A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report

Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
{anhnt,tung,jafar,hungnv,tien}@iastate.edu

***Abstract*—Locating buggy code is a time-consuming task in software development. Given a new bug report, developers must search through a large number of files in a project to locate buggy code. We propose BugScout, an automated approach to help developers reduce such efforts by narrowing the search space of buggy files when they are assigned to address a bug report. BugScout assumes that the textual contents of a bug report and that of its corresponding source code share some technical aspects of the system which can be used for locating buggy source files given a new bug report. We develop a specialized topic model that represents those technical aspects as topics in the textual contents of bug reports and source files, and correlates bug reports and corresponding buggy files via their shared topics. Our evaluation shows that BugScout can recommend buggy files correctly up to 45% of the cases with a recommended ranked list of 10 files.**

***Index Terms*—Defect Localization, Topic Modeling.**

I. Introduction

To ensure software integrity and quality, developers always spend a large amount of time on debugging and fixing software defects. A software defect, which is informally called a *bug*, is found and often reported in a bug report. A bug report is a document that is submitted by a developer, tester, or end-user of a system. It describes the defect(s) under reporting. Such documents generally describe the situations in which the software does not behave as it is expected, i.e. fails to follow the technical requirements of the system. Being assigned to fix a bug report, a developer will analyze the bug(s), search through the program's code to locate the potential defective/buggy files. Let us call this process *bug file localization*.

This process is crucial for the later bug fixing process. However, in a large system, this process could be overwhelming due to the large number of its source files. At the same time, a developer has to leverage much information from the descriptive contents of the bug report itself, from his domain knowledge of the system and source code, from the connections between such textual descriptions in a report and different modules in the system, and from the knowledge on prior resolved bugs in the past, etc. Therefore, to help developers target their efforts on the right files and raise their effectiveness and efficiency in finding and fixing bugs, an automated tool is desirable to help developers to narrow the search space of buggy files for a given bug report.

In this paper, we propose *BugScout*, a topic-based approach to locate the candidates of buggy files for a given bug report. The key ideas of our approach are as follows:

1) there are several technical functionality/aspects in a software system. Some functionality/aspects might be buggy, i.e. incorrectly implemented. As a consequence, a bug report is filed. The textual contents of the bug report and those of the corresponding buggy source files (comments and identifiers) tend to describe some common technical aspects/topics (among other different technical topics). Thus, if we could identify the technical topics that are described in the bug reports and source files, we could recommend the files that describe the common technical topics with a given bug report.

2) Some source files in the system might be more buggy than the others (e.g. they are more defect-prone) [9].

3) Similar bugs might be related to similar fixed files, thus, if a given bug report x has some similar topic(s) with a previously resolved bug report y in the history, the fixed files associated with y could be the candidate buggy files for x.

In this paper, we extend Latent Dirichlet Allocation (LDA) [4] to model the relation among a bug report and its corresponding buggy source files. LDA is a generative, machine learning model that is used to model the topics in a collection of textual documents. In LDA, a document is considered to be generated by a "machine" which is driven via parameters by the hidden factors called *topics*, and its words are taken from some vocabulary [4]. One can train the model with historical data to derive its parameters. Terms in the documents in the project's history are the observed data. LDA considers that all documents are generated by that "machine" with its parameters. When LDA is applied to a new document, it uses its process to "generate" that document, thus, it can tell the topics that are described in the document's contents and the corresponding terms for those topics.

In BugScout, the technical aspects in the system including in bug reports and source code are modeled by topics. BugScout model has two components, representing two sets of documents: source files and bug reports. The *S-component* for a source file is a LDA model in which a source file is modeled as a document influenced by the topic distribution parameter and other parameters of the LDA model. For some buggy source files, some of their technical topics might be incorrectly implemented. As a consequence, a bug report is filed to report on the buggy topic(s). The second component, *B-component*, is designed to model bug reports. B-component is an extended LDA model in which a bug report is modeled as a document that are influenced not only by its own topic distribution

parameter, but also by the topic distribution parameters of the buggy source files corresponding to that bug report.

The rationale behind this design is that the contents of a bug report are written by the tester/reporter and describe about the occurrence of the bug(s). Thus, the technical topics of the buggy files must be mentioned in the bug report. At the same time, a bug report is also a relatively independent document and can discuss about other topics. For example, a bug report on memory leaking could also mention about the related topics on file loading when the memory leaking was observed. The S-component models the source files from the developers' point of view, while the B-component models the bug reports written from the point of view of the reporters. Two components are connected to form BugScout.

We also developed the algorithms for training and predicting buggy files for a given bug report. Parameters in BugScout are the parameters combined from two components. They can be derived by our training algorithm with the historical data of the previous bug reports and corresponding buggy files. When a new bug report is filed, BugScout is applied to find its topics. Then, the topics of that report are compared with the topics of all source files. The source files that have had more defects in the past and have shared topics with the new bug report will be ranked higher and recommended to developers.

We have conducted an empirical evaluation of BugScout on several large-scale, real-world systems. BugScout can recommend candidate buggy files correctly up to 33% of the cases with one single file, and up to 45% of the cases with a ranked list of 10 files. That is, in almost half of the cases, the top 10 files in the recommended ranked list contain the actual buggy file(s). The key contributions of this paper include

1. BugScout, a topic model that accurately recommends a short list of candidate buggy files for a given bug report,

2. Associated algorithms for model training and predicting of buggy files for a new bug report; and

3. An empirical evaluation on the usefulness of BugScout.

Section 2 presents the motivating examples. Section 3 provides the details of our model BugScout. Section 4 describes the associated algorithms for training and inferring candidate buggy files. Section 5 is for our empirical evaluation. Section 6 discusses the related work and conclusions appear last.

II. Motivating Examples

Let us discuss some real-world examples that motivate our approach. We collected the bug reports and their corresponding fixed files from an industrial project of a large corporation. The 3-year development data from that project includes source files, documentation, test cases, defects and bug reports, change data, and communication data among developers. In that project, for a *work item*, a general notion of a development task, the data contains a summary, a description, a tag, and relevant software artifacts. There are 47,563 work items, of which 6,246 are marked as bug reports. As a developer fixed a bug in response to a bug report, (s)he was required to record the fixing changes made to the fixed files. We wrote a simple tool to extract the data and observed the following examples.

Bug Report #50900
Summary: Error saving state returned from update of external object; incoming sync will not be triggered.
Description: This showed up in the server log. It's not clear which interop component this belongs to so I just picked one of them. Seems like the code run in this scheduled task should be able to properly handle a stale data by refetching/retrying.

