Intelligent bug handling systems

Akshay Kumar Chaluvadi
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
achaluv@ncsu.edu

Pritesh Ranjan
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
pranjan@ncsu.edu

Raghavendra Prasad Potluri
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
rpotlur@ncsu.edu

ABSTRACT

Bug tracking systems play an important role in software maintenance. They allow both developers and users to submit problem reports on observed failures. There are intelligent ways in which the problem of bug tracking for various specific platforms is being handled and automated. There are var ious complications and features that are to be addressed by research, including ***a) identification of duplicate bug reports which is an important research problem in the mining software repositories field, b) auto-completion of bug reports for specific software platforms (like android) & c) narrowing the search space of buggy files from a bug report.*** Detecting duplicate bug reports helps reduce triaging efforts and save time for developers in fixing the same issues. Among several automated detection approaches, text-based information retrieval (IR) approaches have been shown to outperform others in term of both accuracy and time efficiency. Topic modeling has also helped identify duplicate bugs and has hence helped in triaging bugs, it has also helped solve the problem of software traceability. We also discuss the age-old problem of automated labeling of multinomial topic models. This essay assesses how bug tracking systems today are evolving and dealing with triaging bugs, dealing with duplicates using machine learning, auto-completing bug reports for modern software such as Android mobile and tablet platforms. We also discuss some core techniques that are at the heart of this research.

Keywords

Statistical topic model; Latent Dirichlet Allocation; Defect localization; Statistical semantic language model

1. INTRODUCTION

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.

Bug fixing is vital in producing high-quality software products. Bug fixing happens in both development and post release time. In either case, the developers, testers, or end users run a system and find its incorrect behaviors that do not conform to their expectation and the system's requirements. Then, they report such occurrences in a bug report, which are recorded in an issue-tracking database. Generally, there are many users interacting with a system and reporting its issues. Thus, a bug is occasionally reported by more than one reporters, resulting in duplicate bug reports. Detecting whether a new bug report is a duplicate one is crucial. It helps reduce the maintenance efforts from developers. Moreover, duplicate reports provide more information in the bug fixing process for that bug.

Tracking bug reports is one of the best practices in the maintenance of computer software. Bug reports are submitted by both developers and users of the system. Allowing users to submit bug reports provides developers with continuous feedback about the operational behavior of their product. However, submitted reports vary in quality, and it is often difficult to understand what is being reported. Bug tracking systems today are evolving quickly by using machine learning algorithms and text analytics. Manual triaging and duplicate bug detection is both challenging and time consuming.

To automate the detection of duplicate bug reports, several approaches have been introduced. Early approaches have applied information retrieval (IR) to this problem with Vector Space Model (VSM) in which a bug report is modeled as a vector of textual features computed via Term Frequency Inverse Document Frequency (Tf-Idf) term weighting measurement [1, 2]. To improve the detection accuracy, natural language processing (NLP) has been combined with those IR methods [2]. Execution trace information on the reported bugs in the bug reports is also used in combination with NLP [3]. However, execution traces might not be available in all bug reports. Another predominant approach to this problem is machine learning (ML). Jalbert and Weimer [4] and by Sun et al. [5] used Machine Learning methods but the key limitation of ML approaches is their low efficiency.

One of the papers [6] we went through introduces DBTM, a duplicate bug report detection model that takes advantage of not only IR-based features but also topic-based features from our novel topic model, which is designed to address textual dissimilarity between duplicate reports. In DBTM, a bug report is considered as a textual document describing one or more technical issues/topics in a system. Duplicate bug reports describe the same technical issue(s) even though the issue(s) is reported in different terms. In their topic model, they extended Latent Dirichlet Allocation (LDA) [6] to represent the topic structure for a bug report as well as the duplication relations among them. Two duplicate bug reports must describe about the shared technical issue(s)/topic(s) in addition to their own topics on different phenomena. The topic selection of a bug report is affected not only by the topic distribution of that report, but also by the buggy topic(s) for which the report is intended. They also estimated Ensemble Averaging technique to combine IR and topic modeling in DBTM. They used Gibbs sampling to train DBTM on historical data with identified duplicate bug reports and then detect other not-yet-identified duplicate ones.

