Bug Localization via Supervised Topic Modeling

Yaojing Wang[1], Yuan Yao[1], Hanghang Tong[2], Xuan Huo[1], Ming Li[1], Feng Xu[1], Jian Lu[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, China
[2]Arizona State University, Tempe, USA
wyj@smail.nju.edu.cn, {y.yao, xf, lj}@nju.edu.cn, hanghang.tong@asu.edu, {huox, lim}@lamda.nju.edu.cn

***Abstract*—Bug tracking systems, which help to track the reported software bugs, have been widely used in software development and maintenance. In these systems, recognizing relevant source files among a large number of source files for a given bug report is a time-consuming and labor-intensive task for software developers. To tackle this problem, information retrieval methods have been widely used to capture either the textual similarities or the semantic similarities between bug reports and source files. However, these two types of similarities are usually considered separately and the historical bug fixings are largely ignored by the existing methods. In this paper, we propose a supervised topic modeling method (STMLOCATOR[1]) for automatically locating the relevant source files for a given bug report. In particular, the proposed model is built upon three key observations. First, supervised modeling can effectively make use of the existing fixing histories. Second, certain words in bug reports tend to appear multiple times in their relevant source files. Third, longer source files tend to have more bugs. By integrating the above three observations, the proposed STMLOCATOR utilizes historical fixings in a supervised way and learns both the textual similarities and semantic similarities between bug reports and source files. We further consider a special type of bug reports with stack-traces in bug reports, and propose a variant of STMLOCATOR to tailor for such bug reports. Experimental evaluations on three real data sets demonstrate that the proposed STMLOCATOR can achieve up to 23.6% improvement in terms of prediction accuracy over its best competitors, and scales linearly with the size of the data. Moreover, the proposed variant further improves STMLOCATOR by up to 76.2% on those bug reports with stack-traces.**

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

I. Introduction

During software development and maintenance, the project team often receives and records a large number of bug reports describing the details of program defects or failures. For example, as recorded in the bug tracking system, there are over 145,000 verified bug reports in the Eclipse project. Based on these bug reports, however, it is time-consuming and labor-intensive for developers to manually recognize the relevant buggy source files and fix the bugs therein [1]. For example, among the 9,000 bug reports we collected from the Eclipse project, it takes 86 days on average to fix a single bug. Therefore, automatically locating the relevant source files for a given bug report is crucial to improve the efficiency of software development and maintenance. We refer to this problem as *bug localization* in this work.

[1]The code is available on https://github.com/Ghostcandywyj/STMLocator

In recent years, Information Retrieval (IR) methods have been used to automatically identify the relevant source files based on bug reports (see the related work section for a review). The basic idea is to identify the possible buggy source files based on their content similarities to the bug reports [2]. There are basically two types of content similarities used in literature: *textual similarities* which can be captured by the vector space, and *semantic similarities* which can be learned by the topic models.

Despite the success of existing IR methods, they typically suffer from the following two limitations. First, the existing bug fixing histories are either ignored or used in an unsupervised way (e.g., finding the source files of similar bug reports). From data mining perspective, using such fixing histories as supervision information can potentially achieve higher localization accuracy compared to the unsupervised IR methods. Second, the textual similarity and the semantic similarity are usually considered separately, although these two types of content similarities are inherently complementary to each other.

In this paper, we propose a supervised topic modeling method (STMLOCATOR) for automatically locating the relevant buggy source files for a given bug report. In particular, the proposed STMLOCATOR consists of three major components. First, to make use of the historical fixings, we encode them as supervision information in a generative model. Second, to seamlessly integrate the textual similarity and semantic similarity, we adopt topic modeling to capture semantic similarity and further incorporate the textual similarity by modeling the word co-occurrence phenomenon. Here, word co-occurrence means that some words have appeared in both the bug reports and the source files, and we provide empirical validation of this phenomenon. Third, we encode the length of source files (e.g., lines of codes) into the model as longer source files tend to have more bugs. The empirical validation on this longer-file phenomenon is also provided. Based on the proposed STMLOCATOR, we further consider a special type of bug reports with stack-traces (e.g., see Figure 4). Since the stack-traces may directly contain the names of buggy source files, we use regular expression to match the source files in stack-traces and propose a variant of STMLOCATOR to integrate the results from regular expression matching and STMLOCATOR for the bug reports with stack-traces. Finally, we conduct experimental evaluations on three real data sets, and the results demonstrate the effectiveness and efficiency of the proposed method.

The main contributions of this paper include:

978-1-5386-9159-5/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDM.2018.00076

TABLE I: Notations.

Symbol	Description
M	# of bug reports
K	# of topics/source files
V	the vocabulary
$D = \{d_1, d_2, ..., d_M\}$	bug report collection
$S = \{s_1, s_2, ..., s_K\}$	source file collection
$d = \{w_1, w_2, ..., w_{N_d}\}$	a bug report d
$s = \{w_1, w_2, ..., w_{T_s}\}$	a source file s
Λ_d	relevant topics/source files for d

- A generative model STMLOCATOR for bug localization based on bug reports. The proposed STMLOCATOR model adopts supervised topic modeling and characterizes both the textual similarities and the semantic similarities between bug reports and source files. We further tailor STMLOCATOR to deal with the cases when there are stack-traces in bug reports.
- Experimental evaluations on three real data sets showing that the proposed STMLOCATOR can achieve up to 23.6% improvement in terms of prediction accuracy over its best competitors. Moreover, up to 76.2% additional improvement can be achieved by tailoring STMLOCATOR for bug reports with stack-traces.

The rest of the paper is organized as follows. Section 2 states the problem definition. Section 3 describes the proposed approach. Section 4 presents the experimental results. Section 5 covers related work, and Section 6 concludes the paper.

II. PROBLEM STATEMENT

In this section, we present the notations and problem definition. We use D to denote the collection of input bug reports[2]. Each bug report $d \in D$ contains a list of N_d words and is relevant to a list of source files Λ_d[3]. Similarly, we use S to denote the collection of input source files. Each source file $s \in S$ contains a list of T_s words. All the words in bug reports[4] form the vocabulary V. Furthermore, we use M to indicate the number of bug reports, and K to indicate the number of topics. The main symbols used in this paper are listed in Table I.

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

Problem 1. *Bug Localization Problem*

Given: *(1) a collection of bug reports D, where each bug report $d \in D$ contains N_d words and is relevant to source files Λ_d, (2) a collection of source files S, where each source file s contains T_s words, and (3) a new bug report $d_{new} \notin D$ which contains $N_{d_{new}}$ words;*

Find: *the relevant source files for the new bug report d_{new}.*

As we can see from the above problem definition, we have the words from bug reports and source files as well as the historical fixings (i.e., bug reports and their relevant source files) as input. The goal is to identify the relevant source files for a new bug report.