Fig. 1: Bug report #50900

InteropService.java
```
// Implementation of the Interop service interface.
// Fetch the latest state of the proxy
// Fetch the latest state of the sync rule
// Only return data from last synchronized state
// If there is a project area associated with the sync rule,
// Get an advisable operation for the incoming sync
// Schedule sync of an item with the state of an external object
// The result of incoming synchronization (of one item state).
// Use sync rule to convert an item state to new external state.
// Get any process area associated with a linked target item....
// For permissions checking, get any process area associated with the
target item. If none, get the process area of the sync rule.
// return an instance of the process server service...
// Do incoming synchronization of one state of an item.
public IExternalProxy processIncoming (IExternalProxyHandle ...) {...
...}
```

Fig. 2: Source file InteropService.java

Example 1. Figures 1 and 2 display bug report #50900 and the corresponding fixed/buggy file InteropService.java (for brevity, only comments are shown in Figure 2). The report is about a software defect in which incoming synchronization tasks were not triggered properly in a server. We found that the developers fixed the bug at a single file InteropService.java by adding code into two methods processIncoming and processIncomingOneState to handle a stale data by refetching. As shown, both bug report #50900 and the buggy file InteropService.java describe the same problematic functionality of the system: the "synchronization" of "incoming data" in the interop service. This faulty technical aspect (was described and) could be recognized via the relevant terms, such as sync, synchronization, incoming, interop, state, schedule, etc. Considering the bug report and the source file as textual documents, we could consider this *technical aspect* as one of their *topics*. This example suggests that the bug report and the corresponding buggy source files *share* the common buggy technical topic(s). Thus, detecting the common technical topic(s) between a bug report and a source file could help in bug file localization.

Example 2. We also found another report #45208 (Figure 3) that was also fixed at the single file InteropService.java, but at two different methods processOutgoing and processOutgoingOneState. It reported a different technical topic: the permission issue with background outgoing tasks in interop service. Examining those two methods, we saw that the bug report and the source file also share that common topic, which is expressed via the relevant terms such as outgoing, synchronize, permissions, process, state, interop, service, etc.

This example also shows that a source file, e.g. InteropService.java, could have multiple buggy technical aspects, and thus, could be traced/linked from multiple bug reports.

Bug Report #45208
Summary: Do not require synchronize permissions for background outgoing sync task.
Description: The background outgoing sync task in Interop component, which runs as ADMIN, is currently invoking a process-enabled operation to save external state returned from some external connection. It causes a problem because ADMIN needs to be granted process permissions. A periodic server task should be "trusted", however, so it shouldn't invoke process-enabled operations.

Fig. 3: Bug Report #45208

Bug Report #40994
Summary: Mutiple CQ records are being created
Description: There are 5 records in CQ db which seem to have identical information. They all have the same headline - CQ Connector Use Case for RTC Instance with Multiple Project Areas. On the client side there is only 1 item, 40415, corresponding to all these.

Fig. 4: Bug Report #40994

Example 3. We also examined bug report #40994 (Figure 4). Analyzing it carefully, we found that it reported on *three* technical aspects including an issue with the interop service connection, an issue with the connection to CQ database, and an issue with the instance of RTC framework. For this bug report, developers made several fixing changes to nine different files including InteropService.java. This example shows that a bug report could also describe multiple technical aspects and could be linked/mapped to multiple files. Moreover, despite having multiple topics, this bug report and the corresponding fixed file InteropService.java share the common buggy topic on "interop service connection". That common buggy topic was described in parts of the report and in parts of InteropService.java.

Observations and Implications. The motivating examples give us the following observations:

1. A system has several technical aspects with respect to multiple functionality. Some aspects/functionality might be incorrectly implemented.

2. A software artifact, such as a bug report or a source file, might contain one or multiple technical aspects. Those technical aspects can be viewed as the topics of those documents. Each topic is expressed via a collection of relevant terms.

3. A bug report and the corresponding buggy source files often share some common buggy technical topics.

4. Some source files in the system might be more buggy than the others.

Those observations suggest that, while finding the source files relevant to a bug report, developers could explore 1) the similarity/sharing of topics between them; and 2) the bug profile of the source files. That is, if a source file shares some common topic(s) with a bug report, and is known to be buggy in the history, it is likely to be relevant to the reported bug(s).

III. Model

In BugScout, each software system is considered to have K technical aspects/topics. Among other types of artifacts in a system, BugScout concerns two types of artifacts: source files and bug reports. Source file is a kind of software artifacts written in a programming language. Each source file implements one or multiple technical aspects of a software system. Some of them might be incorrectly implemented and cause bugs. A bug report is a kind of software artifacts that describe buggy technical aspect(s). Our model has two components for those two types of artifacts: S-component for source files and B-component for bug reports. The S-component models the source files from the developers' point of view, while the B-component models the bug reports written from the point of view of bug reporters. Two components are connected to form BugScout. Let us describe them in details.

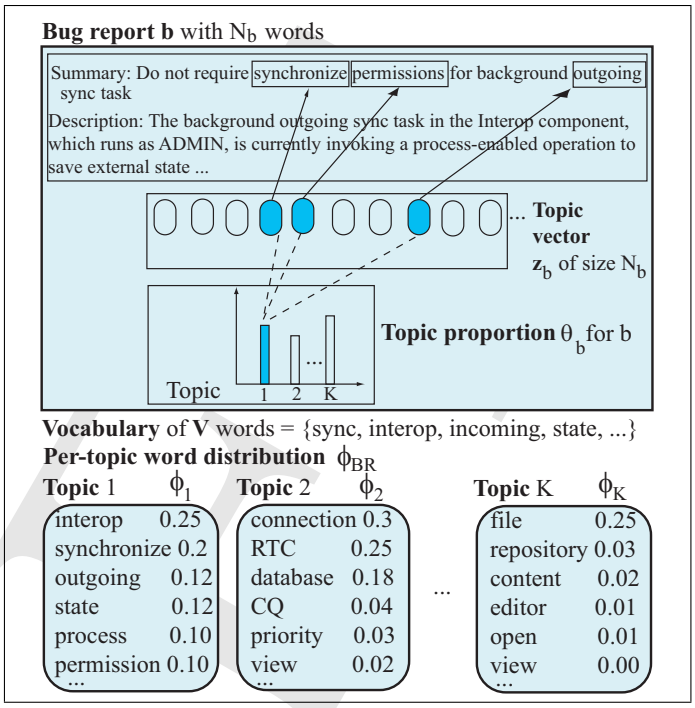

Fig. 5: Illustration of LDA [4]

A. S-Component

S-component in BugScout is adopted from LDA [4]. In general, source code always includes program elements and are written in some specific programming language. In BugScout, a source file is considered as a *text document* s. Texts from the comments and identifiers in a source file are extracted to form the *words* of the document s.

Topic vector. A source document s has N_s **words**. In S-component, each of the N_s positions in document s is considered to describe one specific technical topic. Therefore, for each source document s, we have a **topic vector** z_s with the length of N_s in which each element of the vector is an index to one topic (i.e. 1-K).

