Knowledge and Information Systems (2020) 62:2461–2484
https://doi.org/10.1007/s10115-019-01426-2

REGULAR PAPER

Enhancing supervised bug localization with metadata and stack-trace

Yaojing Wang[1] · Yuan Yao[1] · Hanghang Tong[2] · Xuan Huo[1] · Ming Li[1] · Feng Xu[1] · Jian Lu[1]

Received: 4 January 2019 / Revised: 19 November 2019 / Accepted: 23 November 2019 / Published online: 12 February 2020
© Springer-Verlag London Ltd., part of Springer Nature 2020

Abstract

Locating relevant source files for a given bug report is an important task in software development and maintenance. To make the locating process easier, information retrieval methods have been widely used to compute the content similarities between bug reports and source files. In addition to content similarities, various other sources of information such as the metadata and the stack-trace in the bug report can be used to enhance the localization accuracy. In this paper, we propose a supervised topic modeling approach for automatically locating the relevant source files of a bug report. In our approach, we take into account the following five 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. Fourth, metainformation brings additional guidance on the search space. Fifth, buggy source files could be already contained in the stack-trace. By integrating the above five observations, we experimentally show that the proposed method can achieve up to 67.1% improvement in terms of prediction accuracy over its best competitors and scales linearly with the size of the data.

Hanghang Tong: The work was partly done, while the author was at Arizona State University.

✉ Yaojing Wang
wyj@smail.nju.edu.cn

Yuan Yao
y.yao@nju.edu.cn

Hanghang Tong
htong@illinois.edu

Xuan Huo
huox@lamda.nju.edu.cn

Ming Li
lim@lamda.nju.edu.cn

Feng Xu
xf@nju.edu.cn

Jian Lu
lj@nju.edu.cn

[1] The State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China

[2] University of Illinois Urbana-Champaign, Champaign, USA

Keywords Bug localization · Bug reports · Supervised topic modeling · Metadata · Stack-trace

1 Introduction

Locating the relevant buggy source files for a given bug report is an important but laborsome task in software development and maintenance. Therefore, automatically locating the relevant source files is crucial, and we refer to this problem as *bug localization* in this work. In recent years, information retrieval (IR) methods have been used for the bug localization problem (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 [1]. There are basically two types of content similarities used in the 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 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. Third, various sources of information such as the metadata or the stack-traces in the bug reports are widely ignored by existing work. However, such information could be potentially helpful for locating the buggy source files [2].

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 five major components.[1] 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. Fourth, we employ the metadata of each bug report to narrow the search space of potentially relevant buggy source files. Fifth, we use the source file names in stack-traces and encode them into the model as the stack-trace usually contains the buggy source files. Finally, we conduct experimental evaluations on three real data sets, and the results demonstrate the effectiveness and efficiency of the proposed STMLOCATOR+.

In summary, the main contributions of this paper include:

- 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. Additionally, STMLOCATOR+ makes use of the metadata and stack-trace in an integral manner.

[1] This work is an extended version of our previous work [3] which considers the previous three components. Please refer to the related work section for more details.

– Experimental evaluations on three real data sets showing that the proposed STMLocator+ can achieve up to 67.1% improvement in terms of prediction accuracy over its best competitors. Moreover, each of the five components is helpful in terms of improving the prediction accuracy.

The rest of the paper is organized as follows. Section 2 provides some background knowledge and states the problem definition. Section 3 discusses the key observations, and Sect. 4 describes the proposed approach. Section 5 presents the experimental results. Section 6 covers related work, and Sect. 7 concludes the paper.

2 Related work

In this section, we briefly review the related work including IR-based methods (textual analysis and semantic analysis methods) and deep learning methods. This paper is an extension of our earlier conference version [3]. Among the five components that matter for accurate bug localization, the conference version [3] models three of them, i.e., historical fixings, word co-occurrence phenomenon, and longer-file phenomenon. In this extension, we further integrate the metadata and the stack-trace into the model.

Textual analysis methods are proposed to locate the most relevant source files by analyzing the textually similarities between bug reports and source files. Typically, these methods are built upon the VSM model. For example, based on the VSM model, Zhou et al. [4] propose a method to incorporate similar bug reports and their related source files for a given bug report; Saha et al. [5] further consider the code structure information such as variables and function names; later, Wang et al. [6] propose a method that combines similar bug reports, code structure, and the version history of source files. Recently, Wang et al. [7] examine a special type of bug reports, i.e., crash reports where the crash stack-trace is recorded. Similar studies [8,9] are also proposed to improve the bug localization by stack-trace analysis. Ye et al. [10] propose to use skip-gram [11] to measure the similarities between bug reports and their related source files. Other examples in this class include [12–17].

As to semantic analysis methods, their key insight is to learn the latent topics/representations of bug reports and source files. For example, Lukins et al. [18] directly apply LDA on bug reports and then computed the topic distribution similarities between source files and bug reports. Nguyen et al. [19] propose a modified LDA model to detect latent topics from both bug reports and source files. Kim et al. [20] propose to extract features from both bug descriptions and metadata and use naive Bayes make prediction. Zhang et al. [2] use metadata to reduce the search space of buggy source files. Other examples in this class include [21–23].

Although usually treated separately, the above two types of methods are actually complementary to each other. Our previous work [3] proposes to combine them together by using topic modeling to capture semantic similarity and modeling the word co-occurrence phenomenon to capture textual similarity. The historical fixings are largely ignored by the existing IR methods, and we use them as supervision information to further improve localization accuracy. In this work, in addition to our conference version [3], we integrate metadata into our model. Although metadata is used by some existing work, they tend to use it in a separate manner. Moreover, we also model the stack-trace information into our approach.

Recently, deep learning methods have been used to solve the bug localization problem. Lam et al. [24] apply deep neural networks on both bug reports and source files and combine the results with IR methods. Huo et al. [25,26] propose to use convolution neural network (CNN) to capture the structure of both bug reports and source files. Other deep learning-

based methods include [27–29]. 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.