III. THE PROPOSED APPROACH

In this section, we present the proposed STMLOCATOR. We start with the key intuitions and observations, and then present the proposed model followed by a brief description of the learning algorithm. After that, we present a STMLOCATOR variant for bug reports with stack-traces.

A. Intuitions and Observations

Before describing the proposed STMLOCATOR model, we first present three key observations.

Observation 1: Supervised topic modeling. To make use of the existing fixing histories between source files and bug reports, as well as the rich text content in both bug reports and source files, a natural tool is supervised topic modeling. That is, we use the source files related to a bug report as its labels, and the goal is to predict the relevant source files for a new bug report. In particular, each source file is a unique topic (i.e., $K = |S|$), and we leverage the relevant source files for a bug report to guide its topics. The supervision information is encoded in Λ, where Λ_d is a vector of length K with $\Lambda_{d,s} \in \{0, 1\}$ indicating whether the source file s is relevant to bug report d.

Observation 2: Word co-occurrence phenomenon. The second key observation is the word co-occurrence in bug reports and source files, i.e., the certain words in bug reports tend to appear multiple times in their relevant source files.

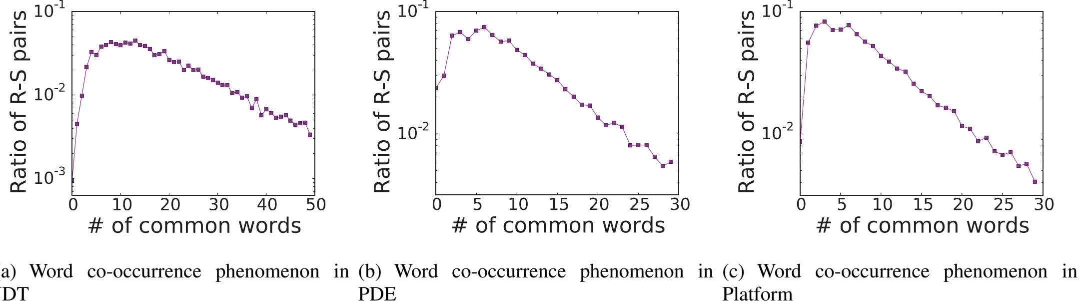
To verify the co-occurrence phenomenon, we collect three Eclipse projects (i.e., JDT, PDE, and Platform; see Section 5 for more details about the data sets). For each bug report, we study the number of words in each of its relevant source files that have also appeared in the bug report. The results of JDT, PDE and Platform data are shown in Figure 1(a), Figure 1(b), and Figure 1(c), respectively. In the figures, we denote a bug report and its related source files as 'R-S pairs'; the x-axis indicates the number of common words in a R-S pair, and the y-axis indicates the percentage of the corresponding R-S pairs. Based on the figure, over 90% R-S pairs have at least one common word, and there are 20, 11, and 10 common words on average for R-S pairs in the JDT, PDE, and Platform, respectively. Therefore, we can conclude that the word co-occurrence phenomenon widely exists in our bug localization data sets. We will explicitly model this phenomenon in our STMLOCATOR.

Observation 3: Longer-file phenomenon. Intuitively, longer source files tend to have more bugs [3]. To verify such phenomenon, we calculate the Spearman correlation coefficients between the length of a source file (i.e., LOC) and the number of bugs in the source file (i.e., the number of relevant bug reports). The results on JDT, PDE, and Platform data are shown in Figure 2(a), Figure 2(b), and Figure 2(c), respectively. The x-axis in the figures is the length of source files, and the y-axis is the number of bugs in the corresponding

[2]In this paper, we interchangeably use 'document' and 'bug report'.

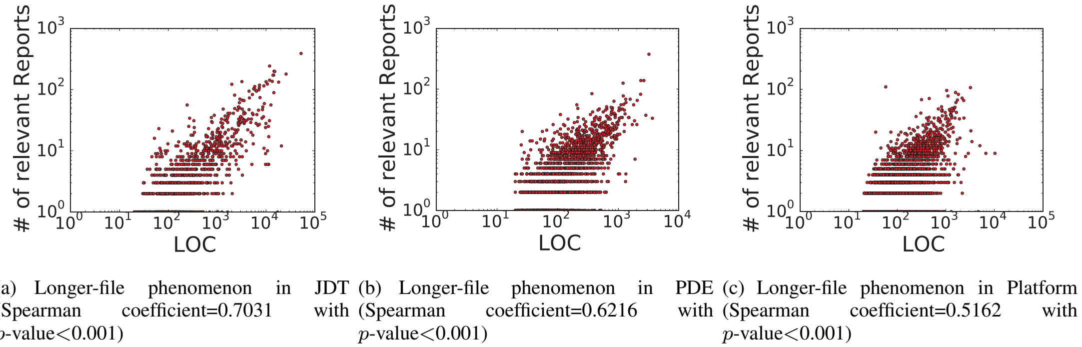
[3]A bug report may relate to multiple source files.

[4]To simplify the processing of source files, we only keep the words in source flies that have appeared in the bug reports.



(a) Word co-occurrence phenomenon in JDT
(b) Word co-occurrence phenomenon in PDE
(c) Word co-occurrence phenomenon in Platform

Fig. 1: Word co-occurrence phenomenon. That is, most of the words in bug reports have appeared in source files. This phenomenon widely exists in the three data sets.



(a) Longer-file phenomenon in JDT (Spearman coefficient=0.7031 with p-value<0.001)
(b) Longer-file phenomenon in PDE (Spearman coefficient=0.6216 with p-value<0.001)
(c) Longer-file phenomenon in Platform (Spearman coefficient=0.5162 with p-value<0.001)

Fig. 2: Longer-file phenomenon. That is, longer source files tend to have more bugs. This phenomenon holds for all the three data sets.

source files. As we can see from the figures, there is a significant positive correlation (i.e., the Spearman correlation coefficient is larger than 0.5) between the length of source files and the number of bugs. We will encode the length of source files as prior in our model.

B. The STMLocator Model

Next, we describe the proposed STMLOCATOR model which combines the above three observations together. Figure 3 shows the overall graphical representation for STMLOCATOR. Corresponding to the above three observations, there are three integral parts in the STMLOCATOR model.