Topic Proportion. Each position in s describes one topic, thus, the entire source document s can describe multiple topics. To represent the existence and importance of multiple topics in a document s, LDA introduces the **topic proportion** θ_s. θ_s for each document s is represented by a vector with K elements. Each element corresponds to a topic. The value of each

element of that vector is a number in [0-1], which represents the proportion of the corresponding topic in s. The higher the value $\theta_s[k]$ is, the more important topic k contributes to the document s. For example, in the file InteropService.java, if $\theta_s = [0.4, 0.4, 0.1, ...]$, 40% of words are about outgoing sync, other 40% are about incoming sync, etc.

Vocabulary and Word Selection. Each position in source code document s is about one topic. However, to describe that topic, one might use different words which are drawn from a vocabulary of all the words in the project (and other regular words in any dictionary of a natural language). Let us call the combined **vocabulary** Voc with the size of V. Each word in Voc has a different usage frequency for describing a topic k, and a topic can be described by one or multiple words. LDA uses a **word-selection vector** ϕ_k for the topic k. That vector has the size of V in which each element represents the usage frequency of the corresponding word at that element's position in Voc to describe the topic k. Each element v in ϕ_k can have a value from 0 to 1. For example, for a topic k, $\phi_k = [0.3, 0.2, 0.4, ...]$. That is, in 30% of the cases the first word in Voc is used to describe the topic k, 20% of the cases the second word is used to describe k, and so on. For a software system, each topic k has its own vector ϕ_k then K topics can be represented by a $K \times V$ matrix ϕ_{src}, which is called **per-topic word distribution**. Note that ϕ_{src} is applicable for all source files, rather than for s individually.

LDA is a machine learning model and from its generative point of view, a source file s in the system is considered as an "instance" generated by a "machine" with three aforementioned variables z_s, θ_s, ϕ_{src}. Given a source code document s of size N_s, based on topic proportion θ_s of the document, the machine generates the vector z_s describing the topic of every position in the document s. For each position, it then generates a word w_s based on the topic assigned to that position and the per-topic word distribution ϕ_{src} corresponding to that topic. This is called a **generative process**. The terms in the source files in the project's history are the observed data. One can train the LDA model with historical data to derive those three parameters to fit the best with the observed data. As a new document s' comes, LDA uses the learned parameters to derive the topics of the document and the proportion of those topics.

B. B-Component

Let us describe the B-component in our BugScout model, which is extended from LDA [4]. As a consequence of an incorrect implementation of some technical aspects in the system, a bug report is filed. Thus, a bug report describes the buggy technical topic(s) in a system. Similar to S-component, B-component also considers each bug report b as a document with three variables z_b, θ_b, ϕ_{BR} (Figure 5). A bug report b has N_b words. The topic at each position in b is described by a topic vector z_b. The selection for the word at each position is modeled by the per-topic word distribution ϕ_{BR}. Note that ϕ_{BR} applies to all bug reports and it is different from ϕ_{src}.

The bug report b has its own topic proportion θ_b. However, that report is influenced not only by its own topic distribution, but also by the topic distribution parameters of the buggy source files corresponding to that bug report. The rationale behind this design is that in addition to its own topics, the contents of a bug report must also describe about the occurrence of the bug(s). That is, the technical topics of the corresponding buggy files must be mentioned in the bug report. At the same time, a bug report might describe about other relevant technical aspects in the system from the point of view of the bug reporter.

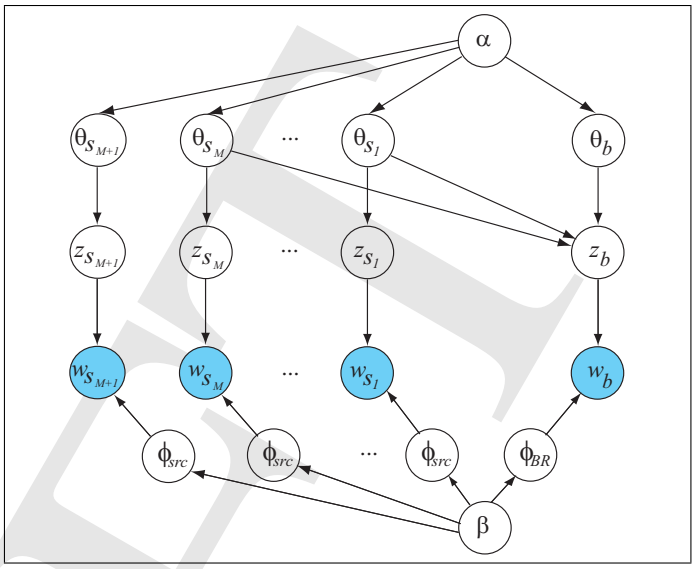

Fig. 6: BugScout Model

Let us use $s_1, s_2, ..., s_M$ to denote the (buggy) source files that are relevant to a bug report b. The topic distribution of b is a combination of its own topic distribution θ_b (from the writing view of a bug reporter) and topic distributions of $s_1, s_2, ..., s_M$. In BugScout, we have $\theta_b^* = \theta_{s1}.\theta_{s2}.....\theta_{sM}.\theta_b$. The equation represents the sharing of buggy topics in a bug report and corresponding source files. If a topic k has a high proportion in all θ_s and θ_b (i.e. k is a shared buggy topic), it also has a high proportion in θ_b^*. The generative process in B-component is similar to S-component except that it takes into account the combined topic proportion $\theta_b^* = \theta_{s1}.\theta_{s2}.....\theta_{sM}.\theta_b$.

C. BugScout Model

To model the relation between a bug report and corresponding buggy source files, we combine the S-component and B-component into BugScout (Figure 6). For a bug report b, in the B-component side, there are 3 variables that control b: z_b, θ_b, and ϕ_{BR}. However, if the source files $s_1, s_2, ..., s_M$ are determined to cause a bug reported in bug report b, the topic vector z_b will be influenced by the topic distributions of those source files. That is, there are links from $\theta_{s_1}, \theta_{s_2}, ...\theta_{s_M}$ to z_b. For each source document, there are 3 variables that control s: z_s, θ_s, and ϕ_{src} (Figure 6). There are two hyper parameters α and β whose conditional distributions are assumed as in LDA. α is the parameter of the uniform Dirichlet prior on topic distributions θ_s and θ_b. β is the parameter of the uniform Dirichlet prior on the per-topic word distributions ϕ_{src} and ϕ_{BR}.

For training, the model will be trained from historical data including source files, bug reports and the links between bug reports and corresponding fixed source files. The variables of BugScout will be trained to derive its parameters and to make the model fit most with both the document data and the links between bug reports and corresponding buggy source files.

For predicting, the model will be applied to a new bug report b_{new}. BugScout uses its trained parameters to "generate" that bug report and estimate its topic proportion $\theta_{b_{new}}$. That topic proportion will be used to find corresponding source files that share most topics. Cosine distance is used to determine the topic proportion similarity. We use $sim(s, b)$ to denote the topic proportion similarity between a source file s and a bug report b. The topics of that bug report are compared with the topics of all source files. Finally, the files that have shared the buggy topics with the new bug report will be ranked and recommended to the developers.

