SS model can be summarized in the following generative process.

1) Draw topic-word distributions
   a) For each topic $k \in K$,
      i) Draw distribution $\Psi_k \sim Beta(\beta)$
      ii) Draw distribution $\Phi_k \sim Dir(\eta)$
      iii) Draw distribution $\Omega_k \sim Dir(\mu)$
2) Draw words for each document $d \in D$
   a) For each topic $k \in K$,
      i) Generate $\Lambda_k^{(d)} \sim Bernoulli(\gamma)$
3) Draw distribution $\theta^{(d)} \sim Dir(\alpha^{(d)})$
   a) For each position $i \in$ [1,$N^{(d)}$] in $d$,
      i) Generate $z_i \in \{k|\Lambda_k^{(d)} = 1\} \sim Mult(\theta^{(d)})$
      ii) Generate $w_{i,\Omega} \in \{w|w \in Sub(z)\} \sim Mult(\Omega_{z_i})$
      iii) Generate $w_{i,\Phi} \in \{w|w \in V\} \sim Mult(\Phi_{z_i})$
      iv) Generate $x_{z_i} \sim Bernoulli(\Psi_{z_i})$
      v) Generate $w_i = (w_{i,\Omega})^{x_i} \cdot (w_{i,\Phi})^{1-x_i}$

where we assume there are $K$ topics ($K = |S|$). $Beta(\cdot)$, $Dir(\cdot)$, $Mult(\cdot)$, and $Bernoulli(\cdot)$ are Beta distribution, Dirichlet distribution, Multinomial distribution, and Bernoulli distribution, respectively. $\beta$ is the Beta prior, and $\eta$, $\mu$, and

$\alpha^{(d)}$ are the Dirichlet priors. $Sub(\cdot)$ is an operator to obtain the set of a source filename's substrings.

For the specific training algorithm, we adopt CVB0 [15] algorithm to learn the parameters because it converges faster and more stable than the Gibbs sampling and expectation maximization [16]. We omit the detailed training algorithms for brevity.

### C. The L2SS+ Model

Next, we show how to incorporate metadata into L2SS. Basically, we incorporate metadata in the prediction stage of L2SS. In the prediction stage, we compute the probabilities for each source file of being the buggy source file for a given bug report. By ranking the source files with their probabilities, we can recommend the suspected buggy files in a ranked list.

To make the recommendations, the goal is to compute probability $p(z|d)$ which stands for the relatedness between bug reprot $d$ and source file $z$. Here, $d$ is the given bug report, and $z$ is a topic (remind that each topic corresponds a source file in our method). By Bayes formula, we have

$$p(z|d) = \frac{p(d|z)p(z)}{p(d)}, \tag{1}$$

where $p(d)$ is a constant value for all files. Therefore, we only need to calculate $p(d|z)$ and $p(z)$.

We first check $p(d|z)$ which indicates the probabilities of the words in bug report $d$ given the topic $z$. The bug report $d$ can be decomposed into a bag of words, and with independency assumption, $p(d|z)$ can be computed as

$$p(d|z) = \prod_{w \in d} p(w|z). \tag{2}$$

Next, since there are two ways to generate word $w$ from topic $z$ in L2SS, we have the following equation to compute $p(w|z)$

$$\begin{aligned} p(w|z) = p(w|x=0,z)p(x=0|z) \\ +p(w|x=1,z)p(x=1|z), \end{aligned} \tag{3}$$

where $x$ is the switching variable. When $x$ is equal to 0, $w$ is generated from distribution $\Phi$; when $x$ is equal to 1, $w$ is generated from distribution $\Omega$. Using the learned parameters of L2SS, we can re-write the above equation as

$$p(w|z) = \Phi_{z,w}\Psi_z + \Omega_{z,w}(1-\Psi_z). \tag{4}$$

We next check $p(z)$. Without considering metadata, we have

$$p(z) = \sum_{d \in D} p(d)p(z|d) = \frac{\sum_{d \in D} p(z|d)}{N_D}, \tag{5}$$

where $D$ is the collection of bug reports and $N_D$ is the number of bug reports. In L2SS, we assume that the probability of every bug report is equal to $\frac{1}{N_D}$. Then we can use the learned parameter $\Theta$ to represent $p(z)$ as

$$p(z) = \frac{\sum_{d \in D} \Theta_{d,z}}{N_D}. \tag{6}$$

Putting $p(d|z)$ and $p(z)$ together, we can compute $p(z|d)$ with the learned parameters from L2SS as follows.