- **Supervised topic modeling.** First, STMLOCATOR builds upon the LDA model [4] and its generalized version LLDA [5] for supervised topic modeling. For each bug report, LDA assumes that it has several latent topics (θ). Words in the bug report are generated from a specific topic (z) and the topic-word distributions (Φ). Then, we assume that the relevant source files (Λ) of the given bug report determine the latent topics during the generative process. Here, each source file corresponds to one unique topic.
- **Modeling word co-occurrence phenomenon.** Next, to characterize the word co-occurrence phenomenon, when generating a word, we either generate it from the topics or directly from the co-occurrence words. Specially, we introduce a latent variable x to indicate the probability that the word w is generated by the co-occurrence words in both source files and bug reports, or by the topic-word distribution Φ. When the word is generated by the co-occurrence words, we use the specific topic/source file z and the topic-word distribution Ω. The latent variable x is sampled from a Bernoulli distribution Ψ, and it is dependent on the specific topic z (i.e., different topics have different probabilities).
- **Modeling longer-file phenomenon.** Finally, to model the longer-file phenomenon, we introduce an asymmetric Dirichlet prior (l) to indicate the different probabilities to choose different topics/source files for each bug report. The intuition is that, if a source file has a longer length, it is likely to contain more bugs and thus has a higher probability to be chosen as the specific topic z.

Although the supervised topic modeling only allows predicting the source files with bugs before, the word co-occurrence phenomenon extends the prediction to non-buggy source files with the help of the co-occurrence words.

In practice, there are several design choices to set the length of source files (e.g., linear or logarithmic) and the range of co-occurrence words (e.g., identifiers or annotations). We will experimentally evaluate these choices in our experiments.

The generative process of STMLOCATOR is shown below.

1) **Draw Dirichlet prior**
    a) Draw asymmetric prior $l \sim Dir(\alpha')$

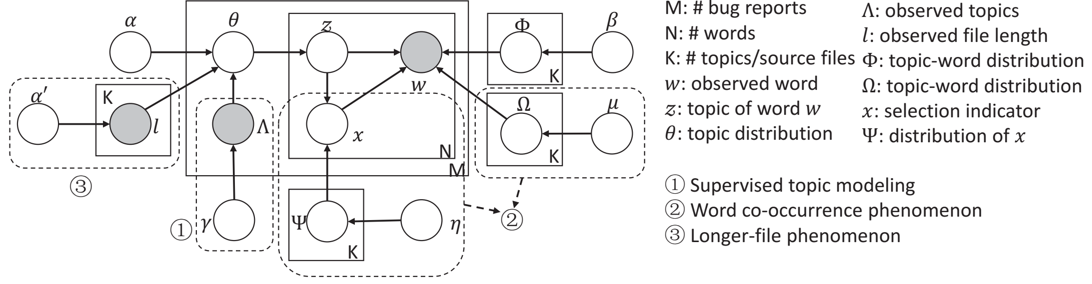


Fig. 3: Graphical representation of STMLOCATOR.

2) **Draw topic-word distributions**
   a) For each topic $k \in [1, K]$:
      i) Draw probability distribution $\Psi_k \sim Beta(\eta)$
      ii) Draw topic-word distribution $\Phi_k \sim Dir(\beta)$
      iii) Draw topic-term distribution $\Omega_k \sim Dir(\mu)$
3) **Draw words for each document** $d \in [1, M]$
   a) For each topic $k \in [1, K]$:
      i) Draw $\Lambda_{d,k} \in \{0, 1\} \sim Bernoulli(\gamma)$
   b) Draw Dirichlet prior $\alpha_d = \Lambda_d \circ \alpha \cdot l$
   c) Draw topic distribution $\theta_d \sim Dir(\alpha_d)$
   d) For each word $i \in [1, N_d]$:
      i) Draw topic $z_i \sim Mult(\theta_d)$
      ii) Draw potential word from Ω_{z_i} distribution $w_{i,\Omega} \sim Mult(\Omega_{z_i})$
      iii) Draw potential word from Φ_{z_i} distribution $w_{i,\Phi} \sim Mult(\Phi_{z_i})$
      iv) Draw $x_i \in \{0, 1\} \sim Bernoulli(\Psi_{z_i})$
      v) Draw the final word $w_i = (w_{i,\Omega})^{x_i} \cdot (w_{i,\Phi})^{1-x_i}$

In the above generative process, Step 1 draws an asymmetric prior l from a Dirichlet prior α'. l indicates the weight of each source file and it satisfies

$$\sum_{k=1}^{K} \alpha' l_k = K\alpha'. \tag{1}$$

C. Learning Algorithm

For the learning algorithm of STMLOCATOR, we first need the computation of the following joint likelihood of the observed variables (i.e., L, W, and Λ) and unobserved variables (i.e., Z and X).

$$\begin{aligned} p(Z, X, W, \Lambda, L) =& p(Z|\alpha L, \Lambda) \cdot p(L|\alpha') \cdot \\ & p(X|\eta, Z) \cdot p(\Lambda|\gamma) \cdot \\ & p(W|Z, X, \beta) \cdot p(W|Z, X, \mu). \end{aligned} \tag{2}$$

Then, we can use the above likelihood to derive update rules for θ, Ψ, Ω, and Φ. In the model, $p(L|\alpha')$ and $p(\Lambda|\gamma)$ are constants and they can be ignored in the inference[5].

[5]We incorporate these two terms in the model for completeness

For Eq. (2), we first have

$$p(Z|\alpha L, \Lambda) = \prod_{m=1}^{M} p(Z_m|\alpha L, \Lambda) = \prod_{m=1}^{M} \frac{\Delta(N_m + \alpha L)}{\Delta(\alpha)}, \tag{3}$$

where N_m indicates the number of words in document m, and L indicates the length of each source file in Z. The above equations can be derived by expanding the probability density expression of Dirichlet distribution, and applying the standard Dirichlet multinomial integral.

Next, for $p(X|\eta, Z)$, we have

$$p(X|\eta, Z) = \prod_{k=1}^{K} \frac{B(N_k + \eta)}{B(\eta)}, \tag{4}$$

where N_k incidates the number of words that belong to topic k, and X is composed of 0s or 1s to determine where a word is generated from. In following equations, we divide N_k into two parts: $N_{k,1}$ and $N_{k,0}$. $B(\eta)$ is a normalization constant to ensure that the total probability integrates to 1. $B(\eta)$ is defined as $B(\eta) = \frac{\Gamma(\eta_1)\Gamma(\eta_0)}{\Gamma(\eta_1 + \eta_0)}$.