Because BugScout has two components and the dependencies among variables in the internal model become much different from LDA, we developed our own algorithms for training BugScout with historical data and predicting for a new bug report. We will present them in Section IV.

Integrating with Defect-Proneness of Source Files: In a software system, some files might be more buggy than the others. We integrate this characteristic into BugScout to improve its accuracy in buggy file prediction. We use the following equation to formulate the idea:

$$P(s|b) = P(s) * sim(s, b)$$

In the equation, $P(s|b)$ is the total relevance measure of a source file to a given bug report b. $sim(s, b)$ is the similarity of the topics of the source file and those of the bug report. $P(s)$ is the bug profile of source file s. In BugScout's current implementation, $P(s)$ is determined by the number of bugs in the file s in the history and by its size. Other strategies for computing defect-proness of a source file can be used for $P(s)$.

The equation implies the inclusion of both defect-proneness and the buggy topics of a source file. Given a new bug report, if a source file is determined as having higher buggy potential, and it also contains shared buggy topics with the bug report, it will be ranked higher in the list of possible buggy files. Next section will describe our training and predicting algorithms.

IV. Algorithms

A. Training Algorithm

The goal of this algorithm is to estimate BugScout's parameters given the training data from a software system. The collection of source files S, that of bug reports B, and the set of links $L_s(b)$ between a bug report and corresponding source file(s) will be used to train BugScout and estimate its parameters (z_s, θ_s, ϕ_{src}), and (z_b, θ_b, ϕ_{BR}).

Algorithm Overview. Our algorithm is based on Gibbs sampling method [7]. The idea of Gibbs sampling is to estimate the parameters based on the distribution calculated from other sampled values. The estimation is made iteratively between the values until the estimated parameters reach their convergent state (i.e. the new estimated value of a parameter do not change in comparison with its previous estimated value).

```
1 // ---------- Training ----------
2 function TrainModel(SourceFiles S, BugReports B, Links L_s(b))
3   z_S, z_B, φ_src, φ_BR ← random();
4   repeat
5     z'_S ← z_S, z'_B ← z_B
6     // Update the variables for source documents
7     for (SourceFile s ∈ S)
8       for (i = 1 to N_s)
9         z_s[i] = EstimateZS(s, i)        //estimate topic assignment at position i
10      end
11      θ_s[k] = N_s[k]/N_s                //estimate topic distribution
12    end
13    φ_src,k[w_i] = N_k[w_i]/N            //estimate per−topic word distribution
14    // Update the variables for bug reports
15    for (BugReports b ∈ B)
16      for (i = 1 to N_b)
17        z_b = EstimateZB1(w_b, L_s(b), i)
18      end
19      θ_b[k] = N_b[k]/N_b
20    end
21    φ_BR,k[w_i] = N_k[w_i]/N
22  until (|z − z'| <= ε)
23  return z_S, z_B, θ_S, θ_B, φ_src, φ_BR
24 end
25 // ---------- Estimate topic assignment for s --------
26 function EstimateZS(SourceFile w_s, int i)
27   for (k = 1 to K)
28     p(z_s[i] = k) ← ((n_s[−i,k]+α)/(n_s−1+Kα)) ((n_src,k[−i,w_i]+β)/(n_src,k−1+Vβ))
29   end
30   z_s[i] ← sample(p(z_s[i]))
31 end
32 // ---------- Estimate topic assignment for b --------
33 function EstimateZB1(BugReport w_b, int i, Links L_ws(w_b))
34   for (k = 1 to K)
35     p(z_b[i] = k) ← ((∏_{s∈L_s(b)} n_s[k] + α)^{(n_b[−i,k])} / (∏_{s∈L_s(b)} n_s + Kα)^{((n_b−1))}) ((n_BR,k[−i,w_i]+β)/(n_BR,k−1+Vβ))
36   end
37   z_b[i] ← sample(p(z_b[i]))
38 end
```

Fig. 7: Model Training Algorithm

Figure 7 shows the pseudo-code of our training algorithm. Function TrainModel() is used to train BugScout by using the collections of source files (S), bug reports (B) and the set of links $L_s(b)$ between the bug reports and the corresponding buggy source files. Line 3 describes the initial step where the parameters z_s, z_b, ϕ_{src}, ϕ_{BR} are assigned with randomly values. Lines 4-22 describe the iterative steps in estimating the parameters using Gibbs sampling. The iterative process terminates when the values of parameters are convergent. The convergent condition is determined by checking whether the difference between the current estimated values and previous estimated ones is smaller than a threshold. In our implementation, the process is stopped after a number of iterations, which is large enough to ensure a small error. In each iteration, the parameters are estimated for all source code documents s in S (lines 7-13) and all bug reports b in B (lines 15-21).

Detailed Description. Let us explain in details all the steps.

Step 1: *Estimating the topic assignment for source documents in S (lines 7-10).* With each document s in S, BugScout estimates the topic assignment $z_s[i]$ for position i (line 9).

Function EstimateZS (lines 26-31) provides the detailed computation. For each topic k in K topics, BugScout estimates the probability that topic k will be assigned for position i in document s. Then, it samples a topic based on the probabilities of ks (line 30). The equation follows the topic assignment estimation by Gibbs sampling in LDA [7]:

$$p(z_i = k|z_s[-i], w_s) = \frac{(n_s[-i,k] + \alpha)}{(n_s - 1 + K\alpha)} \frac{(n_{src,k}[-i,w_i] + \beta)}{(n_{src,k} - 1 + V\beta)} \quad (1)$$

where $n_s[-i,k]$ is the number of words in s (except for the current position i) that are assigned to topic k; n_s is the total number of words in s; $n_{src,k}[-i,w_i]$ is the number of words w_i in all source documents S (except for the current position) that are assigned to topic k; and $n_{src,k}$ is the number of all words in S that are assigned to topic k.

The intuition behind this equation is that, given a word $w_s[i]$ at position i of document s, the probability a topic k that is assigned to that position can be estimated based on both the proportion of the terms in s (excluding the current one) that describe topic k (i.e. $\frac{(n_s[-i,k])}{(n_s-1)}$) and the probability that the current term $w_s[i]$ appears if topic k is assigned (i.e. $\frac{(n_{src,k}[-i,w_i])}{(n_{src,k}-1)}$). Moreover, the current position value can be estimated by prior knowledge of surrounding positions.

Step 2: *Estimating topic proportion θ_s for a source file (line 11).* Line 11 shows the estimation for the topic proportion of source file s. Once topic assignments for all positions in s are estimated, the topic proportion $\theta_s[k]$ of topic k in that document can be approximated by simply calculating the ratio between the number of words describing the topic k and the length of the document.

Step 3: *Estimating word distribution ϕ_{src} (line 13).* Line 13 shows the estimation for the per-topic word distribution for each word w_i from Voc (size V) and topic k. $\phi_{src,k}$ is a vector of size V representing how often each word in vocabulary Voc can be used to describe topic k in the source file collection S. Element at index i in ϕ_k determines how often the word with index i in Voc can be used to describe k. Thus, $\phi_k[w_i]$ can be approximated by the ratio between the number of times that the word index i in Voc is used to describe topic k and the total number of times that any word that is used to describe k.