Another problem we are discussing is the automated labeling of topic models which is a unique problem statement. Many different topic models have been proposed, which can extract interesting topics in the form of multinomial distributions automatically from text. Although the discovered topic word distributions are often intuitively meaningful, a major challenge shared by all such topic models is to accurately interpret the meaning of each topic. Indeed, it is generally very difficult for a user to understand a topic merely based on the multinomial distribution, especially when the user is not familiar with the source collection. It would be hard to answer questions such as “What is a topic model about?” and “How is one distribution different from another distribution of words?”. The paper [27] tries to resolve this question.

In this paper, we will also be discussing Software Traceability and a Statistical Semantic Language Model for Source Code.

2. Relevant Papers

2.1 Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling

Keywords

- ***Latent Dirichlet Allocation (LDA)*:** Latent Dirichlet allocation (LDA) is a generative statistical model that allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similar
- ***Topic Model:*** A topic model is a type of statistical model for discovering the abstract "topics" that occur in a collection of documents
- ***Information Retrieval:*** Information retrieval (IR) is the activity of obtaining information resources relevant to an information need from a collection of information resources.

Motivational Statements

The ability to detect whether two (or more) reported bugs are duplicates has many benefits; like it saves developers’ efforts pondering over the same bug. Moreover, it points out the bugs which have not been fixed (partially or completely).

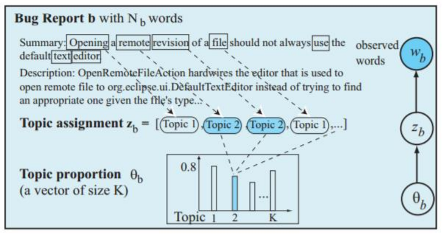
Hypothesis

This paper introduces DBTM, a duplicate bug report detection model that takes advantage of not only IR-based features but also topic-based features from a novel topic model, which is designed to address textual dissimilarity between duplicate reports.

Related Work

There has been significant amounts of prior work in this topic. Jalbert and Weimer [4] use a binary classifier model and apply a linear regression over textual features of bug reports computed from their terms’ frequencies. Support Vector Machine (SVM) was utilized by Sun et al. [5]. To train an SVM classifier, all pairs of duplicate bug reports are formed and considered as the positive samples and all other pairs of non-duplicate bug reports are used as the negative ones. The key limitation of ML approaches is their low efficiency. Earlier approaches have applied traditional information retrieval (IR) methods to this problem [1, 2]. Hiew et al. [1] use Vector Space Model (VSM) by modeling a bug report as a vector of textual features computed from Tf-Idf term weighting measurement. Vector similarity is used to measure the similarity among bug reports.

Informative visualizations


Bug Report b with N_b words
Summary: Opening a remote revision of a file should not always use the default text editor
Description: OpenRemoteFileAction hardwires the editor that is used to open remote file to org.eclipse.ui.DefaultTextEditor instead of trying to find an appropriate one given the file's type...
observed words
w_b
Topic assignment z_b = [Topic 1, Topic 2, Topic 2, Topic 1,...]
z_b
Topic proportion θ_b
(a vector of size K)
0.8
Topic 1 2 K
θ_b

The key insights needed to understand topic modeling are mentioned in the above visualization. Assume the document b contains N_b words. LDA associates to each document b two key parameters: a) Topic Assignment Vector z_b. Each of the N_b positions in document b is considered to describe one technical topic. Thus, topic assignment vector z_b for b has the length of N_b. Each element of the vector z is an index to one topic. b) Topic Proportion θ_b. A document b can describe about multiple topics. Thus, LDA associates to each document b a topic proportion θ_b to represent the significance of all topics in b. How we label these topics for better insights is a research are we discuss in section 3.4

Baseline Results

The results in the paper based on accuracy comparison to other models is impressive. The paper has provided hardware setup and the results of the algorithm including time of running the model which can be used to compare future models. One can compare the efficiency of the model using accuracy and compare the time efficiency. For a new bug report, in 57% of the detection cases, DBTM can correctly detect the duplication (if any) with just a single recommended bug report (i.e. the master report of the suggested group). Within a list of top-5 resulting bug reports, it correctly detects the duplication of a given report in 76% of the cases. With a list of 10 reports, it can correctly detect in 82% of the cases. In comparison, DBTM achieves higher accuracy from 10%- 13% for the resulting lists of top 1-10 bug reports. That is, it can relatively improve REP by up to 20% in accuracy.