Finally, for $p(W|Z, X, \beta, \eta)$, we have

$$\begin{aligned} & p(W|Z, X, \beta, \mu) \\ &= \int p(W|Z, X, \Phi) p(\Phi|\beta)\, d\Phi \int p(W, |Z, X, \Omega) p(\Omega|\mu)\, d\Omega \\ &= \prod_{k=1}^{K} \frac{\Delta(N_{k,0} + \beta)}{\Delta(\beta)} \prod_{k=1}^{K} \frac{\Delta(N_{k,1} + \mu)}{\Delta(\mu)}, \end{aligned} \tag{5}$$

where $N_{k,1}$ is the number of words generated by topic-word distribution Ω_k, and $N_{k,0}$ indicates the number of words generated by topic-word distribution Φ_k. The computation of $p(W|Z, X, \Phi)$ and $p(\Phi|\beta)$ in the above equation is similar to that of the traditional LDA model. For Eq. (5), when X is set to 1, $\prod_{k=1}^{K} \frac{\Delta(N_{k,0} + \beta)}{\Delta(\beta)}$ will be a constant; when X is set to 0, $\prod_{k=1}^{K} \frac{\Delta(N_{k,1} + \mu)}{\Delta(\mu)}$ will be a constant.

Putting the above equations together, we have

Algorithm 1 The Learning Algorithm for STMLOCATOR

Input: Collection of bug reports D and source files S
Output: θ, Ψ, Φ, Ω

1: Initialize topic $z_{m,i}$ for all m and i randomly;
2: **while** not convergent **do**
3:   **for** document $m \leftarrow [1, M]$ **do**
4:     **for** word $i \leftarrow [1, N_m]$ **do**
5:       $z_{m,i} \leftarrow 0$;
6:       update $n_{m,k}$, n_m, $n_{k,x}$, n_k and $n_{k,i,x}$;
7:       **for** topic $k \leftarrow [1, K]$ **do**
8:         **if** word $i \in T_k$ **then**
9:           $P(z_{m,i} = k, x = 1) \leftarrow \frac{n_{m,k}+\alpha l_k}{n_m+K\alpha} \cdot \frac{n_{k,1}+\eta_1}{n_k+\eta_0+\eta_1} \cdot \frac{n_{k,i,1}+\mu}{n_{k,1}+V\mu}$;
10:           $P(z_{m,i} = k, x = 0) \leftarrow \frac{n_{m,k}+\alpha l_k}{n_m+K\alpha} \cdot \frac{n_{k,0}+\eta_0}{n_k+\eta_0+\eta_1} \cdot \frac{n_{k,i,0}+\beta}{n_{k,0}+V\beta}$;
11:         **else**
12:           $P(z_{m,i} = k) \leftarrow \frac{n_{m,k}+\alpha l_k}{n_m+K\alpha} \cdot \frac{n_{k,i,0}+\beta}{n_{k,0}+V\beta}$;
13:         **end if**
14:       **end for**
15:       sample topic $z_{m,i}$ by $P(z_{m,i})$;
16:       update $n_{m,k}$, n_m, $n_{k,x}$, n_k and $n_{k,i,x}$;
17:     **end for**
18:   **end for**
19:   update $\theta, \Psi, \Phi, \Omega$ via Eq. (7);
20: **end while**
21: **return** $\theta, \Psi, \Phi, \Omega$

$$
\begin{aligned}
p(Z, X, W, \Lambda, L) \propto & \prod_{m=1}^{M} \frac{\Delta(N_m + \alpha L)}{\Delta(\alpha)} \cdot \prod_{k=1}^{K} \frac{B(N_k + \eta)}{B(\eta)} \\
& \cdot \prod_{k=1}^{K} \frac{\Delta(N_{k,0} + \beta)}{\Delta(\beta)} \cdot \prod_{k=1}^{K} \frac{\Delta(N_{k,1} + \mu)}{\Delta(\mu)}.
\end{aligned} \tag{6}
$$

The purpose of the training stage is to obtain θ, Ψ, Ω, and Φ, where θ represents the topic distribution of each document, and Ψ represents the distribution to indicate whether the word is generated by the topic-word distribution Ω or generated by the topic-word distribution Φ. Based on Eq (6), the equations for computing these parameters are listed as follows

$$
\begin{aligned}
\theta_{m,k} =& \frac{n_{m,k} + \alpha_k l_k}{\sum_{k'=1}^{K} (n_{m,k'} + \alpha_{k'})}, \\
\Psi_k =& \frac{n_{k,1} + \eta_1}{\sum_{x=0}^{1} (n_{k,x} + \eta_x)}, \\
\Phi_{k,v} =& \frac{n_{k,v,0} + \beta_v}{\sum_{v'=1}^{V} (n_{k,v',0} + \beta_{v'})}, \\
\Omega_{k,v} =& \frac{n_{k,v,1} + \mu_v}{\sum_{v'=1}^{V} (n_{k,v',1} + \mu_{v'})},
\end{aligned} \tag{7}
$$

where $n_{m,k}$ indicates the number of words that belong to topic k in document m, $n_{k,1}$ indicates the number of words that belong to topic k and are generated by topic-word distribution Ω, $n_{k,0}$ indicates the number of words that belong to topic k and are generated by topic-word distribution Φ, $n_{k,v,1}$ indicates the number of word v that belongs to topic k and is generated by topic-word distribution Ω, $n_{k,v,0}$ indicates the number of word v that belongs to topic k and is generated by topic-word distribution Φ.

```
     When finding references  Java Search fails with NullPointerException, I receive the following error when trying to find
     references to anything: An internal error occurred during: ”Java Search”.java.lang.NullPointerException
     I have deleted my workspace created a new one and still am receiving this issue. Here is the stack trace:
     java.lang.NullPointerException:
1    at org.eclipse.core.runtime.Path.<init>(Path.java:183)
2    at org.eclipse.core.internal.resources.WorkspaceRoot.getProject(WorkspaceRoot.java:182)
3    at org.eclipse.jdt.internal.core.JavaModel.getJavaProject(JavaModel.java:169)
4    at org.eclipse.jdt.internal.core.search.IndexSelector.getJavaProject(IndexSelector.java:304)
5    at org.eclipse.jdt.internal.core.search.IndexSelector.initializeIndexLocations(IndexSelector.java:232)
6    at org.eclipse.jdt.internal.core.search.IndexSelector.getIndexLocations(IndexSelector.java:294)
7    at org.eclipse.jdt.internal.core.search.JavaSearchParticipant.selectIndexURLs(JavaSearchParticipant.java:148)
8    at org.eclipse.jdt.internal.core.search.PatternSearchJob.getIndexes(PatternSearchJob.java:84)
9    at org.eclipse.jdt.internal.core.search.PatternSearchJob.ensureReadyToRun(PatternSearchJob.java:52)
10   at org.eclipse.jdt.internal.core.search.processing.JobManager.performConcurrentJob(JobManager.java:174)
11   at org.eclipse.jdt.internal.core.search.BasicSearchEngine.findMatches(BasicSearchEngine.java:215)
12   at org.eclipse.jdt.internal.core.search.BasicSearchEngine.search(BasicSearchEngine.java:516)
13   at org.eclipse.jdt.core.search.SearchEngine.search(SearchEngine.java:584)
14   at org.eclipse.jdt.internal.ui.search.JavaSearchQuery.run(JavaSearchQuery.java:144)
15   at org.eclipse.search2.internal.ui.InternalSearchUI$InternalSearchJob.run(InternalSearchUI.java:91)
16   at org.eclipse.core.internal.jobs.Worker.run(Worker.java:54)
```