Remotely related to our work, there is a line of work that aims to locate bugs based on other types of inputs. These methods typically use dynamic execution information or static program analysis to locate the bugs. Examples of dynamic methods include statistical debugging [30,31], spectrum analysis [23,32–34], change impacts in dynamic call-graphs [35,36], machine learning on dynamic properties of execution [37], etc. The combination of IR-based methods and spectrum-based methods has also been studied [38,39]. Static analysis methods are based on program slicing [40], postmortem symbolic evaluation [41], change impact analysis [42], etc.

3 Background and problem statement

In this section, we first introduce some background knowledge and then present the notations and problem definition.

3.1 Background

During software 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 [43]. For example, among the 9000 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.

A bug report in the bug tracking system typically contains the descriptions of the bug (i.e., bug description) and some metadata. An example of metadata is shown in Fig. 1. As we can see, the metadata includes status, product, component, version, hardware (platform and operating system), importance (priority and severity), etc. These types of metadata 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.

3.2 Problem statement

We use D to denote the collection of input bug reports.[2] For each bug report $d \in D$, it 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. In addition, each bug report also contains a metadata field κ_d and may contain a list of source

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

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

Bug 177679 - GoogleEarthView - SAXParseException starting embedded web server

Status: CLOSED FIXED
Alias: None
Product: z_Archived
Component: OHF (show other bugs)
Version: unspecified
Hardware: PC Windows XP
Importance: P3 normal (vote)
Target Milestone: ---
Assignee: John Thomas ECA
QA Contact: Daniel Ford ECA

Reported: 2007-03-15 18:56 EDT by John Thomas ECA
Modified: 2012-01-05 13:32 EST (History)
CC List: 1 user (show)
See Also:

Fig. 1 An example of metadata from a bug report

Table 1 Notations

Symbol	Description
M	# of bug reports
K	# of topics/source files
V	The vocabulary
$D = \{d_1, d_2, \ldots, d_M\}$	Bug report collection
$S = \{s_1, s_2, \ldots, s_K\}$	Source file collection
$d = \{w_1, w_2, \ldots, w_{N_d}\}$	A bug report d
$s = \{w_1, w_2, \ldots, w_{T_s}\}$	A source file s
Λ_d	Relevant topics/source files for d
R	Source files in stack-traces
κ	Metadata field

files R_d in stack-trace. 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 1.

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 a metadata field κ_d, a bug description field of N_d words (which may contain a list of stack-trace R_d), 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}.

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

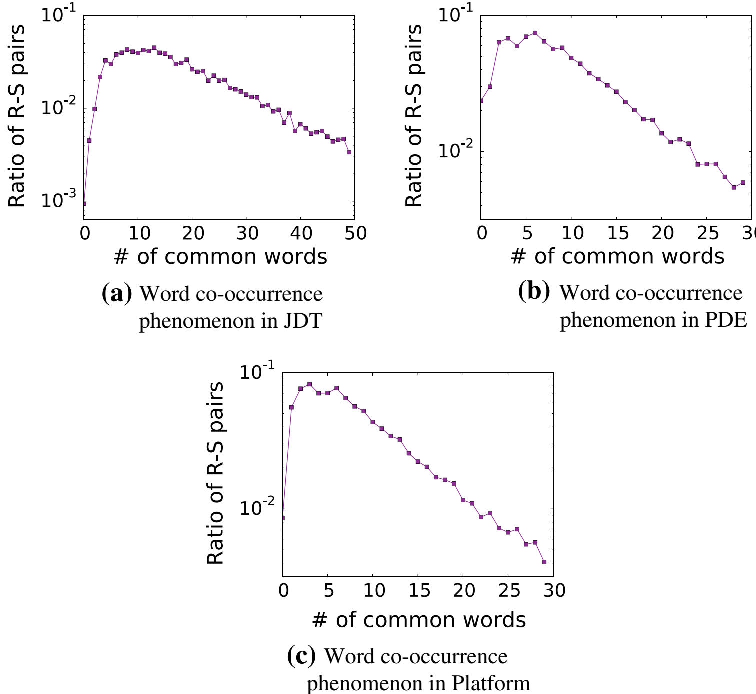


Fig. 2 Word co-occurrence phenomenon. That is, most of the words in bug reports have appeared in source files. This phenomenon widely exists in the three data sets

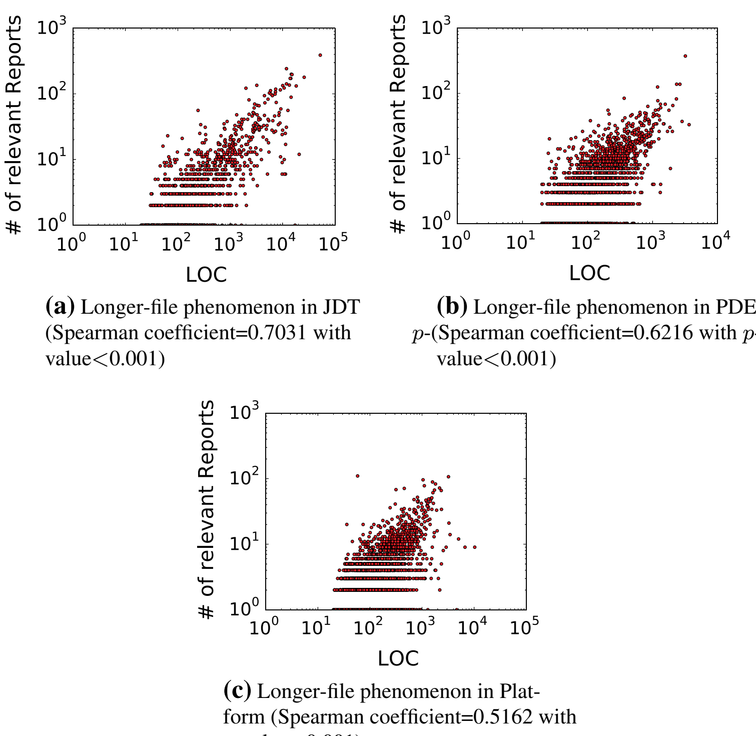
As we can see from the above problem definition, the input data are consisted of the words from bug reports and source files as well as the historical fixings (i.e., bug reports and their relevant source files). The bug localization task is to identify the most relevant source files for a new bug report.