Tutorial materials

The paper provides a case study like example where we can understand the advantages of this approach. The paper also does a good work defining all the key words the algorithm is referring to and the pseudocode makes it a lot easier to understand the model.

Future work

This model has performed well compared to other models which give a tough scope for improvement. Although the data sets are not wide ranging and the universal application of this model must be tested across various development platforms. The authors make an ambitious claim that DBTM can be 20% more accurate compared to state-of-the art models. This model must be applied in real time environment which will give us the scope to evaluate and validate the performance of this model.

Delivery tools

The paper provides a detailed pseudocode for the implementation of the DBTM model but they have not provided any direct interface that can be used by a user to run the model on custom data sets.

Handling issues of early papers

Earlier papers primarily concentrated on Machine Learning and Information retrieval methods. This paper resolves the low efficiency results by performing about 20% better than state-of-the art techniques for duplicate bug detection. Information retrieval methods do not consider the fact duplicate bugs will have a very high similarity in the topics used to describe the bug. This paper addresses both these issues.

Machine learning approaches should be deprecated since natural language is a more complex and unique problem to simply classify using an SVM or any other Machine Learning model. Information retrieval and Topic modeling will continue being the forefronts of the future research on this topic. Topic modeling is not the most concrete algorithm today and its improvement in the future will lead to a better DBTM model.

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

Keywords

- ***S-component:*** An LDA model which is generated based on a source file. The source file contains comments and identifiers which contribute to the words of the topics generated.
- ***B-component:*** An LDA model which is generated based on a bug report. The bug report consists of the technical description of the bug which helps identify the topic of the bug.

Motivational Statements

Locating bugs in source code based on bug reports is a time-consuming task, which prompts the requirement of an automated approach to help developers narrow the search space of source code files. 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. A bug report and the corresponding buggy source files often share some common buggy technical topics. Some source files in the system might be more buggy than the others. All these points help and make realize the importance and feasibility of detecting buggy files.

Hypothesis

This paper proposes an automated approach to help developers reduce the search effort by narrowing the search space of buggy files. 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.

Related Work

Lukins et al. [7] 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.

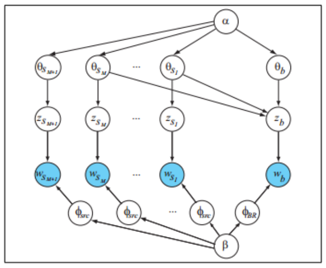
TRASE [11] is another approach that combines LDA with prospective traceability, i.e. capturing developers' activities during development, for tracing between architecture-based documents and source code.

There are various other approaches that the authors of this paper have gone through and summarized their approaches before coming up with their own algorithm, papers such as Premraj et al. [8], Ostrand et al. [9] and Bell et al. [10]

Handling issues of early papers

Previous papers have used unbalanced data with a huge number of negative examples, which are the incorrect pairs of bug reports and 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%.

Informative visualizations



There are two hyper parameters α and β whose conditional distributions are assumed as in LDA. 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 $s_1, s_2, \ldots s_m$ 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 θ_1, θ_2, ...θ_{sm} to z_b. For each source document, there are 3 variables that control s: z_b, θ_s and ϕ_{src}. The overall model is represented in the above figure.

Baseline Results

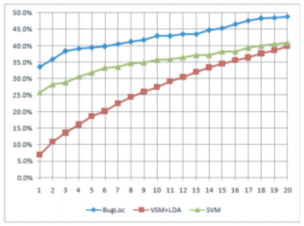


Fig. 11: Accuracy Comparison on Jazz dataset

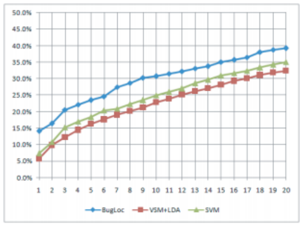


Fig. 13: Accuracy Comparison on Eclipse dataset

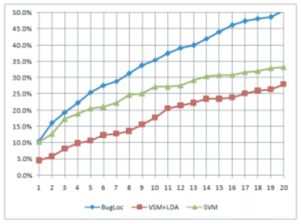


Fig. 12: Accuracy Comparison on AspectJ dataset

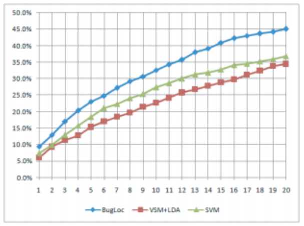


Fig. 14: Accuracy Comparison on ArgoUML dataset

The blue lines in the above graphs represent the performance of bugscout model. As per the paper the model has outperformed most of the models that are existing.