Fig. 4: An example bug report that contains a stack-trace.

Algorithm. Based on Eq. (7), Gibbs sampling is widely used to train the parameters. The algorithm is summarized in Alg. 1. The $z_{m,i}$ in the algorithm indicates the topic that word i in bug report m belongs to, and n_k indicates the number of words that belong to topic k. In the algorithm, Line 1 initializes all the $z_{m,i}$ for each word with a random topic. Lines 2-20 iteratively estimate the parameters based on Gibbs sampling. Line 6 and Line 16 update the statistical variables $n_{m,k}$, n_m, $n_{k,x}$, n_k, and $n_{k,i,x}$ which are computed based on $z_{m,i}$. Line 19 updates the four parameters via Eq (7). The iterative process terminates when the parameters converge or when the maximum iteration number is reached.

In practice, Gibbs sampling is inherently stochastic and unstable, while the CVB0 learning algorithm [6] converges faster and is more stable [7]. Therefore, we further build the learning algorithm based on CVB0 learning, and the details are omitted for brevity.

Time Complexity. In short, the time cost of the learning algorithm scales linearly wrt the data size (e.g., the number of topics/source files and the total number of words in the bug reports). We will experimentally validate the time complexity of the learning algorithm in the experiments.

Prediction Stage. Finally, based on the learned parameters, we can predict the buggy files for a given bug report d_{new} as follows,

$$
\begin{aligned}
p(t|d_{new}) &= \sum_{w} p(t|w) \cdot p(w|d_{new}) \\
&= \sum_{w} \frac{p(w|t)p(t)}{p(w)} \cdot p(w|d_{new}).
\end{aligned} \tag{8}
$$

We omit the detailed computations for brevity.

D. STMLocator Variant for Stack-Traces

Here, we present a variant of STMLOCATOR. For some bug reports, the reporters may include the stack-traces of the bugs. An example of bug reports containing stack-traces

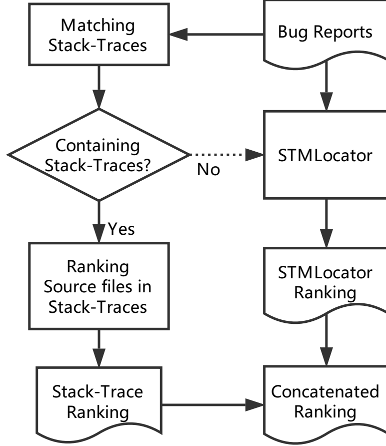


Fig. 5: STMLOCATOR variant for bug reports with stack-traces.

(each stack-trace is a list of method calls that trigger an exception) is shown in Figure 4, where the software throws a 'NullPointerException' and outputs the sequence of method calls. The related classes (i.e., source file names) are colored in red. Intuitively, the source files in the stack-traces are highly related to the bug report and possibly contain the buggy source files. For example, the buggy source file is 'org.eclipse.jdt.internal.core.search.IndexSelector.java' in the above example which appears in Lines 4-6 of the stack-trace. Based on this observation, we present a variant of STMLOCATOR as shown in Figure 5.

To be specific, we first recognize whether a bug report contains a stack-trace, and then extract all the source file names in the stack-trace. These two steps are accomplished by regular expression matching. Next, we rank these source files according to their number of occurrences in a descending order. For those source files have the same number of occurrences, we rank them by the original up-down order in the stack-trace. Finally, based on the ranking of above method, we combine it with STMLOCATOR to obtain the final ranking list (which contains the possibly related buggy files) for bug reports with stack-traces. Specially, we combine the ranking list from stack-traces (referred to as R_{ST}) and the ranking list generated by STMLOCATOR (referred to as R_{STM}) as follows,

$$R(n) = concat(R_{ST}(1 : \epsilon \cdot n), R_{STM}(1 : (1 - \epsilon) \cdot n)), \quad (9)$$

where n indicates the top-n results as output, ϵ is the proportion of source files we select from the stack-trace, and $concat(\cdot, \cdot)$ concatenates the two lists into one list. For the bug reports without stack-traces, we simply set $\epsilon = 0$, i.e., only the results from STMLOCATOR are used as output.

IV. EXPERIMENTAL EVALUATIONS

In this section, we present the experimental results. The experiments are mainly designed to answer the following questions:

TABLE II: Statistics of the data sets.

Data set	# reports	# sources	vocabulary size
PDE	3900	2319	2964
Platform	3954	3696	3677
JDT	6267	7153	4304

- *Effectiveness*: How accurate is the proposed method for bug localization?
- *Efficiency*: How scalable is the proposed method for bug localization?

A. Data sets

We collect the data sets from three open-source projects, i.e., PDE, Platform, and JDT. All the data sets are collected from the official Bug Tracking Website of Eclipse[6] and the Eclipse Repository[7]. The statistics of the data sets are shown in Table II.

For each project, we collect the bug reports (each bug report contains bug id, title, and description) from the bug tracking website. Then, we collect the corresponding source files from the git repository by bug id. For each bug report, we combine its title and description as its content. We adopt standard NLP steps including stop-words removal, low-frequency and high-frequency words removal to reduce noise. In bug reports, there are many combined words (e.g., 'updateView'). We separate these words into simple words (e.g., 'update' and 'view') while keeping the combined words at the same time. For the source files, there are typically two types of words, i.e., annotations and source code. For annotations, we adopt the same processing steps with bug reports. For source code, we additionally remove the keywords (e.g., 'for' and 'if') and extract identifiers (i.e., variable and function names) before adopting the processing steps. The identifiers are also separated from combined words to simple words.