4 Key observations

In this section, we discuss the key intuitions and observations for solving the problem defined in the previous section.

Observation 1: Supervised topic modeling Since the fixing histories between source files and bug reports cannot be ignored as well as the rich text content in both bug reports and source files, we naturally model it as a supervised topic model. Specifically, we treat each source file as a specific topic. Then, we transfer the goal of identifying the most relevant source files for a new bug report to predicting the most relevant topics for a new document. 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 or not.

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



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

To verify the word co-occurrence phenomenon, we collect data sets from three Eclipse projects (i.e., JDT, PDE, and Platform; see Sect. 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 Fig. 2a–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 contain more bugs [4]. To verify this phenomenon, we calculate the Spearman's 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 Fig. 3a–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 source files. As we can see from the figures, there is a significant positive correlation (i.e., the Spearman's 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 a prior in our model.

```
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

Observation 4: Metadata matters Once a bug report is reported, the reporter will be asked to provide metainformation in addition to the descriptions of the bug. Existing studies have shown that these metainformation especially the 'component' metadata can be helpful to locate bugs [2,20]. Therefore, in this work, we model the 'component' metadata for simplicity, while the other metadata can be similarly integrated into our model.

Observation 5: Stack-trace matters For some bug reports, the reporters can include the stack-trace of the bugs in the descriptions. An example of bug reports containing stack-traces (each stack-trace is a list of method calls that trigger an exception) is shown in Fig. 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 reports. We filter the source files from the bug reports with stack-traces, and we find that there are over 58%, 57%, and 70% bug reports containing relevant buggy source files in their stack-traces on the three data sets of PDE, Platform, and JDT, respectively.

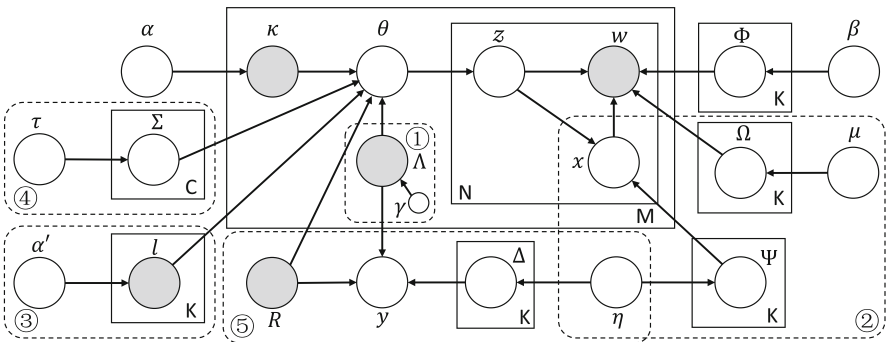
5 The proposed approach

In this section, we present the proposed STMLocator+. We start with the proposed model followed by a brief description of the learning algorithm.

5.1 The STMLocator+ model

Figure 5 shows the overall graphical representation for STMLocator+. Corresponding to the above five observations, there are five integral parts in the STMLocator+ model.

- **Supervised topic modeling** First, STMLocator+ builds upon the LDA model [44] and its generalized version LLDA [45] 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



M: # of bug reports
N: # of words
C: # of components
K: # topics/source files
w: observed word
z: topic of word w
θ: topic distribution
Λ: observed topics

l: observed file length
Φ: topic-word distribution
Ω: topic-word distribution
x: selection indicator
Ψ: selection distribution of x
y: selection indicator
κ: observed component
Δ: selection distribution of y

Σ: component-topic distribution
R: observed source files in stack-trace

① Supervised topic modeling
② Word co-occurrence phenomenon
③ Longer-file phenomenon
④ Meta-information constraints
⑤ Stack-trace tracking

Fig. 5 Graphical representation of STMLOCATOR+

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** 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.
- **Modeling metadata** For the component metadata, we introduce a component-topic distribution Σ to indicate the topic distribution in each different component. The component-topic distribution Σ is related to the historical observed fixings. By observing the specific component of each bug report, the topic distribution θ is selected from Σ.
- **Modeling stack-trace** Finally, to make use of the stack-traces, we introduce a latent variable y to indicate the probability that the source files R_d from the stack-traces can be encoded into the supervision information Λ for bug report d, where R_d is a vector of length K with $R_{d,s} \in \{0, 1\}$ indicating where the source file s in the stack-trace is a relevant buggy source file of d. The latent variable y is sampled from a Bernoulli distribution Δ. In practice, we set y as a vector of length five and select top five source files R_d from stack-traces according to their position and number of occurrence.

Although the supervised topic modeling only allows predicting the source files with bugs before, the word co-occurrence phenomenon extends the prediction to nonbuggy 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 $l \sim Dir(\alpha')$
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) Draw $Y \sim Bernoulli(\Delta)$
   (b) Draw source files R_d from stack-trace in d
   (c) For each topic $k \in [1, K]$:
      i. Draw $\Lambda_{d,k} \in \{0, 1\} \sim Bernoulli(\gamma)$
   (d) Draw Dirichlet prior $\alpha_d = (\Lambda_d \vee R_d) \circ \alpha \cdot l$
   (e) Draw component $\kappa_d \sim Dir(\alpha)$
   (f) Draw topic distribution $\theta_d \sim Dir(\alpha_d) \circ Mult(\kappa_d)$
   (g) 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

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