Tutorial materials

The paper provides 3 examples in which the all the parameters of the model are evaluated. Providing practical examples is the best way to provide tutorials rather than just provide definitions which was handled very well in this paper.

Future work

The model should add probabilistic models to take into consideration the following two parameters. Recently modified files have a higher probability of having bugs which can be included in ranking the files to predict buggy files. Depending on who the assignee for the bug is there is a possibility to better predict which file contains the bug based on which topic that assignee has developed. As mentioned they are planning to explore measures of Gibbs sampling on the data sets.

2.3 Software traceability with topic modeling

Keywords

- ***Automated Software Traceability:*** The goal of automatically discovering relationships between software artifacts (e.g. requirement documents, design documents, code, bug reports, test cases) to facilitate efficient retrieval of relevant information.
- ***Retrospective traceability:*** Artifact relationships are inferred from a static set of artifacts that have been generated in the past. Information Retrieval techniques are good at retrospective traceability.
- ***Prospective traceability:*** This method generates trace links as soon as the artifacts are created and modified during the development process.

Motivational Statements

Managing the relationships that exist between different software artifacts is a challenge for software developers. Automating this process is a non-trivial task and the authors propose a model that allows for the semantic categorization of artifacts and the topical visualization of a software system.

Hypothesis

This paper marks the initial investigation into combining prospective link capture with machine learning techniques, and there are many open research questions. The results suggest that the combination of prospective traceability with topic modeling is a promising area of research that can be useful in practice.

Related Work

Earlier approaches to Topic Modeling have employed LSI to generate models. Latent Semantic Indexing (LSI) has made significant inroads into the area of automated traceability [12, 13] These works have focused retrospective traceability using software artifacts including source codes, while the authors focus mainly on prospective traceability using on text-based artifacts such as requirements/design documents; but not source codes.

Informative visualizations

The authors made sure to provide good visualizations to explain their algorithm and its scope. They have provided flow charts for the LDA model they used and another image which describes how software artifacts are linked to specific modules.

Flowcharts representing relation between software traceability and topic modeling are provided. The interaction between the three tools described in the paper is also provided (TRASE, ACTS and TEAM).

Baseline Results

The paper provides detailed results including the hardware configuration and recall rate which are necessary for necessary for comparison with future work. The paper also provides various new features which were not available in previous models used for traceability, and all these features need to be implemented in future work to do better than this paper.

Tutorial materials

The authors have provided the definitions of the techniques they have used but on a very high level. There are many high-level techniques that went into writing this paper which makes it difficult to define and explain the techniques at a low level. They concentrated on the primary definition of the techniques and their limitations.

Future work

Although this paper has come up with a new approach to software traceability their results do not seem the best. This paper only provides its performance against one data set(EasyClinic) and the authors have mentioned that this model only beats the LSI model using the precision-recall parameter. This does not seem convincing enough to say that this model is better than LSI model. The data can be skewed to favor this algorithm. To make the comparison robust the author must compare more performance parameters across various datasets. Hence there is a lot of room for improvement in this paper.

This paper is only an initial investigation into combining prospective link capture with machine learning techniques, leaving many research questions open: Can prospective link capture be successfully combined with retrospective IR techniques? Is it possible to relate the automatically generated topics to high-level concepts such as features, non-functional properties, or concerns? Is it possible to successfully perform topic modeling on long artifacts by section?

Also, the topics can be provided at finer levels of granularity to minimize the number of components to examine. The representation of topics should be made clearer because similar high-probability words appeared in more than one topic.

Delivery tools

The authors have provided a bunch of tools which are responsible for separate parts of the model. TRASE, which is a search engine over the artifacts of the project which dynamically learns an LDA topic model on the search results in real-time. The user's interaction with TRASE can be recorded with ACTS (tool 2), which produces trace links. These trace links are then augmented with topical information and are visualized within TEAM (tool 3).

Handling issues of early papers

This paper had a precursor in the form of LSI and the topic modeling approach described in this paper was primarily compared to LSI performance. The LDA model gained a lot of ground on LSI since it considered new attributes like user interaction, and complementing features from Retrospective traceability and Prospective traceability.