$$\begin{aligned} p(z|d) &= \frac{1}{N_D p(d)} \sum_{d \in D} \Theta_{d,z} \prod_{w \in d} [\Phi_{z,w}\Psi_z + \Omega_{z,w}(1-\Psi_z)] \\ &\propto \sum_{d \in D} \Theta_{d,z} \prod_{w \in d} [\Phi_{z,w}\Psi_z + \Omega_{z,w}(1-\Psi_z)] \end{aligned} \tag{7}$$

Next, since metadata is potentially useful for the bug localization problem, we can model metadata into $p(z)$. In other words, instead of computing $p(z|d)$, we aim to compute $p(z|d,m)$. For example, if we use the component metadata, instead of letting all files share the same $p(z)$, we can assume that bug reports with different components have different $p(z)$. In this work, we use $p(z|m)p(m)$ to substitute $p(z)$, where $m$ stands for the metadata of a given bug report, and $p(z|m)$ is the probability of choosing topic $z$ with metadata $m$. For a given bug report, $m$ is observed and $p(m)$ is a constant value. Therefore, instead of computing $p(z)$ as shown in Eq. (6), we estimate it by the following formula when metadata $m$ is incorporated

$$p(z|m) = \frac{\sum_{d \in D_m} \Theta_{d,z}}{N_m}, \tag{8}$$

where $D_m$ is the collection of bug reports whose corresponding metadata value is also $m$, and $N_m$ is the size of $D_m$. For all the seven types of metadata, we can incorporate each of them as the above equation.

Overall, for L2SS+, the probability $p(z|d,m)$ can be computed as follows.

$$p(z|d,m) \propto \frac{\sum_{d \in D_m} \Theta_{d,z}}{N_m} \prod_{w \in d} [\Phi_{z,w}\Psi_z + \Omega_{z,w}(1-\Psi_z)] \tag{9}$$

We call the revised L2SS model as L2SS+ (e.g., L2SS with component metadata as L2SS+CM). For the seven types of metadata, we have seven methods: L2SS+CM with component metadata, L2SS+OS with operating system metadata, L2SS+PF with platform metadata, L2SS+VS with version metadata, L2SS+PR with priority metadata, L2SS+SV with severity metadata, and L2SS+RP with reporter metadata.

## IV. Experimental Setup

In this section, we present the experimental setup as well as the research questions we aim to answer in this paper.

### A. Subject Systems

We use four open source projects from the website of Eclipse as the subjects for evaluation. The statistics of the four projects are shown in Table I.

- The JDT project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins.
- AspectJ is a seamless aspect-oriented extension to the Java programming language.
- The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy

TABLE I
STATISTICS OF FOUR PROJECTS.

| Project | bugs | changed files | source files | bug reports |
|---|---|---|---|---|
| JDT | 15621 | 797 | 1842 | 5211 |
| AspectJ | 2994 | 275 | 1045 | 1415 |
| PDE | 12164 | 1309 | 2577 | 3844 |
| BIRT | 5638 | 832 | 8495 | 2677 |

Eclipse plug-ins, fragments, features, site updates and RCP products.

- Eclipse BIRT is an open source Eclipse-based reporting system that integrates with Java/Java EE application to produce compelling reports.

All bug reports are collected from Bugzilla of Eclipse[2]. Related fixing histories can be collected from the log of git repository by bug id. These projects are widely used for evaluating various IR based bug localization methods.

In the bug reports, we have the bug description (including title) and the seven types of metadata. For the bug description, we use bag-of-words to extract word tokens, and then use standard natural language processing steps including stemming, stopwords removal, low-frequency words and high-frequency words removal.

### B. Research Questions

We evaluate our proposed approaches and address the following research questions.

***RQ*1**: What is the predictive power of the L2SS and L2SS+ in bug localization?

We run L2SS and L2SS+ on the four projects we have mentioned above to check their effectiveness. We report the results via 10-fold cross-validation. That is, we randomly divide the data into 10 folds with similar size, choose each fold as test set and use the rest as training data, and compute the average results over these 10 experiments. For a bug report in the test set, we output the top-k source files. If the fixed files are ranked in top-k, we consider the bug report has been effectively located. We call the percentage of bug reports which can be successfully located as hit rate. As shown in the next subsection, we also consider other evaluation metrics including MAP (Mean Average Precision), Precision, and Recall.