5.2 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, κ, R, and Λ) and unobserved variables (i.e., Z, X, and Y).

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

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

For Eq. (2), we first have

$$\begin{aligned} p(Z|\alpha L, \Lambda, R, \kappa, \tau) &= \Sigma_\kappa \cdot \prod_{m=1}^{M} p(Z_m|\alpha L, \Lambda, R) \\ &= \Sigma_\kappa \cdot \prod_{m=1}^{M} \frac{\Delta(N_m + \alpha L)}{\Delta(\alpha)}, \end{aligned} \tag{3}$$

where N_m indicates the number of words in document m, and L indicates the length of each source file in Z. Source files in Z are determined by Λ or R. The above equations can be derived by expanding the probability density expression of Dirichlet distribution and applying the standard Dirichlet multinomial integral. Σ_κ indicates the importance of each source file in Z for component κ. For each source file k, $\Sigma_{\kappa,k}$ is defined as

$$\Sigma_{\kappa,k} = \prod_{m=1}^{M} \frac{N_{\kappa,k}}{N_\kappa} \tag{4}$$

where $N_{\kappa,k}$ indicates the number of source file k appears in component κ and N_κ indicates the total number of source files appearing in component κ.

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

$$p(X|\eta, Z) = \prod_{k=1}^{K} \frac{B(N_k + \eta)}{B(\eta)}, \tag{5}$$

where N_k indicates 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 the 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 $B(\eta) = \dfrac{\Gamma(\eta_1)\Gamma(\eta_0)}{\Gamma(\eta_1 + \eta_0)}$

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

$$\begin{aligned} & p(W|Z, X, \beta, \mu) \\ &= \int p(W|Z, X, \Phi) p(\Phi|\beta)\, d\Phi \int p(W, |Z, X, \Omega) p(\Omega|\mu)\, d\Omega \\ &= \prod_{k=1}^{K} \frac{\Delta(N_{k,0} + \beta)}{\Delta(\beta)} \prod_{k=1}^{K} \frac{\Delta(N_{k,1} + \mu)}{\Delta(\mu)}, \end{aligned} \tag{6}$$

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. (6), when X is set to 1, $\prod_{k=1}^{K} \dfrac{\Delta(N_{k,0} + \beta)}{\Delta(\beta)}$ will be a constant; when X is set to 0, $\prod_{k=1}^{K} \dfrac{\Delta(N_{k,1} + \mu)}{\Delta(\mu)}$ will be a constant.

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

Similarly, for $p(Y|\eta, R, \Lambda)$, we have

$$p(Y|\eta, R, \Lambda) = \prod_{k=1}^{K} \frac{B(N_{k,R,\Lambda} + \eta)}{B(\eta)}, \tag{7}$$

where $N_{k,R,\Lambda}$ indicates the number of the source file k both appearing in R and Λ, and Y is composed of 0s or 1s to determine where a source file is generated from. As we have mentioned that $p(\Lambda|\gamma)$ and $p(R)$ are constants, Eq. (7) will be used in prediction.

Finally, putting the above equations together, we have

$$\begin{aligned} p(Z, X, W, \kappa, R, Y, \Lambda, L) \propto\ & \Sigma_\kappa \cdot \prod_{m=1}^{M} \frac{\Delta(N_m + \alpha L)}{\Delta(\alpha)} \\ & \cdot \prod_{k=1}^{K} \frac{B(N_{k,R,\Lambda} + \eta)}{B(\eta)} \cdot \prod_{k=1}^{K} \frac{B(N_k + \eta)}{B(\eta)} \\ & \cdot \prod_{k=1}^{K} \frac{\Delta(N_{k,0} + \beta)}{\Delta(\beta)} \cdot \prod_{k=1}^{K} \frac{\Delta(N_{k,1} + \mu)}{\Delta(\mu)}. \end{aligned} \tag{8}$$

The purpose of the training stage is to obtain θ, Σ, Δ, Ψ, Ω, and Φ, where θ represents the topic distribution of each document, Σ represents the distribution to indicate whether the source file in stack-trace is the correct buggy source file, 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. The parameter Σ is shown in Eq (4). Based on Eq (8), the equations for computing these parameters are listed as follows

$$\begin{aligned} \theta_{m,k} &= \Sigma_\kappa \cdot \frac{n_{m,k} + \alpha_k l_k}{\sum_{k'=1}^{K} (n_{m,k'} + \alpha_{k'})} \\ \Delta_k &= \frac{n_{k,R_k,\Lambda_k} + \eta_1}{n_{k,R_k} + \eta_1} \\ \Psi_k &= \frac{n_{k,1} + \eta_1}{\sum_{x=0}^{1} (n_{k,x} + \eta_x)}, \\ \Phi_{k,v} &= \frac{n_{k,v,0} + \beta_v}{\sum_{v'=1}^{V} (n_{k,v',0} + \beta_{v'})}, \\ \Omega_{k,v} &= \frac{n_{k,v,1} + \mu_v}{\sum_{v'=1}^{V} (n_{k,v',1} + \mu_{v'})}, \end{aligned} \tag{9}$$

where $n_{m,k}$ indicates the number of words that belong to topic k in document m, n_{k,R_k,Λ_k} indicates the number of occurrences that source file k appears in both stack-trace R_k and buggy source files Λ_k, n_{k,R_k} indicates the number of occurrences that source file k appear in stack-trace R, $n_{k,1}$ 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.

Algorithm Based on Eq. (9), 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

Algorithm 1 The Learning Algorithm for STMLOCATOR+

Input: Collection of bug reports D and source files S
Output: $\theta, \Psi, \Phi, \Omega$
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 $\theta, \Sigma, \Delta, \Psi, \Phi, \Omega$ via Eq. (4)(9);
20: **end while**
21: **return** $\theta, \Sigma, \Delta, \Psi, \Phi, \Omega$

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 (9). 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 [46] converges faster and is more stable [47]. 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 w.r.t. 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.

5.3 Prediction stage

Here we explain how to use the learned parameters to predict buggy file for a given bug report d_{new}. For a new bug report D_{new}, the topic/source file distribution is computed as follows,

$$\begin{aligned} p(t|d_{new}) &= \sum_w p(t|w) \cdot p(w|d_{new}) \\ &= \sum_w \frac{p(w|t)p(t)}{p(w)} \cdot p(w|d_{new}). \end{aligned} \tag{10}$$

First, we compute the $p(w|d_{new})$ as follows,

$$p(w|d_{new}) = \frac{count(w)}{len(d_{new})} \tag{11}$$

where $count(w)$ is the number of word w shown in document d_{new} and $len(d_{new})$ is the total number of words in document d_{new}.