B. Experimental Setup

For the three data sets described above, we follow existing experiment framework [8] of 10-fold cross validation. For the test set, we sort the source files based on the predicted probabilities in a descending order, and use the ranking list as output. For the hyper-parameters, we fix $\alpha = 200/K$, $\eta = 0.01$, and $\beta = \mu = 0.1$.

Evaluation Metrics. For the evaluation metrics, we first adopt Hit@n for effectiveness comparison. Hit@n is defined as follows,

$$Hit@n = \sum_{d=1}^{M_{test}} \frac{hit_{n,d}}{M_{test}},$$

$$hit_{n,d} = \begin{cases} 1, & hit(n, d) > 0, \\ 0, & hit(n, d) = 0, \end{cases}$$

where $hit(n, d)$ is the hit number of source files that have been successfully recommended in the top-n ranking list for the d-th bug report, $hit_{n,d}$ indicates whether the top-n ranking list

[6] https://bugs.eclipse.org/
[7] http://git.eclipse.org/,https://github.com/eclipse/

contains a hit or not, and M_{test} is the number of bug reports in the test data. Note that Hit@n cares about whether there is a hit or not. The reason is that finding one of the buggy source files will help developers find other relevant buggy source files [3].

We also use the Mean Reciprocal Rank (MRR) to evaluate the quality of the ranking list. The MRR is defined as

$$MRR = \frac{1}{M_{test}} \sum_{i}^{M_{test}} \sum_{j \in \Lambda_i} \frac{1}{rank(j)},$$

where Λ_i indicates the relevant source files of document i, and $rank(j)$ indicates the rank postition of source file j in the ranking list for bug report i. Larger MRR value is better. Both Hit@n and MRR are widely used in other studies [3], [9], [10].

In addition to Hit@n and MRR, developers also care about the position of the buggy files in the ranking list. The higher the first hit in the ranking list, the fewer source files that the developers would need to check. Therefore, we adopt the AFH@n (Average First Hit at Top-n) for measuring the workload of developers. AFH@n is defined as

$$AFH@n = \frac{\sum_{d=1}^{M_{test}} \delta_n(pos_d)}{M_{test}},$$

$$\delta_n(pos_d) = \begin{cases} pos_d, & pos_d \leqslant n, \\ n, & pos_d > n, \end{cases}$$

where pos_d indicates the first hit in the ranking list for document/bug report d. Smaller AFH is better.

As to the list size n, we choose $n = 5$ and $n = 10$ for *Hit@n*, and $n = 10$ for *AFH@n* as such choices will not cause many burdens to the developers. For efficiency, we record the wall-clock time of the proposed algorithm. All the experiments were run on a machine with eight 3.5GHz Intel(R) Xeon(R) and 64GB memory.

Compared Methods. To evaluate the effectiveness of the proposed method, we compare the following methods in our experiments.

- *LLDA* [5]: LLDA is a supervised generative model proposed for tag recommendation. It can be seen as a special case of STMLOCATOR by ignoring word co-occurrence phenomenon and longer-file phenomenon.
- *tag-LDA* [11]: tag-LDA is another generative model for tag recommendation. The basic idea of tag-LDA is to combine two LDA models with the same θ value. It can be similarly adapted for bug localization by treating source files as tags.
- *VSM* [12]: VSM model embeds each document (both bug report and source file) into a vector, and then uses the cosine distance between vectors to identify the most similar source files for a given bug report.
- *rVSM* [3]: rVSM (which is also known as BugLocator) is built upon VSM. It further finds similar bug reports and uses their relevant source files as output. The LOC of each source file is also considered.
- *NP-CNN* [8]: NP-CNN is a deep learning based method to locate bugs. It uses a CNN network to train the features/embeddings of source files, and then calculates the similarities between embeddings to obtain the ranking list.
- *LLDA+W*: LLDA+W is a special case of STMLOCATOR. It combines the supervised modeling and word co-occurrence phenomenon.
- *LLDA+S*: LLDA+S is a special case of STMLOCATOR by combining the supervised modeling and longer-file phenomenon.
- STMLOCATOR: STMLOCATOR is the proposed method that combines supervised modeling, word co-occurrence phenomenon, and longer-file phenomenon.

C. *Effectiveness Results*

(A) Effectiveness Comparisons. For effectiveness, we first compare the proposed STMLOCATOR with several existing methods. In the compared methods, LLDA and tag-LDA use topic models, VSM and rVSM are textual models, and NP-CNN applies deep convolutional neural networks. The results are shown in Table III, where the relative improvements compared to the best competitors are also shown in the brackets.

We can first observe from Table III that STMLOCATOR outperforms all the compared methods on all the three data sets. For example, on PDE data set, STMLOCATOR achieves 3.2% and 3.8% relative improvements on Hit@5 and Hit@10 over its best competitor. On Platform data set, STMLOCATOR improves its best competitor by 4.9% and 2.0% wrt Hit@5 and Hit@10, respectively. On JDT data set, the improvement of STMLOCATOR over the best competitor is 13.1% and 0.8% wrt Hit@5 and Hit@10, respectively. Similarly, STMLOCATOR achieves the highest MRR values and smallest AFH@10 values compared to other methods as shown in Table III. For example, STMLOCATOR achieves up to 23.6% improvement wrt MRR over its best competitors.

For all the reported results above, we conduct paired t-test on the average rankings, and the results show that the improvements on all three data sets are statistically significant, with p-values less than 0.001.

In the compared methods, VSM and rVSM consider the textual similarity and ignore the semantic similarity, while tag-LDA and LLDA consider semantic similarity and ignore textual similarity, and thus they are less effective than STMLOCATOR. This result indicates the usefulness of combining textual similarity with semantic similarity. LLDA is a supervised topic model and it can be seen as a special case of STMLOCATOR by ignoring word-occurrence and source file length. This result further indicates the usefulness of modeling the two corresponding observations. Our method also outperforms the NP-CNN method. The possible reason is that NP-CNN may need a large volume of data to avoid overfitting.

(B) Performance Gain Analysis. Next, since STMLOCATOR has three integral building blocks, we further study the effectiveness of each block. The results are also shown in Table III.

TABLE III: Effectiveness comparisons of Hit@n, AFH@n and MRR on three data sets. The proposed STMLocator outperforms the compared methods. (For Hit@N and MMR, larger is better. For AFH@n, smaller is better.)