Step 4: *Estimating the topic assignment for bug reports in B (lines 16-18).* For each bug report b in B, BugScout estimates the topic assignment $z_b[i]$ for position i (line 17). Function EstimateZB1() (lines 33-38) provides the detail. For each topic k in K, BugScout estimates the probability that topic k will be assigned for position i. It then samples a topic based on the probabilities of ks (line 37). The estimate equation is similar to that for a source file document:

$$p(z_b[i] = k|z_b[-i], w_b) = \frac{(n_b^*[-i,k] + \alpha)}{(n_b^*[-i] + K\alpha)} \frac{(n_{BR,k}[-i,w_i] + \beta)}{(n_{BR,k} - 1 + V\beta)} \quad (2)$$

where $n_{BR,k}[-i,w_i]$ is the number of words w_i in all bug reports in B, except the current position, that are assigned to topic k, and $n_{BR,k}$ is the number of words in S describing k.

The crucial difference between (2) and (1) is that because a bug report describes the buggy topic(s) in the corresponding source documents, the proportion θ^* of a topic k described in the bug report includes its own topic proportion θ_b and the topic proportions of corresponding source files $\theta_{s_1}, \theta_{s_2}, ..., \theta_{s_M}$, where $s_1, s_2, ..., s_M \in L_s(b)$ (i.e. the set of buggy source files linking to bug report b). That leads to $n_b^*[-i,k] = n_b[-i,k] \prod_{s \in L_s(b)} n_s[k]$ and $n_b^*[-i] = (n_b - 1) \prod_{s \in L_s(b)} n_s$, in which $n_b[-i,k]$ is the number of words in b (except for the current position i) that are assigned to topic k. n_b is the total number of words in b. For each buggy source document s linked to b, $n_s[k]$ is the number of words in s (except for the current position i) that are assigned to topic k. n_s is the total number of words in s.

Step 5: *Estimating topic proportion θ_b for a bug report b and estimate word distribution ϕ_{BR} (line 19 and line 21).* Those estimation steps are similar to the steps for θ_s and ϕ_{src}.

B. Predicting and Recommending Algorithm

The goal of this algorithm is to estimate the topic proportion of a newly arrived bug report b_{new} and derive a candidate list of potential buggy source files that cause the reported bug(s). The algorithm uses the trained model from the previous algorithm to estimate the topic proportion of b_{new}, then it uses a similarity measure to compute the topic similarity between b_{new} and each source file s in S. The similarity, in combination with P(s), will be used to estimate how likely s can cause the bug reported in b. The output of the algorithm will be a list of potential buggy source files corresponding to the given bug report. Our algorithm is also based on Gibbs sampling.

Figure 8 describes the steps of our algorithm. Lines 4-10 show the estimation step for parameters $z_{b_{new}}$ and $\theta_{b_{new}}$ for new bug report b_{new} (we do not need to recalculate ϕ_{BR} because they are fixed after the training phase). Because we do not know the buggy links between source files and b_{new}, we use LDA Gibbs sampling formula to estimate topic assignment and topic proportion for b_{new}. The function for estimating $z_{b_{new}}$ is described in EstimateZB2 (lines 18-23). In the equation, $n_{b_{new}}[-i,k]$ is the number of words in b_{new} (except the current position i) that are assigned to topic k. $n_{b_{new}}$ is the total number of words in b_{new}. $n_{BR,k}[-i,w_i]$ is the number of words w_i in all source files S (except the current position) that are assigned to topic k. $n_{BR,k}$ is the number of all words in S that are assigned to topic k. BugScout calculates $\delta(s, b_{new})$, i.e. the probability that source file s causes the bug reported in b_{new} (lines 12-14). $\delta(s, b_{new})$ is calculated by multiplying the buggy profile $p(s)$ of s and the topic similarity measure sim(...) between s and b_{new} (lines 24-28). Finally, it returns a ranked list of potential buggy files corresponding to b_{new}.

V. Evaluation

This section describes our empirical evaluation on buggy files recommendation accuracy of BugScout for given bug reports in comparison with the state-of-the-art approaches. All

```
// ---------- Predict and return relevant list ----------
function Predict(z_S, z_B, θ_S, θ_B, φ_src, φ_BR, BugReport b_new, Prior P(s))
  // Estimate topic proportion of new bug report b_new
  repeat
    z'_{b_new} ← z_{b_new}
    for (i = 1 to N_b)
      z_{b_new} = EstimateZB2(b_new, i) //estimate topic assignment at position i
    end
    θ_{b_new}[k] = N_{b_new}[k]/N_{bnew}   //estimate topic proportion
  until (|z_{b_new} − z'_{b_new}| <= ε)
  // Calculate relevance of source files to a bug report
  for (SourceFile s ∈ S)
    δ(s, b_new) ← P(s) * sim(s, b_new) //calculate prob of s causing the bug
  end
  return rankedList(δ(s, b_new))
end
// ---------- Estimate topic assignment for b ---------
function EstimateZB2(BugReport b_new,, int i)
  for (k = 1 to K)
    p(z_{b_new}[i] = k) ← ((n_{b_new}[−i,k]+α)/(n_{b_new}−1+Kα)) ((n_{BR,k}[−i,w_i]+β)/(n_{BR,k}−1+Vβ))
  end
  z_{b_new}[i] ← sample(p(z_{b_new}[i]))
end
// −−Calculate topic similarity between a source file and a bug report −
function sim(SourceFile s,BugReport b_new)
  σ ← Σ_{k=1..K} θ_s[k]θ_{b_new}[k]      //calculate dot product
  Sim ← 1/(1+exp(−σ))
end
```

Fig. 8: Predicting and Recommending Algorithm

experiments were carried out on a computer with CPU AMD Phenom II X4 965 3.0 GHz, 8GB RAM, and Windows 7.

A. Data Sets

We collected several datasets in different software projects including Jazz (a development framework from IBM), Eclipse (an integrated development environment), AspectJ (a compiler for aspect-oriented programming), and ArgoUML (a graphical editor for UML). Eclipse, ArgoUML, and AspectJ datasets are publicly available [24], and have been used as the benchmarks in prior bug file localization research [19], [24]. All projects are developed in Java with a long history.

Each data set contains three parts. The first part is the *set of bug reports*. Each bug report has a summary, a description, comments, and other meta-data such as the levels of severity and priority, the reporter, the creation date, the platform and version. The second part is the *source code files*. We collected all source files including the buggy versions and the fixed files for all fixed bug reports. The third part is *the mapping* from bug reports to the corresponding fixed files. For Jazz project, the developers were required to record the fixed files for bug reports. For other projects, the mappings were mined from both version archives and bug databases according to the method in [24]. Generally, the change logs were mined to detect special terms signifying the fixing changes. Details are in [24]. Table I shows the information on all subject systems.