***RQ*2**: Which metadata in bug reports is useful to improve the performance of L2SS model?

As shown in Fig. 1, the metadata in a bug report consists of component, version, hardware, operation system, severity, priority, and reporter. We add each of the seven types of metadata, respectively, to verify whether it is useful to improve the effectiveness. Further, to show the effectiveness of L2SS+, we compare it with the simple case where we directly add metadata into bug description and perform L2SS thereon.

***RQ*3**: Can L2SS and L2SS+ outperform other bug localization methods?

[2]https://bugs.eclipse.org/bugs/

We conduct experiments to compare our methods with several existing methods including revised Vector Space Model (rVSM) [6], Two-Phase model [11], and LLDA [13]. The rVSM method represents the IR techniques. The Two-Phase model also treats the problem in a supervised manner, and uses naive Bayes classifier to locate bugs. Since our method is built upon LLDA, we also verify if the proposed method can improve over LLDA.

***RQ*4**: How efficient is the proposed method?

In addition to effectiveness metrics, we are also interested in the efficiency aspect of the proposed method. An inefficient method would be costly if we need to update and retrain the model. Efficiency includes training time and predicting time.

### C. Evaluation Metrics

To measure the effectiveness of the proposed methods as well as the compared methods, we use the following metrics.

1) $Hit@k$ [7], [17] measures the accuracy of prediction results at top-k ranked files. $Hit@k$ reflects the percentage of bug reports that their fixed files are ranked in top $k$.
2) $Precision@k$ characterizes the number of correctly predicted files over the number of recommended files at top-k ranked files.
3) $Recall@k$ characterizes the number of correctly predicted files over the number of actual fixed files at top k-ranked files.
4) $MAP$ (Mean Average Precision) provides a single-figure measure for the quality of information retrieval when a query may have multiple related documents.

## V. EXPERIMENTAL RESULTS

In this section, we present the experimental results.

### A. Experimental Results for Research Questions

**RQ1**: What is the predictive power of the L2SS and L2SS+ in bug localization?

To answer this question, we present the Hit@k, Precision@k, Recall@k, and MAP results of L2SS and L2SS+ (we use L2SS+CM as an example) in Table II and Table III, respectively. In the tables, we choose $k = 1, 5, 10, 20$.

As we can see from the tables, when we set $k = 20$, over half of bug reports can get at least one corresponding buggy source file in the returned ranked list by L2SS and L2SS+ (indicated by the Hit metric), and around half bug reports can have all the related buggy source files in the returned ranked list (indicated by the Recall metric). Even when we set $k = 1$, we can hit the buggy source for over 30% bug reports in the JDT data. The recall results are smaller than the corresponding Hit results because one bug report may correspond to multiple buggy source files. Precision@k is equal to Hit@k when $k = 1$. When $k = 20$, the precision is around $0.05$ indicating that near one buggy source file is accurately found. For MAP, it can be seen as the weighted average of different Precision@k results. The MAP results of L2SS+ are larger than 0.2, indicating a fairly good performance.

TABLE II
THE PERFORMANCE OF L2SS.

| Project | Measures | Top1 | Top5 | Top10 | Top20 |
|---|---|---|---|---|---|
| JDT | Hit | 0.3218 | 0.6087 | 0.7081 | 0.7983 |
| | Precision | 0.3218 | 0.1499 | 0.0978 | 0.0630 |
| | Recall | 0.1955 | 0.4052 | 0.4959 | 0.5961 |
| | MAP | 0.3320 | | | |
| AspectJ | Hit | 0.1477 | 0.3321 | 0.4551 | 0.6141 |
| | Precision | 0.1477 | 0.0814 | 0.0592 | 0.0444 |
| | Recall | 0.0900 | 0.2278 | 0.3341 | 0.4945 |
| | MAP | 0.1918 | | | |
| PDE | Hit | 0.1936 | 0.4251 | 0.5247 | 0.6132 |
| | Precision | 0.1936 | 0.1063 | 0.0747 | 0.0492 |
| | Recall | 0.1101 | 0.2667 | 0.3541 | 0.4225 |
| | MAP | 0.2132 | | | |
| BIRT | Hit | 0.1375 | 0.3373 | 0.4453 | 0.5502 |
| | Precision | 0.1375 | 0.0684 | 0.0531 | 0.0349 |
| | Recall | 0.1006 | 0.2513 | 0.3409 | 0.4342 |
| | MAP | 0.1905 | | | |