Methods		LLDA	tag-LDA	VSM	rVSM	NP-CNN	LLDA+W	LLDA+S	STMLocator
PDE	Hit@5	0.347	0.204	0.276	0.443	0.375	0.404	0.384	**0.457** (3.16%)
	Hit@10	0.434	0.284	0.338	0.523	0.516	0.483	0.457	**0.543** (3.82%)
	AFH@10	6.569	7.341	7.147	6.507	6.469	6.827	6.562	**6.328** (2.18%)
	MRR	0.288	0.192	0.231	0.329	0.323	0.314	0.291	**0.346** (5.16%)
Platform	Hit@5	0.422	0.364	0.332	0.612	0.489	0.525	0.463	**0.642** (4.90%)
	Hit@10	0.527	0.422	0.392	0.689	0.643	0.612	0.555	**0.703** (2.03%)
	AFH@10	5.781	6.473	6.224	5.593	5.478	5.676	5.779	**5.247** (4.22%)
	MRR	0.335	0.301	0.283	0.443	0.401	0.398	0.372	**0.494** (11.5%)
JDT	Hit@5	0.333	0.152	0.203	0.344	0.337	0.381	0.364	**0.389** (13.1%)
	Hit@10	0.425	0.212	0.271	0.442	0.488	0.482	0.435	**0.492** (0.82%)
	AFH@10	6.703	7.483	7.372	6.749	6.843	6.713	6.686	**6.668** (0.53%)
	MRR	0.218	0.112	0.127	0.241	0.234	0.274	0.223	**0.298** (23.6%)

TABLE IV: Results on different design choices of length functions.

Length function	Experession	MRR
Linear	$f(x) = x$	0.346
Logarithmic	$f(x) = log(x)$	0.345
Exponential	$f(x) = e^x$	0.331
Square root	$f(x) = \sqrt{x}$	0.343

TABLE V: Results on different design choices of co-occurrent words.

Co-occurrent words	Words	MRR
(1)	Identifiers	0.325
(2)	Annotations	0.312
(1)+(2)	Identifiers + Annotations	0.346

In the table, the results when the source file length or the word-occurrence is not incorporated are denoted as 'LLDA+W' and 'LLDA+S', respectively.

As we can see from the table, both LLDA+W and LLDA+S are better than LLDA, indicating the usefulness of both components. Additionally, STMLocator significantly outperforms all its sub-variants. For example, on the Hit@10 metric of the Paltform data, LLDA+W and LLDA+S improves LLDA by 16% and 5.3%, respectively; STMLocator improves LLDA+W and LLDA+S by 14.8% and 26.5%, respectively.

(C) The Effect of Different Design Choices. Next, as mentioned before, there are several design choices in terms of how to set the length of source files and how to determine the range of co-occurrence words. In this part, we consider the following choices.

- *Source file length*. To determine the source file length, we consider several length functions as shown in Table IV.
- *Co-occurrent words*. The source file contains several types of information. In this work, we consider the words from variable/function identifiers and the annotations as shown in Table V.

The results are shown in Table IV and V, respectively. Here we only report the MRR results on PDE data set for brevity. Similar results are observed on JDT and Platform as well as on other metrics.

As we can see from Table IV, we experiment with four length functions including linear, logarithmic, exponential, and square root. These functions weight source file length to different scales. The results show that most functions have close performance. This indicates that the proposed method is robust wrt the difference choices source file length function. In our experiments, we use linear function.

TABLE VI: Hit@10 results on the ST cases and the NST cases.

Methods	PDE		Platform		JDT	
	ST	NST	ST	NST	ST	NST
LLDA	.38	.44	.32	.57	.31	.46
tag-LDA	.27	.28	.38	.43	.21	.21
VSM	.26	.35	.26	.42	.22	.28
rVSM	**.47**	.53	**.65**	.70	.38	.46
NP-CNN	.45	.55	.61	.65	**.50**	.48
STMLocator	.39	**.57**	.57	**.73**	.42	**.53**

In Table V, we change the range of co-occurrent words. We consider three cases: using identifiers only, using annotations only, and using both identifiers and annotations. As we can see from the table, combining both identifiers and annotations can produce the best MRR result. In other words, this result indicates that both identifiers and annotations are useful for our bug localization problem.

(D) The Effect of Stack-traces. As mentioned above, a bug report may contain program stack-traces. Intuitively, textual similarity may be more suitable for such bug reports, as the buggy file names may have already been included in the stack-traces. Here, we split the bug reports into two parts: with stack-traces (denoted as 'ST') and without stack-traces (denoted as 'NST'), and then compare the Hit@10 results on these two parts. Based on our split, there are 18.6%, 20.5%, and 26.2% ST bug reports in PDE, Platform, and JDT, respectively. The results are shown in Table VI. As we can see, STMLocator performs better than the compared methods for the NST case. For the ST case, rVSM performs relatively well, which is consistent with our intuition. This result indicates that better combinations of textual methods and semantic methods can be explored to further improve localization accuracy, especially for the bug reports containing stack-traces.

(E) The Effectiveness of the STMLocator *Variant for ST Cases*. For the ST case (i.e., bug reports with stack-traces), we use the variant of STMLocator in Figure 5 to generate the top-10 ranking list. We set the parameter ϵ in Eq. (9) as 1, 0, and 0.5, standing for the ranking results from stack-trace only, from STMLocator only, and from the combination of the above two ranking results, respectively. The Hit@10 results

TABLE VII: Hit@10 results of the STMLOCATOR variant on ST bug reports and all bug reports. The variant can significantly improve the localization accuracy for ST bugs.

Data	Hit@10		
	$\epsilon = 1$	$\epsilon = 0$	$\epsilon = 0.5$
PDE ST bugs	0.27	0.39	0.49
Platform ST bugs	0.58	0.57	0.62
JDT ST bugs	0.65	0.42	0.74
PDE all bugs	0.51	0.54	0.55
Platform all bugs	0.70	0.70	0.71
JDT all bugs	0.56	0.49	0.58

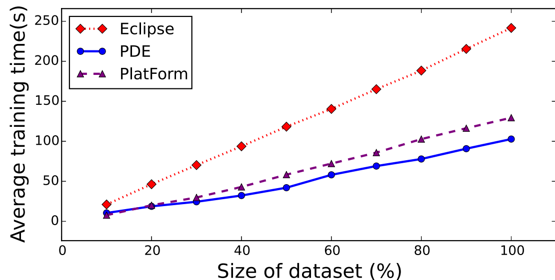


Fig. 6: Scalability of STMLOCATOR. It scales linearly wrt the data size.

are shown in Table VII, where we report the results on the bug reports that contain stack-traces and the results over all bug reports.