B. Feature Extraction

Our first step was to extract the features from bug reports and source files for our model. For the bug reports/files, grammatical words and stopwords were removed to reduce noises, and other words were stemmed for normalization as in previous work [19], [13]. Tf-Idf was then run to determine and remove the common words that appear in most of the bug reports. The remaining words in the bug reports were collected into a common vocabulary Voc. A word was indexed by its position in the vocabulary.

TABLE I
Subject Systems

System	Jazz	Eclipse	AspectJ	ArgoUML
# mapped bug reports	6,246	4,136	271	1,764
# source code files	16,071	10,635	978	2,216
# words in corpus	53,820	45,387	7,234	16,762

Only fixed bug reports were considered because those reports have the information on corresponding fixed source files. We used the summary and description in a bug report as a bug report document in BugScout. For a fixed source document, we used the comments, names, and identifiers. Identifiers were split into words, which were then stemmed. Next, a feature vector was extracted from each document. A vector has the form $W_i = (w_{i0}, w_{i1}, \ldots, w_{iN})$, where w_{ik} is an index of the word at position k in Voc, and N is the length of the source or bug report document. The vectors were used for training and predicting. For prediction, BugScout outputs a ranked list of relevant files to a given bug report.

C. Evaluation Metrics and Setup

To measure the prediction performance of BugScout, we use the *top rank* evaluation approach. Our prediction tool provides a ranked list of 1-20 (n) potential fix files for each bug report in a test set. n could be seen as the number of candidate files to which developers should pay attention. The prediction accuracy is measured by the intersection set of the predicted and the actually fixed files. We consider a *hit* in prediction, if BugScout predicts at least one correct fixed/buggy file in the ranked list. If one correct buggy file is detected, a developer can start from that file and search for other related buggy files. Prediction accuracy is measured by the ratio of the number of hits over the total number of prediction cases in a test set. Accuracy was reported for all top-rank levels n.

In our experiment, we used the longitudinal setup as in [19] to increase the internal validity and to compare with prior results. The longitudinal setup allows data in the past history to be used for training to predict for the more recent bug reports.

First, all bug reports in a subject system were sorted according to their filing dates, and then distributed into ten equally sized sets called *folds*: fold 1 is the oldest and fold 10 is the newest in the chronological order. BugScout was executed several times in which older folds were used for training and the last fold was used for prediction. Specifically, at the first run, fold 1 was used for training to predict the result for fold 2 (fold 1 was not used for prediction because there is no prior data). For each bug report in fold 2, we measured the accuracy result for that report by comparing the predicted fixing files with the actual fixed files. An average accuracy was recorded for fold 2. We continued for fold 3 using both folds 1 and 2 as the training set. We repeated until

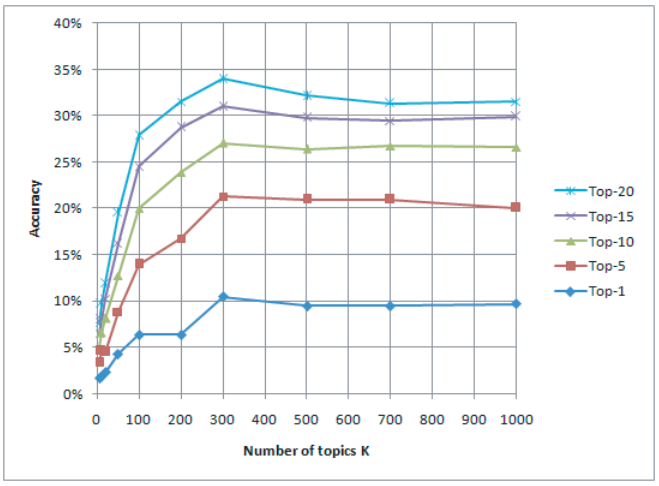

Fig. 9: Accuracy and the Number of Topics without P(s)

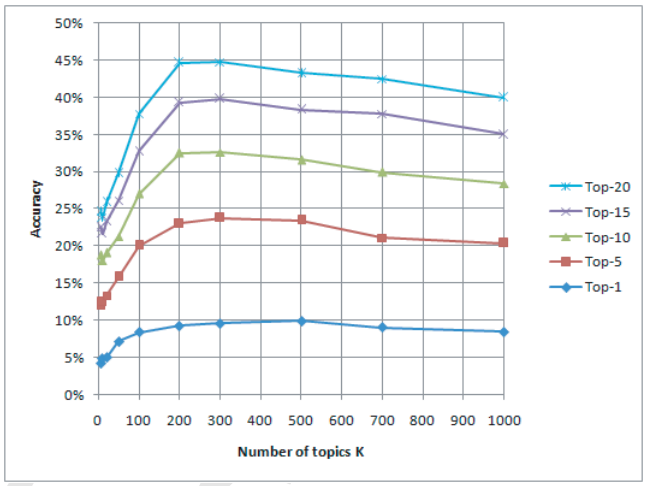

Fig. 10: Accuracy and the Number of Topics with P(s)

fold 10 using all first nine folds as the training set. For each top-rank level n=1-20, we also measured the average accuracy across all nine test sets from folds 2-10. By using this setup, we could have a realistic simulation of real-world usage of our tool in helping bug fixing as a new bug report comes. If data is randomly selected into folds, there might be the cases where some newer data would be used for training to predict the buggy files corresponding to the older bug reports.

D. Parameter Sensitivity Analysis

Our first experiment was to evaluate BugScout's accuracy with respect to the number of chosen topics K. We chose ArgoUML for this experiment. Two hyper-parameters α and β were set to 0.01. We compared the results when the defect-proneness information of source files $P(s)$ was used and was not used (Section III). We varied the values of K: if K is from 1-100, the step is 10 and if K is from 100-1,000, the step is 100. The accuracy values were measured for each top-rank level n=1-20. Figure 9 shows the top-1 to top-20 accuracy results. As shown, for this dataset in ArgoUML, the accuracy achieves its highest point in the range of around 300 topics. That is, this particular data set might actually contain around that number of topics. As K is small (< 50), accuracy was low because there are many documents classified into the same topic group even though they contain other technical topics. When K is around 300, the accuracy reaches its peak. That is because those topics still reflect well those reports and files. However, as K is large (>500), then the nuanced topics may appear and topics may begin to overlap semantically with each other. It causes one document having many topics with similar proportions. This overfitting problem degrades accuracy. This phenomenon is consistent for all top-rank levels.

We repeated the same experiment, however, in this case, we used BugScout with the defect-proneness information $P(s)$ of the files, i.e. the number of bugs of the files in the past history and the sizes of the files (Section III). Figure 10 shows the result. As seen, with this information about the source files, at K = 300, BugScout can improve from 3-11% for top-5 to top-20 accuracy. Importantly, for this dataset, accuracy is generally very good. With top-5 accuracy of 24%, when BugScout recommends a ranked list of 5 files, one in four cases, that list contains a correct buggy file for the bug report. With the ranked list of 10 files, the accuracy is about 33%, that is, one of three cases, a buggy file for the bug report is actually in that recommended list. This result also shows that BugScout can potentially be combined with other defect-proness prediction algorithms [15], [17], [21] to improve accuracy.