TABLE III
THE PERFORMANCE OF L2SS+CM.

| Project | Measures | Top1 | Top5 | Top10 | Top20 |
|---|---|---|---|---|---|
| JDT | Hit | 0.3516 | 0.6469 | 0.7448 | 0.8350 |
| | Precision | 0.3516 | 0.1601 | 0.1039 | 0.0667 |
| | Recall | 0.2161 | 0.4370 | 0.5293 | 0.6328 |
| | MAP | 0.3599 | | | |
| AspectJ | Hit | 0.1844 | 0.3541 | 0.4714 | 0.6176 |
| | Precision | 0.1844 | 0.0857 | 0.0614 | 0.047 |
| | Recall | 0.1187 | 0.2443 | 0.3479 | 0.4996 |
| | MAP | 0.2148 | | | |
| PDE | Hit | 0.2443 | 0.5078 | 0.5760 | 0.6592 |
| | Precision | 0.2443 | 0.1241 | 0.0858 | 0.0571 |
| | Recall | 0.1435 | 0.3100 | 0.3987 | 0.4934 |
| | MAP | 0.2575 | | | |
| BIRT | Hit | 0.1868 | 0.4187 | 0.5345 | 0.6470 |
| | Precision | 0.1868 | 0.0962 | 0.0635 | 0.0442 |
| | Recall | 0.1367 | 0.3186 | 0.4219 | 0.5347 |
| | MAP | 0.2493 | | | |

Overall, the results show that the proposed L2SS and L2SS+ can help developers accurately locate buggy files.

**RQ2**: Which metadata in bug reports is useful to improve the performance of L2SS model?

L2SS uses the bug description only, and L2SS+ carefully adds metadata into L2SS. Here, we check the usefulness of the metadata including component, version, platform, operating system, priority, severity, and reporter. We show the Hit results and MAP results in Fig. 4 and Fig. 5, respectively. For the results, we vary $k$ from 1 to 20. We omit the Precision and Recall results for brevity.

As is shown in Fig. 5, all the seven types of metadata are useful on AspectJ and PDE projects. For example, on the PDE project, the component (L2SS+CM), version (L2SS+VS), operating system (L2SS+OS), platform (L2SS+PF), operating system (L2SS+OS), severity (L2SS+SV), and priority (L2SS+PR) can improve L2SS by 20.1%, 13.8%, 5.9%, 11.1%, 8.7%, 12.6%, and 2.9% in terms of MAP, respectively. For the JDT project, six types of metadata (except the reporter metadata) are useful. For the BIRT project, only component metadata shows significant improvement. In general, component (L2SS+CM) is the most helpful metadata. Similar results are observed in the Hit metric as shown in Fig. 4.

In Fig. 4 and Fig. 5, we also report the results when we include all metadata. However, the performance is inferior to L2SS+CM. The possible reason is that some metadata may conflict against each other. Additionally, we try to add metadata into the description part and name this method as L2SS#. The result shows that the performance is similar to L2SS and inferior to L2SS+CM. This result indicates the usefulness of our treatment on the metadata.

Overall, we can conclude that all the seven types of metadata are useful to some extent especially on the JDT, AspectJ, and PDE projects. Additionally, we find that the component information is generally the most useful metadata for the bug localization task, and we recommend L2SS+CM in practice.

**RQ3**: Can L2SS and L2SS+ outperform other bug localization methods?

Next, we compare L2SS and L2SS+ (L2SS+CM) with other methods including LLDA [13], Two-Phase [11], and rVSM [6] on all the four projects[3]. We still report the Hit results and the MAP results in Fig. 6 and Fig. 7, respectively.

We can first observe from the figures that, in general, L2SS+ outperforms all the compared methods in both metrics on the four projects. L2SS can outperform the existing methods except on the PDE project. For example, in project BIRT, L2SS+ and L2SS can locate 51.1% and 42.0% bugs within the top 10 list, while LLDA, rVSM, and Two-Phase can locate 38.7%, 36.5%, and 32.4% bugs, respectively.