Then, the probability $p(t)$ can be inferred approximately through the training set by summarizing all the $p(t|d)$ of each document d:

$$\begin{aligned} p(t) &= \sum_d p(t|d) \cdot p(d) \\ &\propto \sum_d \theta_{d,t}, \end{aligned} \tag{12}$$

here, we assume that the appearance chance of each post is the same, and thus we can ignore the $p(d)$ term in the above equation. For $p(t|d)$, we directly use the parameter θ learned by our model. In addition, source files in stack-traces will increase the chance to be chosen and we have

$$p(t) \propto (1 + R_t) y_t \circ \sum_d \theta_{d,t}, \tag{13}$$

where R_t indicates whether source file t appears in stack-traces and y_t indicates the chance if the source file t in stack-trace is the related buggy source file. y is learned by Eq. (7).

Next, since each word is generated from a certain topic t, the probability $p(w)$ is computed as,

$$p(w) = \sum_t p(w|t) \cdot p(t) \tag{14}$$

Here, Eq. (12) shows how to compute $p(t)$ and we finally need to compute $p(w|t)$ as follows,

$$\begin{aligned} p(w|t) &= \begin{cases} p(w|t, x=0)p(x=0|w \in t) \\ +p(w|t, x=1)p(x=1|w \in t), & w \in t \\ p(w|t, x=0)p(x=0|w \notin t), & w \notin t \end{cases} \\ &= \begin{cases} \Phi \cdot (1 - \Psi) + \Omega \cdot \Psi, & w \in t \\ \Phi, & w \notin t \end{cases} \end{aligned} \tag{15}$$

where we consider $p(w|t)$ in two circumstances. When the word $w \in t$, w can be generated from two topic-word distribution Ω and Φ and which should be choose is determined by distribution Ψ. Otherwise, if word $w \notin t$, then it must be generated by Φ.

6 Experimental evaluations

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

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

6.1 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 2.

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

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

Table 2 Statistics of the data sets

Data set	# Reports	# Sources	Vocabulary size
PDE	3900	2319	2964
Platform	3954	3696	3677
JDT	6267	7153	4304

For each project, we collect the bug reports (each bug report contains bug id, metadata, title, and description) from the bug tracking Web site. Then, we collect the corresponding source files from the git repository by bug id. 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.

6.2 Experimental setup

For the three data sets described above, we follow existing experiment framework [25] 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.

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 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 [4].

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 position 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 [4,12,13].

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 Intel(R) Xeon(R) E5 CPU and 64GB memory.

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

- *LLDA* [45]: 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, longer-file phenomenon, metadata information, and stack-traces.
- *tag-LDA* [48]: 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* [49]: 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* [4]: 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* [25]: 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.
- STMLocator+: STMLocator+ is the proposed method in this paper.

For the hyper-parameters of STMLocator+, we fix $\alpha = 200/K$, $\eta = 0.01$, and $\beta = \mu = 0.1$. As to the hyper-parameters of the baseline methods, we either use the same parameters as STMLocator+ (e.g., LLDA and tag-LDA) or use the default parameter settings as reported in the original papers (e.g., VSM, rVSM, and NP-CNN).

6.3 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 3, where the relative improvements compared to the best competitors are also shown in the brackets.

We can first observe from Table 3 that STMLocator+ generally outperforms all the compared methods on all the three data sets. For example, on the PDE data, STMLocator+ achieves 4.3% and 5.9% relative improvements on Hit@5 and Hit@10 over its best competitors. On Platform, STMLocator+ improves its best competitors by 6.7% and 8.1%

Table 3 Effectiveness comparisons of Hit@n, AFH@n and MRR on three data sets

Methods	LLDA	tag-LDA	VSM	rVSM	NP-CNN	STMLOCATOR+
PDE						
Hit@5	0.347	0.204	0.276	0.443	0.375	**0.462** (4.29%)
Hit@10	0.434	0.284	0.338	0.523	0.516	**0.554** (5.93%)
AFH@10	6.569	7.341	7.147	6.507	**6.469**	6.505 (−0.56%)
MRR	0.288	0.192	0.231	0.329	0.323	**0.336** (2.13%)
Platform						
Hit@5	0.422	0.364	0.332	0.612	0.489	**0.653** (6.69%)
Hit@10	0.527	0.422	0.392	0.689	0.643	**0.745** (8.12%)
AFH@10	5.781	6.473	6.224	5.593	5.478	**4.778** (12.7%)
MRR	0.335	0.301	0.283	0.443	0.401	**0.537** (21.2%)
JDT						
Hit@5	0.333	0.152	0.203	0.344	0.337	**0.408** (18.6%)
Hit@10	0.425	0.212	0.271	0.442	0.488	**0.521** (6.76%)
AFH@10	6.703	7.483	7.372	6.749	6.843	**6.078** (9.32%)
MRR	0.218	0.112	0.127	0.241	0.234	**0.391** (67.1%)

The proposed STMLOCATOR+ generally outperforms the compared methods. (For Hit@N and MMR, larger is better. For AFH@n, smaller is better. The relative improvements compared to the best competitors are also shown in the brackets.)
Bold values indicate the best results among the compared methods