E. Accuracy Comparison

Our next experiment was to evaluate BugScout's accuracy in comparison with that of the state-of-the-art approaches: the Support Vector Machine (SVM)-based approach by Premraj *et al.* [19] and the approach by Lukins *et al.* [12] that combines LDA and Vector Space Model (VSM). For the former approach, we re-implemented their approach by using the same machine learning tool LIBSVM [5] as in their work. For the latter one, we re-implemented their LDA+VSM approach with our own code. For our tool, we performed the tuning process to pick the right number of topics as described earlier.

Figure 11 shows the accuracy result on Jazz dataset. The X-axis shows the size n of the top-ranked list. As seen, BugScout outperforms both SVM and LDA+VSM. For top-1 accuracy, it achieved about 34%: when BugScout recommended one single file for each bug report in a test set, it correctly predicted the buggy file 34% on average. That is, in one of three cases, the single recommended file was actually the buggy file for the given bug report. The corresponding top-1 accuracy levels for SVM and LDA+VSM are only 25% and 7%, respectively. Thus, in top-1 accuracy, BugScout outperformed those two approaches by 9% and 27%, respectively. With the ranked list of 5 files, the top-5 accuracy is around 40%. That is, in four out of ten cases, BugScout was able to recommend at least one correct buggy file among its 5 recommended files. The corresponding numbers for SVM and LDA+VSM are only 31% and 18%. At top-10 accuracy, BugScout also outperformed the other two approaches by 7% and 16%, respectively.

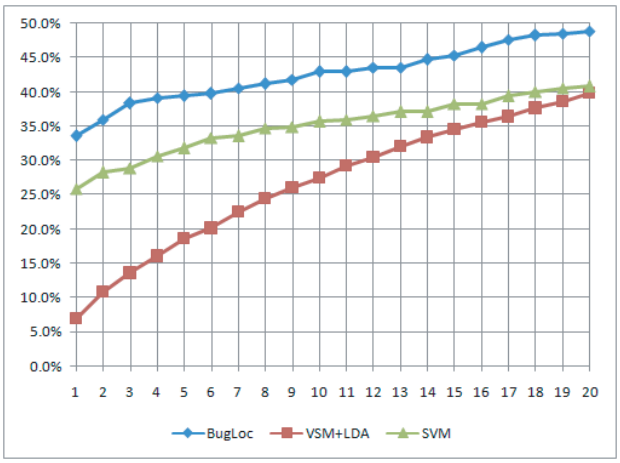

Fig. 11: Accuracy Comparison on Jazz dataset

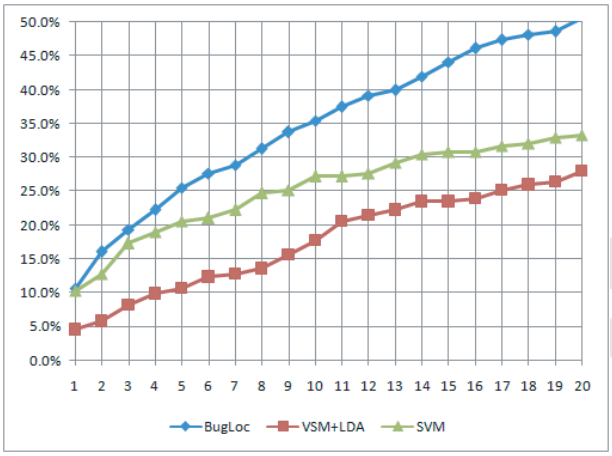

Fig. 12: Accuracy Comparison on AspectJ dataset

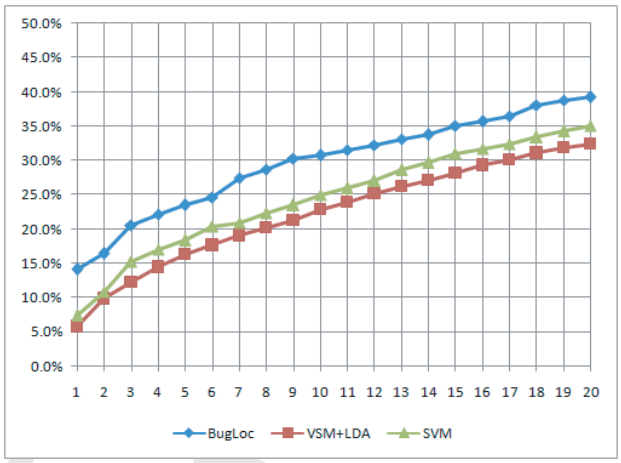

Fig. 13: Accuracy Comparison on Eclipse dataset

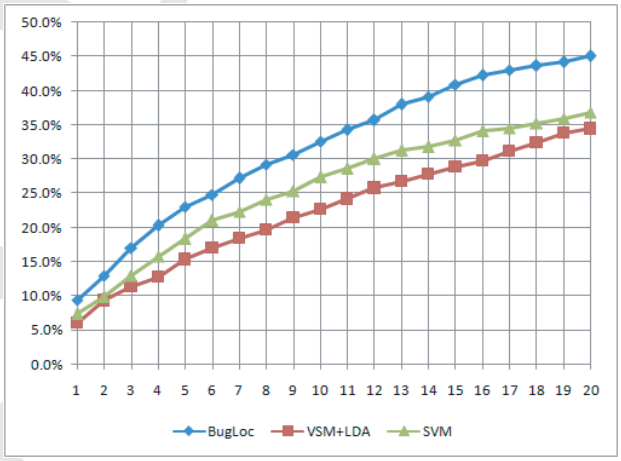

Fig. 14: Accuracy Comparison on ArgoUML dataset

Interesting examples. Examining those results, we found that BugScout could detect within the top-10 ranked list all buggy files of bug reports in Section II and several similar cases. BugScout also correctly detected *the buggy files that have never been defective in the past*. For example, for bug report #47,611 in Jazz, BugScout correctly detected with its single recommendation the buggy file com.ibm.team.scm.service.internal.IScmDataMediator, which was not in the training set (i.e. not found buggy before).

Figure 13 shows the comparison result on Eclipse dataset. Figure 12 and Figure 14 display the comparison results on AspectJ and ArgoUML datasets, respectively. As seen, BugScout consistently achieved higher accuracy from 8-20% than the other two approaches for top-1 to top-5 ranked lists. For top-10 accuracy, the corresponding number is from 5-19%.

Time Efficiency. Table II displays running time of our tool. Both average training time and prediction time for one bug report is reasonably fast: 0.3s-1.3s and 0.8s-25s, respectively. Generally, BugScout is scalable for systems with large numbers of bug reports, thus, is well-suited for daily practical use.