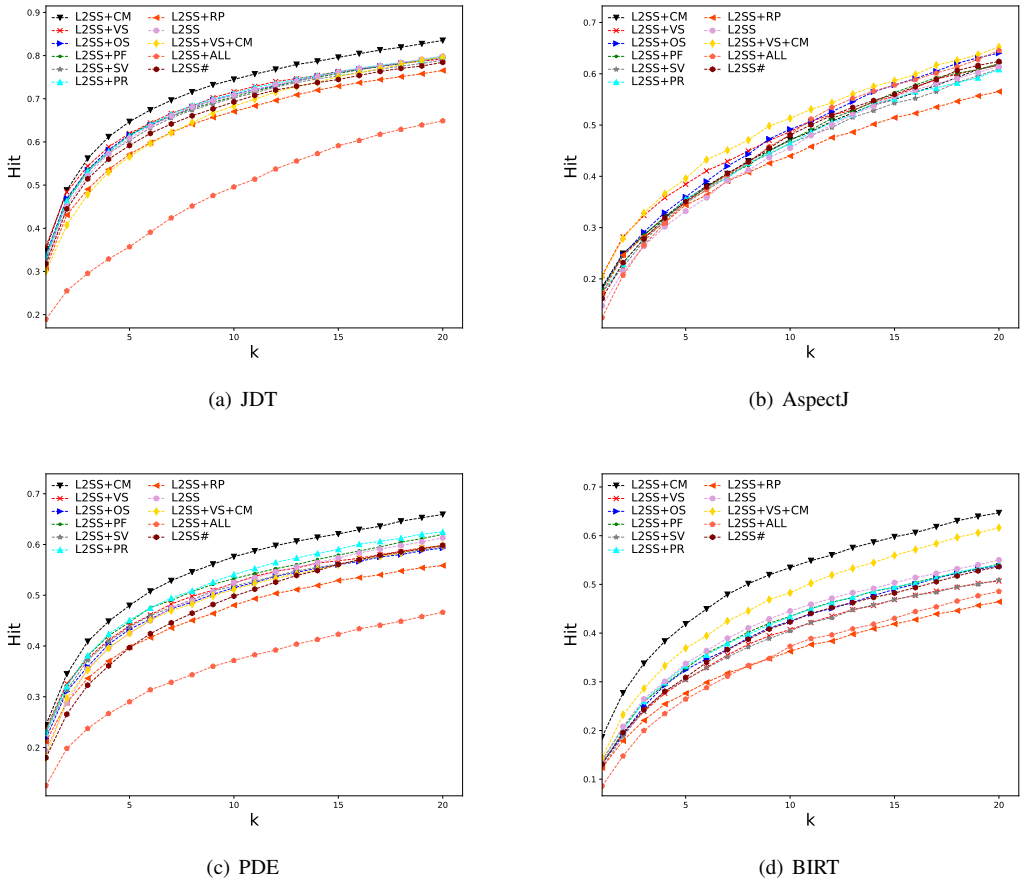
For LLDA and Two-Phase, they do not need to use the content of source files. In other words, they use the same input to L2SS and L2SS+. For rVSM, since it is a classic IR method, it uses the content in source files and compute the similarities between source files and bug reports. This is probably one of the reasons that rVSM can outperform L2SS on the PDE project (we will discuss another reason in the next subsection). This result also motivates the usage of source file content in our method, and we leave it as future work.

Overall, L2SS+ can outperform the existing methods.

**RQ4**: How efficient is the proposed method?

One advantage of the proposed method is that it does not need to compute the similarities (which is quadratic in terms of time complexity). Here, we executed L2SS and L2SS+ on a desktop computer with CPU Intel(R) Core(TM) i7 3.4 GHz and 8G RAM. Table IV presents the CPU time of L2SS and L2SS+CM. We can see in the table that it only takes seconds to run the two models (both training stage and prediction stage) even for large open source projects like JDT, which means

[3]For rVSM, we directly use the code published by the authors; for LLDA and Two-Phase, we implement them ourselves by following the descriptions in the papers.




(a) JDT (b) AspectJ

(c) PDE (d) BIRT

Fig. 4. The Hit results of L2SS+ with different metadata. Most metadata especially the component metadata is useful.

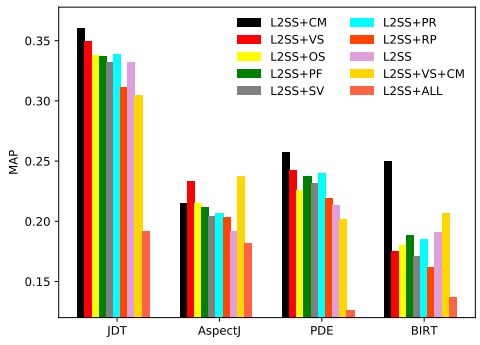



Fig. 5. The MAP results of L2SS+ with different metadata. Most metadata especially the component metadata is useful.