w.r.t. Hit@5 and Hit@10, respectively. On JDT data set, the improvement of STMLOCATOR+ over the best competitor is 18.6% and 6.8% w.r.t. Hit@5 and Hit@10, respectively. Similarly, STMLOCATOR+ achieves the highest MRR values and smallest AFH@10 values compared to other methods in most cases as shown in Table 3. For example, STMLOCATOR+ achieves up to 67.1% improvement w.r.t. MRR over its best competitors on the MRR metric. For all the reported results above, we conduct paired t test on the average rankings, and the results show that all the positive improvements 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, source file length, metainformation, and stack-traces. This result further indicates the usefulness of modeling the 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 over-fitting.

(B) *Performance gain analysis* Next, since STMLOCATOR+ has five integral building blocks, we further study the effectiveness of these blocks. The results are shown in Table 4. Since our method is built upon the LLDA model, we also show the results for comparison. In the table, 'L+W' and 'L+S' mean the methods when the word-occurrence and the source file length are incorporated, respectively. 'L+WS' is the method when both source file length and word-occurrence are modeled.[8] When metadata is further incorporated, we denote the

[8] This is exactly the STMLocator method in the previous conference version [3].

Table 4 Performance gain analysis of different components in STMLOCATOR+

Methods	LLDA	L+W	L+S	L+WS	L+WSM	STMLOCATOR+
PDE						
Hit@5	0.347	0.404	0.384	0.457	0.459	**0.462**
Hit@10	0.434	0.483	0.457	0.543	0.541	**0.554**
AFH@10	6.569	6.827	6.562	**6.328**	6.828	6.505
MRR	0.288	0.314	0.291	**0.346**	0.331	0.336
Platform						
Hit@5	0.422	0.525	0.463	0.642	0.650	**0.653**
Hit@10	0.527	0.612	0.555	0.703	0.722	**0.745**
AFH@10	5.781	5.676	5.779	5.247	5.007	**4.778**
MRR	0.335	0.398	0.372	0.494	0.504	**0.537**
JDT						
Hit@5	0.333	0.381	0.364	0.389	0.386	**0.408**
Hit@10	0.425	0.482	0.435	0.492	0.504	**0.521**
AFH@10	6.703	6.713	6.686	6.668	6.542	**6.078**
MRR	0.218	0.274	0.223	0.298	0.357	**0.391**

All the components are useful to improve the prediction accuracy. (For Hit@N and MMR, larger is better. For AFH@n, smaller is better.)
Bold values indicate the best results among the compared methods

Table 5 Results on different design choices of length functions

Length function	Expression	MRR
Linear	$f(x) = x$	0.336
Logarithmic	$f(x) = log(x)$	0.335
Exponential	$f(x) = e^x$	0.332
Square root	$f(x) = \sqrt{x}$	0.331

method as 'L+WSM.' When the stack-trace is further incorporated, we have the proposed STMLOCATOR+ that considers all the five components.

As we can first observe from the table, L + W and L + S are generally better than LLDA, indicating the usefulness of both components. Second, L + WS further improves the performance when these two components are combined. For example, on the Hit@10 metric of the Platform data, L + WS improves L + W and L + S by 14.8% and 26.5%, respectively. Third, L + WSM improves L + WS in most cases especially on the Platform and JDT data, indicating the usefulness of metadata. Finally, STMLOCATOR+ outperforms all its sub-variants in most cases. For example, on the Hit@10 metric, STMLOCATOR+ improves L + WS and L + WSM by 5.9% and 3.1%, 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 5.
– *Co-occurrence 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 6.

Table 6 Results on different design choices of co-occurrence words

Co-occurrence words	Words	MRR
(1)	Identifiers	0.331
(2)	Annotations	0.330
(1) + (2)	Identifiers + Annotations	0.336

Table 7 Hit@10 results on the ST cases and the NST cases

Methods	PDE		Platform		JDT	
	ST	NST	ST	NST	ST	NST
LLDA	0.38	0.44	0.32	0.57	0.31	0.46
tag-LDA	0.27	0.28	0.38	0.43	0.21	0.21
VSM	0.26	0.35	0.26	0.42	0.22	0.28
rVSM	0.47	0.53	0.65	0.70	0.38	0.46
NP-CNN	0.45	0.55	0.61	0.65	0.50	0.48
L + WS	0.39	**0.57**	0.57	0.73	0.42	**0.53**
STMLocator+	**0.51**	**0.57**	**0.67**	**0.76**	**0.53**	0.51

Bold values indicate the best results among the compared methods

The results are shown in Tables 5 and 6, 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 5, 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 w.r.t. the difference choices source file length function. In our experiments, we use linear function.

In Table 6, we change the range of co-occurrence 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 7. As we can see, rVSM performs relative well in the compared methods as it is based on textual similarity. Moreover, L+WS and STMLocator+ (especially STMLocator+) perform better than the compared methods on both the NST and ST cases. This result indicates that the modeling of stack-trace of STMLocator+ helps to identify relevant buggy source files from the stack-traces.

6.4 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 Fig. 6. As we

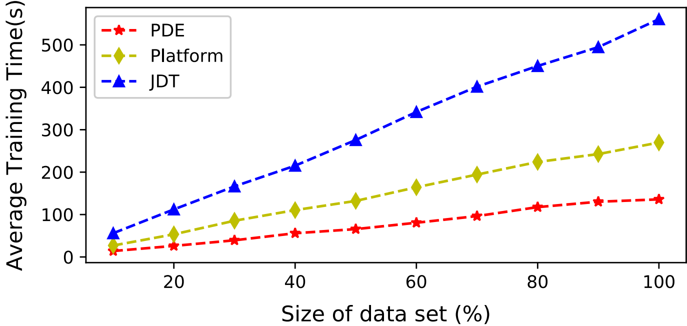


Fig. 6 Scalability of STMLOCATOR+. It scales linearly w.r.t. the data size

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 800 ms to return the ranking list for a bug report on the largest JDT data.

7 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, characterizes the word co-occurrence phenomenon and the long-file phenomenon, and models metadata and stack-trace information. 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 typical type of bug report such as crash bugs. It will be also interesting to make use of the more metainformation in addition to the component metadata in the bug reports.

Acknowledgements This work is supported by the National Natural Science Foundation of China (Nos. 61690204, 61672274, 61702252) and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Hanghang Tong is partially supported by NSF (1651203, 1715385, and 1939725).

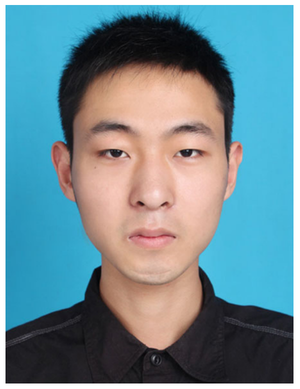
References