Threats to Validity. Our experiment was only on 4 systems. We also re-implemented the existing approaches since their tools are not available. However, we used the same library as used in their tools for our re-implementation.

TABLE II
Time Efficiency

System	Jazz	Eclipse	AspectJ	ArgoUML
Average Training Time per BR (s)	1.31	1.16	0.32	0.97
Average Prediction Time per BR (s)	25	20.1	0.79	11.6

VI. Related Work

A related work to BugScout is from Lukins *et al.* [12]. They directly applied LDA analysis on bug reports and files to localize the buggy files. They perform *indexing on all source files* with the detected topics from LDA. Then, for a new bug report, a textual query is formed from its description and a *search* via Vector Space Model (VSM) is performed among such indexed source files. In contrast, BugScout correlates the topics in both source files and bug reports, and uses topics as a random variable in our model. Moreover, their approach does not work well if the code contains few common terms with a new bug report. As shown in Section V, BugScout outperformed their approach of LDA+VSM.

TRASE [1] combines LDA with *prospective traceability*, i.e. capturing developers' activities during development, for tracing between architecture-based documents and source code. Instead of directly using LDA, BugScout correlates bug reports and buggy files via shared topics. Prospective tracing links are also incorporated in BugScout via the links from bug reports and corresponding fixed files recorded during bug fixing.

Bugtalks from Premraj *et al.* [19] addresses the bug localization using bug reports' information. They combine a machine learning approach using Support Vector Machine (SVM) on textual features in documents with a usual suspect list (i.e. the list of frequently buggy locations) with the philosophy that bugs tend to concentrate in selected code components. To train the SVM model, bug reports are paired with their fixed files in the usual suspect list to form positive examples. However, their approach faces unbalanced data with a huge number of negative examples, which are the incorrect pairs of bug reports and files. Thus, their accuracy depends much on the randomly selected set of such negative examples. Moreover, their approach assumes that a bug report contains similar terms as the identifiers in the fixed files. BugScout does not need negative examples and it correlates the reports and fixed files via common topics. Evaluation results also show that BugScout achieves higher top-5 accuracy from 5-12%.

Ostrand *et al.* [18] and Bell *et al.* [3] developed negative binomial regression models to predict the expected number of faults in each file of the next release. Despite using information from modification requests (MRs) (release IDs, abstract, category), their model is mainly based on the code, bugs, and the modification histories of the files.

In **software traceability** and **concept/feature location** research, several Information Retrieval (IR) approaches have been proposed to trace the relations of high-level concepts in code. The followings and their combined approaches are popular: formal concept analysis [11], Latent Semantic Indexing [13], probabilistic topic models and LDA [10], [1], name-based model [2], and a combination of IR and execution traces [11]. Comparing to IR approaches, BugScout is able to learn the correlation of the topics in two different types of documents: bug reports and corresponding buggy code.

Our approach complements well to **bug prediction** approaches [17], [23], [9]. Some relies on *code churns* and *code changes* [15], [22], [17], [21]. They focus on code properties and changes, rather than on textual information in bug reports as in BugScout. They can provide excellent a-priori information on defect-proneness of source files for BugScout. Moser *et al.* [15] built machine learners with logistic regression, Naïve Bayes, and decision trees with *metrics on code changes*. Nagappan *et al.* [17]'s model uses the metrics based on *change bursts*. Canfora and Cerulo [6] store textual descriptions of fixed change requests, use them to index the source files for searching from a new change request. Other prediction approaches rely on *code and change complexity* metrics [16], [14], [8]. Others also show that files depending on buggy modules are likely to be error-prone [23]. BugCache [9] maintains a cache of locations that are likely to have faults.

VII. Conclusions

We propose BugScout, an automated approach to localize the buggy files given a bug report. It assumes that the textual contents of a bug report and those of its corresponding source code share some technical aspects of the system. We develop a specialized topic model, that represents the technical aspects in the textual contents of bug reports and source files as topics, and correlates bug reports and buggy files via the shared topics. Empirical results show that BugScout is accurate in localizing buggy files and outperforms existing approaches. We plan to explore convergence measures for Gibb sampling as in [20].

Acknowledgments

This project is funded in part by NSF CCF-1018600 grant. The first and fourth authors were funded in part by Vietnamese Education Foundation (VEF) fellowships.

References

[1] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *ICSE '10*, pages 95–104. ACM Press, 2010.
[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *ICSE '10*, pages 375–384. ACM Press, 2010.
[3] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Looking for bugs in all the right places. In *ISSTA '06*, pages 61–72. ACM Press, 2006.
[4] D. Blei, A.Y. Ng, and M. Jordan. Latent Dirichlet Allocation. J. Mach. Learn. Res. 3 (Mar. 2003), 993-1022.
[5] C. Chang and C. Lin, *LIBSVM: library for support vector machines*.
[6] G. Canfora and L. Cerulo. How software repositories can help in resolving a new change request. In *Workshop on Empirical Studies in Reverse Engineering*, 2005.
[7] T. Griffiths, Gibbs sampling in the generative model of Latent Dirichlet Allocation. Technical Report, 2002
[8] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE '09*, pages 78–88. IEEE CS, 2009.
[9] S. Kim, T. Zimmermann, J. Whitehead, Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498. IEEE CS, 2007.
[10] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *ASE'07*, ACM.
[11] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE '07*, pp. 234–243. ACM Press, 2007.
[12] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *J. of Inf. Softw. Technol.*, 52(9):972–990, 2010.
[13] A. Marcus and J. I. Maletic. Recovering documentation to source code traceability links using Latent Semantic Indexing. In *ICSE'03*, IEEE.
[14] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE TSE*, 33(1):2–13, 2007.
[15] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE '08*, pages 181–190. ACM Press, 2008.
[16] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE '06*, pages 452–461. ACM Press, 2006.
[17] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *ISSRE'10*, pp. 309-318. IEEE CS.
[18] T. J. Ostrand, E. J. Weyuker, R. Bell. Predicting the location and number of faults in large software systems. *IEEE TSE*, 31(4):340–355, 2005.
[19] R. Premraj, I.-X. Chen, H. Jaygarl, T.N. Nguyen, T. Zimmermann, S. Kim, and A. Zeller. Where should I fix this bug? (bugtalks.wikispaces.com), 2008.
[20] A. E. Raftery and S. Lewis. How Many Iterations in the Gibbs Sampler. In *Bayesian Statistics 4*. Oxford University Press, pp. 763-773, 1992.
[21] J. Ratzinger, M. Pinzger, and H. Gall. EQ-mine: Predicting short-term defects for software evolution. In *FASE'07*, pp. 12–26. Springer-Verlag.
[22] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: raising risk awareness. In *ESEC/FSE-13*, pages 107–110. ACM Press, 2005.
[23] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE'08*, pp. 531–540. ACM, 2008.
[24] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE'07*, pages 433-436. ACM Press, 2007.