TABLE IV
AVERAGE TRAINING TIME AND PREDICTING TIME OF L2SS AND L2SS+CM ON THE FOUR PROJECTS.

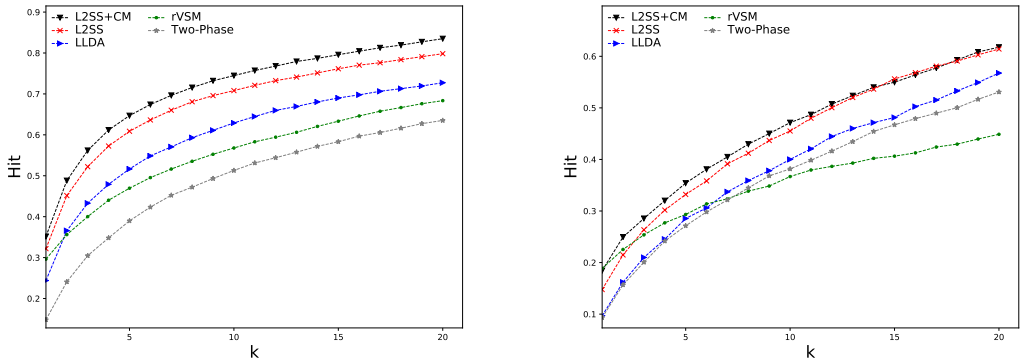
| Projects | L2SS | | L2SS+CM | |
|---|---|---|---|---|
| | training/s | predicting/s | training/s | predicting/s |
| JDT | 29.45 | 10.17 | 29.63 | 10.16 |
| AspectJ | 6.75 | 1.11 | 6.64 | 0.97 |
| PDE | 22.01 | 11.50 | 22.13 | 11.62 |
| BIRT | 14.00 | 5.53 | 13.97 | 7.20 |

that they are easy to be deployed for developers with limited computing resources.

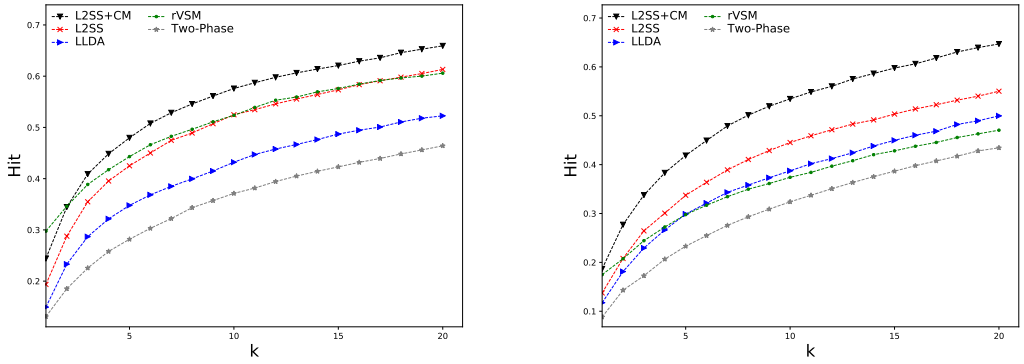
### *B. Discussions of the Results*

**a)** How does dataset affect the performance of out models?

Fig. 7 shows that LLDA can outperform rVSM in projects expect PDE. In other words, the difference in datasets may affect the performance of different methods. In Table I, we can see that in PDE, 1309 files need to be fixed for 3844 bug reports while the code repository has 2577 source files. The buggy files account for 50.8% of all source files. Each bug report is related to 2.94 buggy files. The percentage of buggy files in JDT, AspectJ, and BIRT is 43.3%, 26.3%, and 9.8%, respectively, and each bug report has 6.54, 5.15, and 3.22 buggy files, respectively. Overall, we can see that PDE has the largest percentage of buggy files and the fewest related buggy files for each bug report. This means that we have




(a) JDT (b) AspectJ




(c) PDE (d) BIRT

Fig. 6. The Hit comparisons in the four projects. L2SS+CM performs the best.