1. Le T-DB, Thung F, Lo D (2017) Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. Empir Softw Eng 22(4):2237–2279
2. Zhang X, Yao Y, Wang Y, Xu F, Lu J (2017) Exploring metadata in bug reports for bug localization. In: Asia-Pacific software engineering conference (APSEC), 2017 24th. IEEE, pp 328–337
3. Wang Y, Yao Y, Hanghang T, Huo X, Li M, Xu F, Lu J (2018) Bug localization via supervised topic modeling. In: ICDM
4. Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: ICSE. IEEE, pp 14–24
5. Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: ASE. IEEE, pp 345–355
6. Wang S, Lo D (2014) Version history, similar report, and structure: putting them together for improved bug localization. In: ICPC. ACM, pp 53–63
7. Wang S, Khomh F, Zou Y (2013) Improving bug localization using correlations in crash reports. In: MSR. IEEE, pp 247–256

8. Moreno L, Treadway JJ, Marcus A, Shen W (2014) On the use of stack traces to improve text retrieval-based bug localization. In: 2014 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 151–160
9. Wong C-P, Xiong Y, Zhang H, Hao D, Zhang L, Mei H (2014) Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 181–190
10. Ye X, Shen H, Ma X, Bunescu R, Liu C (2016) From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th international conference on software engineering. ACM, pp 404–415
11. Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp 3111–3119
12. Wu R, Zhang H, Cheung S.-C, Kim S (2014) Crashlocator: locating crashing faults based on crash stacks. In: Proceedings of the 2014 international symposium on software testing and analysis. ACM, pp 204–214
13. Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: The foundations of software engineering. ACM, pp 689–699
14. Xia X, Lo D, Shihab E, Wang X, Zhou B (2015) Automatic, high accuracy prediction of reopened bugs. Autom Softw Eng 22(1):75–109
15. Ashok B, Joy J, Liang H, Rajamani SK, Srinivasa G, Vangala V (2009) Debugadvisor: a recommender system for debugging. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM, pp 373–382
16. Shepherd D, Fry ZP, Hill E, Pollock L, Vijay-Shanker K (2007) Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the 6th international conference on Aspect-oriented software development. ACM, pp 212–224
17. Saha RK, Lawall J, Khurshid S, Perry DE (2014) On the effectiveness of information retrieval based bug localization for c programs. In: 2014 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 161–170
18. Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using latent Dirichlet allocation. Inf Softw Technol 52(9):972–990
19. Nguyen AT, Nguyen TT, Al-Kofahi J, Nguyen HV, Nguyen TN (2011) A topic-based approach for narrowing the search space of buggy files from a bug report. In: ASE. IEEE, pp 263–272
20. Kim D, Tao Y, Kim S, Zeller A (2013) Where should we fix this bug? A two-phase recommendation model. IEEE Trans Softw Eng 39(11):1597–1610
21. Liu C, Yan X, Fei L, Han J, Midkiff SP (2005) Sober: statistical model-based bug localization. In: ACM SIGSOFT Software Engineering Notes, vol 30. ACM, pp 286–295
22. Poshyvanyk D, Gueheneuc Y-G, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans Softw Eng 33(6):420–432
23. Youm KC, Ahn J, Kim J, Lee E (2015) Bug localization based on code change histories and bug reports. In: APSEC, pp 190–197
24. Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2015) Combining deep learning with information retrieval to localize buggy files for bug reports. In: ASE. IEEE, pp 476–481
25. Huo X, Li M, Zhou Z-H (2016) Learning unified features from natural and programming languages for locating buggy source code. In: IJCAI, pp 1606–1612
26. Huo X, Li M (2017) 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, pp 1909–1915
27. Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2017) Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on program comprehension (ICPC). IEEE, pp 218–229
28. Xiao Y, Keung J, Mi Q, Bennin KE (2017) Improving bug localization with an enhanced convolutional neural network. In: 2017 24th Asia-Pacific software engineering conference (APSEC). IEEE, pp 338–347
29. Xiao Y, Keung J, Bennin KE, Mi Q (2018) Machine translation-based bug localization technique for bridging lexical gap. Inf Softw Technol 99:58–61
30. Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI (2005) Scalable statistical bug isolation. ACM SIGPLAN Not 40(6):15–26
31. Liu C, Fei L, Yan X, Han J, Midkiff SP (2006) Statistical debugging: a hypothesis testing-based approach. IEEE Trans Softw Eng 32(10):831–848
32. Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE. ACM, pp 273–282

33. Abreu R, Zoeteweij P, Van Gemund AJ (2007) On the accuracy of spectrum-based fault localization. In: TAICPART-MUTATION. IEEE 2007, pp 89–98
34. Xuan J, Monperrus M (2014) Learning to combine multiple ranking metrics for fault localization. In: ICSME
35. Ren X, Shah F, Tip F, Ryder BG, Chesley O (2004) Chianti: a tool for change impact analysis of java programs. In: ACM Sigplan Notices, vol. 39, no. 10. ACM, pp 432–448
36. Chesley OC, Ren X, Ryder BG, Tip F (2007) Crisp—a fault localization tool for java programs. In: 29th international conference on software engineering, 2007 (ICSE 2007). IEEE, pp 775–779
37. Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: Proceedings of 26th international conference on software engineering, 2004 (ICSE 2004). IEEE, pp 480–490
38. Le T-DB, Oentaryo RJ, Lo D (2015) Information retrieval and spectrum based bug localization: better together. In: FSE. ACM, pp 579–590
39. Hoang TV-D, Oentaryo RJ, Le T-DB, Lo D (2018) Network-clustered multi-modal bug localization. IEEE Trans Softw Eng 45(10):1002–1023