As we can see, the ranking results only from the stack-traces already perform better than STMLOCATOR in some cases. For example, on JDT, it achieves 54.7% relative improvement compared to STMLOCATOR. Combining stack-traces with STMLOCATOR can achieve the best results. For example, for the ST case, it achieves additional 25.6%, 8.8%, and 76.2% relative improvements compared to STMLOCATOR on PDE, Platform, and JDT, respectively. This result indicates that leveraging the stack-traces can further improve the localization accuracy. As to the improvement of the STMLOCATOR extension over all bug reports, it can also achieve 18.3% relative improvement on the JDT data due to the relatively higher proportion of ST cases.

D. Efficiency Results

(F) Scalability. Finally, we study the scalability of the proposed method in the training stage. We vary the size of training data, and report the results on the three data sets in Figure 6. As we can see, STMLOCATOR scales linearly with the size of the data, which is consistent with our algorithm analysis. As for the response time, it takes around 200 ms to return the ranking list for a bug report on the largest JDT data.

V. RELATED WORK

In this section, we briefly review the related work including textual analysis methods, semantic analysis methods, spectrum-based methods, and deep learning methods.

The key insight of textual analysis methods is to find textually similar source files for a bug report. Typically, these methods are built upon the VSM model. For example, based on the VSM model, Zhou et al. [3] propose a method to incorporate similar bug reports and their related source files for a given bug report; Saha et al. [14] further consider the code structure information such as variables and function names; later, Wang et al. [15] propose a method that combines similar bug reports, code structure, and the version history of source files. Recently, Wang et al. [16] examine a special type of bug reports, i.e., crash reports where the crash stack-trace is recorded. Ye et al. [17] propose to use skip-gram [18] to measure the similarities between bug reports and their related source files. Other examples in this class include [9], [10], [19], [20].

As to semantic analysis methods, they usually learn the latent topics/representations of bug reports and source files. For example, Lukins et al. [21] directly apply LDA on bug reports and then computed the topic distribution similarities between source files and bug reports. Nguyen et al. [22] propose a modified LDA model to detect latent topics from both bug reports and source files. Other examples in this class include [23]–[25].

Although usually treated separately, the above two types of methods are actually complementary to each other. In this work, we propose to combine them together by using topic modeling to capture semantic similarity and modeling the word co-occurrence phenomenon to capture textual similarity. Additionally, the historical fixings are largely ignored by the existing IR methods, and we use them as supervision information to further improve localization accuracy.

In addition to the above IR methods, there are other methods for locating buggy files. These methods, which are referred to as spectrum-based methods, take the program execution information instead of bug reports as input. Examples include [25]–[28]. The combination of IR-based method and spectrum-based method has also been studied [29], [30].

Recently, deep learning methods have been used to solve the bug localization problem. Lam et al. [31] apply deep neural networks on both bug reports and source files, and combine the results with IR methods. Huo et al. [8], [13] propose to use convolution neural network (CNN) to capture the structure of both bug reports and source files. Other deep learning based methods include [32]–[34]. The main limitation of these deep learning methods lies in the efficiency aspect. Moreover, although these methods make use of the historical fixings, they still follow the IR methods by using the learned representations of bug reports and source files to locate source files. Instead, we directly propose a supervised model and use it to make the predictions.

VI. CONCLUSIONS

In this paper, we have proposed STMLOCATOR for finding relevant buggy source files based on bug reports. STMLOCATOR seamlessly combines textual analysis and semantic analysis, uses historical fixings as supervised information, and

characterizes the word co-occurrence phenomenon and the long-file phenomenon. We further tailor STMLOCATOR for the bug reports with stack-traces. Experimental evaluations on three real projects show that the proposed method significantly outperforms existing methods in terms of accurately locating the relevant source files for bug reports. For future directions, it will be interesting to further improve the accuracy of bug reports with stack-traces. It will be also interesting to make use of the additional metadata such as component and platform information in the bug reports.

VII. ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2016YFB1000802), the National Natural Science Foundation of China (No. 61690204, 61672274, 61702252), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Hanghang Tong is partially supported by NSF (IIS-1651203, IIS-1715385 and IIS-1743040), and DHS (2017-ST-061-QA0001).

REFERENCES

[1] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *ESEC/FSE*. ACM, 2009, pp. 111–120.

[2] T.-D. B. Le, F. Thung, and D. Lo, "Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2237–2279, 2017.

[3] 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 *ICSE*. IEEE, 2012, pp. 14–24.

[4] 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.

[5] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning, "Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora," in *EMNLP*. Association for Computational Linguistics, 2009, pp. 248–256.

[6] 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*. The Association for Uncertainty in Artificial Intelligence Press, 2009, pp. 27–34.

[7] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling, "Fast collapsed gibbs sampling for latent dirichlet allocation," in *KDD*. ACM, 2008, pp. 569–577.

[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] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 204–214.

[10] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *The Foundations of Software Engineering*. ACM, 2014, pp. 689–699.

[11] X. Si and M. Sun, "Tag-lda for scalable real-time tag recommendation," *Journal of Computational Information Systems*, vol. 6, no. 1, pp. 23–31, 2009.

[12] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[13] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 1909–1915.

[14] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *ASE*. IEEE, 2013, pp. 345–355.

[15] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *ICPC*. ACM, 2014, pp. 53–63.

[16] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *MSR*. IEEE, 2013, pp. 247–256.

[17] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 404–415.

[18] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[19] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2015.

[20] 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.

[21] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[22] 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 *ASE*. IEEE, 2011, pp. 263–272.

[23] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *ACM SIGSOFT Software Engineering Notes*, vol. 30. ACM, 2005, pp. 286–295.

[24] 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, pp. 420–432, 2007.

[25] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug localization based on code change histories and bug reports," in *APSEC*, 2015, pp. 190–197.

[26] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*. ACM, 2005, pp. 273–282.

[27] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[28] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *ICSME*, 2014.

[29] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *FSE*. ACM, 2015, pp. 579–590.

[30] T. V.-D. Hoang, R. J. Oentaryo, T.-D. B. Le, and D. Lo, "Network-clustered multi-modal bug localization," *IEEE Transactions on Software Engineering*, 2018.

[31] 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," in *ASE*. IEEE, 2015, pp. 476–481.

[32] ——, "Bug localization with combination of deep learning and information retrieval," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 218–229.

[33] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Improving bug localization with an enhanced convolutional neural network," in *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 2017, pp. 338–347.

[34] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Machine translation-based bug localization technique for bridging lexical gap," *Information and Software Technology*, vol. 99, pp. 58–61, 2018.