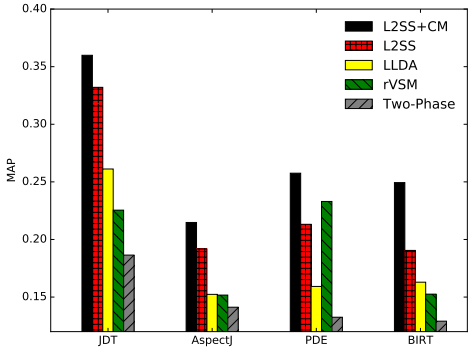



Fig. 7. The MAP comparisons in the four projects. L2SS+CM performs the best.

relatively fewer supervision labels, making L2SS less effective than rVSM.

**b)** Why we build L2SS+ upon L2SS?

L2SS is a supervised topic modeling method. We build the L2SS+ model upon L2SS for two reasons. First, L2SS can perform relatively well for the bug localization task. It formulates the problem as a supervised learning problem, and further encodes the word co-occurrence phenomenon to capture the content similarities as widely used by IR methods. Second, L2SS is flexible to incorporate additional metadata. We incorporate the metadata in the prediction stage. We may also consider to incorporate metadata in the training stage, and we leave this as future work.

**c)** How does L2SS+ improve the performance of L2SS?

As we have discussed in section III-C, bug reports with different metadata have different $p(z|m)$ in L2SS+. As shown above, the component metadata is the most useful metadata, so we take L2SS+CM as an example. In JDT, 88.1% bug reports' component are 'core'. In BIRT, bug reports with component 'Report Engine' occupy the largest proportion of 36.8%. When most of bug reports share the same component, L2SS+CM tends to have similar performance with L2SS. As we can see in Table II and Table III, L2SS+CM improve L2SS by only 2.92% when we recommend the top 10 files in project JDT, while L2SS+CM can improve L2SS by 9.1% when the distribution of components is more uniform in project BIRT. Note that we do not directly use the component to shrink the search space. The reason are as follows. First, the components are manually defined and they may not correspond to a certain collection of source files. Second, we found that many buggy

source files do not belong to the source file collection of previous bug reports with the same component. Therefore, we treat the metadata in a probabilistic way instead of directly searching a subset of source files.

In some datasets, the reporter metadata does not help improve the performance. Take the BIRT dataset as an example, 378 reporters reported 2677 bugs, while 84.7% reporters only reported fewer than 5 bug reports. We cannot learn $p(z|m)$ well from such a small dataset for most reporters. As a result, the performance of L2SS+RP may be affected.

## VI. Threats To Validity

There are potential threats to the validity of our work:

1) Bug reports provide crucial information for developers to fix bugs, and the quality of bug reports matters. In this paper, we do not consider the quality of bug reports and its influence. If a bug report contains insufficient information, our method may be misleading.
2) In our method, we treat the name of source files as supervision information in L2SS and L2SS+. In other words, we rely on good programming practice in naming classes or source files. If developers use meaningless strings to name classes and source files, our method may not have good performance.
3) We choose all datasets from open source projects. We need to evaluate if our solution can be directly applied to commercial projects in the future.

## VII. Related Work

Various related techniques and methods have been proposed for bug localization including dynamic analysis and static analysis.

Dynamic analysis (i.e., spectrum-based methods) uses a program's runtime behavior to discover bugs over multiple executions. The basic idea is to compare the program's passed executions with failed executions to help developers locate bugs. Examples of dynamic analysis include [18]–[22]. In general, dynamic analysis is expensive to apply, and static analysis is proposed by only examining program model or source files directly. In this paper, we focus on the static analysis for bug localization based on bug reports.

For static analysis, researchers mainly apply Information Retrieval techniques to locate buggy files from code repository for a given bug report. The basic idea of these techniques is to learn feature representations of bug reports and source files, and then compute the similarities between these representations. Based on the similarity computation, IR methods return the most similar source files for a given bug report.

In literature, some methods use the vector space model (VSM) to compute the textual similarity between bug reports and source files. For example, Zhou et al. [6] proposed rVSM which revises the VSM model by incorporating the similarity between bug reports. Saha et al. [23] further considered the code structure information such as variable and function names. Rao et al. [24] incorporated code change information for bug localization, where the main idea is to consider recently changed source code elements as potential bug candidates.