40. Weiser M (1982) Programmers use slices when debugging. Commun ACM 25(7):446–452
41. Manevich R, Sridharan M, Adams S, Das M, Yang Z (2004) Pse: explaining program failures via post-mortem static analysis. In: ACM SIGSOFT software engineering notes, vol 29, no. 6. ACM, pp 63–72
42. Acharya M, Robinson B (2011) Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd international conference on software engineering. ACM, pp 746–755
43. Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: ESEC/FSE. ACM, pp 111–120
44. Blei DM, Ng AY, Jordan MI (2003) Latent Dirichlet allocation. J Mach Learn Res 3:993–1022
45. Ramage D, Hall D, Nallapati R, Manning CD (2009) Labeled lda: a supervised topic model for credit attribution in multi-labeled corpora. In: EMNLP. Association for Computational Linguistics, pp 248–256
46. Asuncion A, Welling M, Smyth P, Teh YW (2009) 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, pp 27–34
47. Porteous I, Newman D, Ihler A, Asuncion A, Smyth P, Welling M (2008) Fast collapsed Gibbs sampling for latent Dirichlet allocation. In: KDD. ACM, pp 569–577
48. Si X, Sun M (2009) Tag-lda for scalable real-time tag recommendation. J Comput Inf Syst 6(1):23–31
49. Salton G, Wong A, Yang C-S (1975) A vector space model for automatic indexing. Commun ACM 18(11):613–620

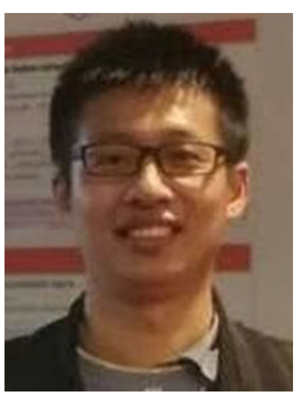
Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Yaojing Wang is a Ph.D. student in the Department of Computer Science and Technology, Nanjing University, China. His research interests include software repository mining and network embedding. He received the B.Sc. degree from Nanjing University in 2014.

Yuan Yao is currently an assistant researcher in the Department of Computer Science and Technology, Nanjing University, China. He received the Ph.D. degree in computer science from Nanjing University in 2015. His research interests include social media mining, software repository mining, and software.

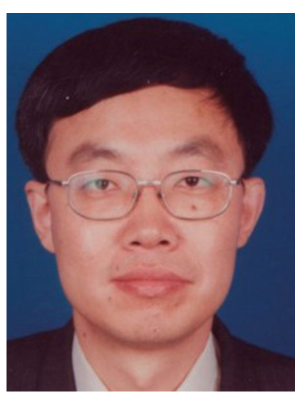
Hanghang Tong is currently an associate professor at Department of Computer Science at University of Illinois at Urbana-Champaign. Before that he was an associate professor at School of Computing, Informatics, and Decision Systems Engineering (CIDSE), Arizona State University. He received his M.Sc. and Ph.D. degrees from Carnegie Mellon University in 2008 and 2009, both in Machine Learning. His research interest is in large-scale data mining for graphs and multimedia. He has received several awards, including SDM/IBM Early Career Data Mining Research award (2018), NSF CAREER award (2017), ICDM 10-Year Highest Impact Paper award (2015), four best paper awards (TUP'14, CIKM'12, SDM'08, ICDM'06), seven 'bests of conference,' 1 best demo, honorable mention (SIGMOD'17), and 1 best demo candidate, second place (CIKM'17). He has published over 100 refereed articles. He is the Editor-in-Chief of SIGKDD Explorations (ACM), an action editor of Data Mining and Knowledge Discovery (Springer), and an associate editor of Knowledge and Information Systems (Springer) and Neurocomputing Journal (Elsevier), and has served as a program committee member in multiple data mining, database, and artificial intelligence venues (e.g., SIGKDD, SIGMOD, AAAI, WWW, CIKM, etc.).



Xuan Huo received his B.Sc. degree from Nanjing University, China, in June 2013. Since September 2013, he become a Ph.D. candidate in the LAMDA Group led by professor Zhi-Hua Zhou, under the supervision of Prof. Ming Li.

Ming Li is a Professor at LAMDA led by Professor Zhi-Hua Zhou, my Ph.D. supervisor. He received my B.Sc. and Ph.D. degrees in computer science from Department of Computer Science and Technology, Nanjing University, China, in 2003 and 2008, respectively. He joined Department of Computer Science and Technology of Nanjing University in 2008. His major research interests include machine learning and data mining, especially on software mining. Major research interests include machine learning and data mining, especially on software mining. He has served as the area chair of IEEE ICDM, senior PC member of the premium conferences in artificial intelligence such as IJCAI and AAAI, and PC members for other premium conferences such as KDD, NIPS, ICML, etc., and the chair of the International Workshop on Software Mining. He has served as the associate editor (junior) for Frontiers of Computer Science and editorial board member for International Journal of Data Warehousing and Mining. He is the executive board member of ACM SIGKDD China Chapter. He has been granted various awards including the Excellent Youth Award from NSFC, the New Century Excellent Talents program of the Education Ministry of China, the CCF Distinguished Doctoral Dissertation Award, and Microsoft Fellowship Award, etc.

Feng Xu received the B.S. and M.S. degrees from Hohai University in 1997 and 2000, respectively. He received the Ph.D. degree from Nanjing University in 2003. He is a professor in the Department of Computer Science and Technology at Nanjing University. His research interests include trust management, trusted computing, and software reliability.



Jian Lu Jian Lu received the B.S. and Ph.D. degrees in Computer Science from Nanjing University, China, in 1982 and 1988, respectively. He is currently a professor in the Department of Computer Science and Technology and the Director of the State Key Laboratory for Novel Software Technology at Nanjing University. He serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. His research interests include software methodologies, software automation, software agents, and middleware systems.