Some other methods use Latent Semantic Indexing (LSI) [25] and Latent Dirichlet Allocation (LDA) [5] to compute textual similarity. For example, Lukins et al. [4] directly applied LDA on bug reports and used topic distribution vector to represent bug reports and source files; then, they computed the topic distribution similarities between source files and bug reports to obtain the returned ranked list. Nguyen et al. [26] applied extended LDA model to detect latent topics from both source files and bug reports; the basic idea is to introduce a defect-proneness factor which gives frequently fixed files or large-size files more weights of being buggy.

Deep learning methods have also been used to solve bug localization problem recently. For example, Lam et al. [7] applied deep neural networks on both bug reports and source files to learn their representations/features, and combined the results with IR methods. Huo et al. [8] proposed to use convolution neural networks (CNN) to further capture the structure of source code.

Most of the existing methods follow the IR framework by similarity computation. One exception is from Kim et al. [11]. They proposed to treat the problem as classification problem (multiple classes) and used Naive Bayes to predict the potential buggy files for a bug report. They extracted features from both bug description and metadata via bag-of-words. In this paper, we follow the supervised modeling for bug localization. However, different from Kim et al. [11], we further use topic modeling based on the bag-of-words, and we propose special treatment for each type of the metadata.

One assumption of our supervised method is that the developers follow naming conventions, and the names are meaningful. Considering the case when developers fail to follow the naming conventions, similarity computation between the bug report and the source file may be meaningless. To this end, Ye et al. [9] and Almhana et al. [10] proposed to introduce API documentation of classes and methods. We may also consider to incorporate API documentation into our method in future work.

Bug localization is within a broader area of feature/concept localization [27]–[30] which associates human-oriented concepts expressed in natural language (such as functionality requirements) with their counterparts in source code. For example, Poshyvanyk et al. [31] combined LSI and Scenario-based Probablistic Ranking (SPR) to locate unwanted features (i.e., bugs). Gay et al. [32] also proposed a concept localization approach that arguments information retrieval based concept localization via an explicit relevance feedback mechanism.

## VIII. Conclusions

In this paper, we explore metadata in bug reports for the bug localization problem. In particular, we first build a supervised topic modeling method L2SS, and then add seven types of metadata into L2SS. Experimental evaluations on four real open source projects show that when we combine component metadata, L2SS+CM can have the best performance.

In particular, L2SS+CM can locate the related source files within the top 10 recommendations for over 70% bug reports in the JDT project, and over 50% bug reports in PDE and BIRT. Additionally, L2SS+CM can outperform several existing methods.

## ACKNOWLEDGMENT

We would like to thank the reviewers for their constructive comments. This work is supported by the National Key Research and Development Program of China (No. 2016YFB1000802), the National 863 Program of China (No. 2015AA01A203), the National Natural Science Foundation of China (No. 61702252, 61672274, 61690204), the Fundamental Research Funds for the Central Universities (No. 020214380033), and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

## REFERENCES

[1] P. Tiejun, Z. Leina, and F. Chengbin, "Defect tracing system based on orthogonal defect classification," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 2. IEEE, 2008, pp. 574–577.

[2] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 8.

[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE Press, 2013, pp. 712–721.

[4] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 155–164.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[6] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[7] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Proceedings of the 30st IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2015, pp. 476–481.

[8] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, 2016, pp. 1606–1612.

[9] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2016.

[10] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 286–295.

[11] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[12] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *Requirements Engineering Conference (RE), 2015 IEEE 23rd International*. IEEE, 2015, pp. 116–125.

[13] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning, "Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*. Association for Computational Linguistics, 2009, pp. 248–256.

[14] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 181–190.

[15] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh, "On smoothing and inference for topic models," in *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 2009, pp. 27–34.

[16] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling, "Fast collapsed gibbs sampling for latent dirichlet allocation," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 569–577.

[17] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 111–120.

[18] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 467–477.

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[20] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[21] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[22] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[23] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 345–355.

[24] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 43–52.

[25] S. T. Dumais, "Latent semantic analysis," *Annual review of information science and technology*, vol. 38, no. 1, pp. 188–230, 2004.

[26] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 263–272.

[27] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Identifying the starting impact set of a maintenance request: A case study," in *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*. IEEE, 2000, pp. 227–230.

[28] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 214–223.

[29] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification: a novel approach and a case study," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 357–366.

[30] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the 22nd IEEE/ACM international Conference on Automated Software Engineering*. ACM, 2007, pp. 234–243.

[31] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, 2007.

[32] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 351–360.