2.4 Automatic Labeling of Multinomial Topic Models

Keywords

- ***Multinomial word distributions:*** This is a 'bag of words model'. Even though there is not conditioning on preceding context, nevertheless still gives the probability of a ordering of terms. Any other ordering of the bag of words will also have the same probability. Hence the distribution is called the multinomial distribution of words.
- ***K-L divergence:*** It is a measure of difference between two probability distributions P and Q. It is sometimes defined as the information gain achieved if P is used instead of Q

Motivational Statements

A big challenge today is to label a multinomial topic so that a user can interpret the discovered topic. Without understandable labels, the use of topic models in real world applications is seriously limited. So far, these labels have been generated in a manual subjective way. The requirement for automating this and generating labels in an objective way is clear. Manual label generation can be biased towards user's opinion. Relying manual labeling also makes it difficult to apply such topic models to online tasks such as summarizing search results.

Hypothesis

The objective of the paper is to generate understandable semantic labels for each topic, given a set of latent topics extracted from a text collection in the form of multinomial distributions. LDA is one techniques that is used to derive prominent topics from text after processing it.

Related Work

The unique quality of this paper is that it is one of the first attempts to label topics generated from various techniques. Previous attempts to label topics are very trivial and use approaches like labeling the topic with the top priority word returned in that topic.

Most of the related work uses a multinomial word distribution to represent a topic. The paper [14] generalized the representation of a topic model as a multinomial distribution over n-grams. Such topics are labeled with either top words in the distribution or manually selected phrases.

Informative visualizations

There are various visualizations provided in this paper for clear understanding of this approach. The semantics of any latent topic is fully captured by a multinomial distribution. Any reasonable measure of the semantic relevance of a label to a topic should compare the label with this distribution. An intricate description of this algorithm has been provided as a visualization in this paper.

The idea is to represent a candidate label with a multinomial distribution of words, which can then be used to compare with the topic model distribution to decide whether the label and the topic have the same meaning, which is the first-order relevance. Another visualization captures how the first-order relevance is calculated from the above mentioned multinomial distribution.

Baseline Results

This algorithm is unique in its generation of results. It is very difficult to convert the results of this algorithm into actual performance metrics which makes it very difficult to provide a concrete set of baseline results. The results of this algorithm are labels for the topics generated from text documents. One needs to compare these results with the topic name given by a human. The question arises of what exactly should match with the human produced results, is it the complete label? should the tense of both the labels match? how would we handle synonyms?

Due to all these questions, it will be very difficult for a paper in the future to compare its results to this paper. It must be an abstract comparison via human intuition rather than using any performance metrics. The results of this paper are however compared and provided in a tabular format.

Tutorial materials

One step to understand the requirements or the problem statement of this paper is to understand LDA. Learning LDA can be done using the internet. Although there are not too many new terms/ complex techniques being used in this paper the authors provide a very detailed explanation of all the techniques they have used to build their algorithm in the paper itself.

Future work

Candidate labels are evaluated to generated actual labels. Generation of these candidate labels is an important part of this model. There must be some new approach to generating these candidate labels which will improve the performance of this model.

Another interesting approach is to incorporate prior knowledge such as domain ontology, which will be helpful in predicting the topic labels better. Another research direction is to study how to generate labels for hierarchical topic models.

Delivery tools

They built an index for each collection and implemented the topic labeling methods using the Lemur toolkit http://www.lemurproject.org/

They generated candidate clusters using another NLP tool kit http://opennlp.apache.org/

They delivered their own model using algorithm, no links for an interactive tool of this model has been provided in this paper.

Handling issues of early papers:

Since this is one of the new approaches to handle labeling of there are no known issues that this paper resolves which had occurred in previous papers.

This is a very significant problem that the paper has tried to solve. LDA and other algorithms are good at generating topics using probability of word occurrences, but there has been no known method to label these topics.

2.5 A Statistical Semantic Language Model for Source Code

Keywords

- ***n-gram Model:*** In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech.
- ***Lexical Code Token:*** A lexical code token is a unit in the textual representation of source code and associated with a lexical token type including identifier, keyword, or symbol, specified by the programming language.
- ***Lexical Code Sequence:*** A lexical code sequence is a sequence of consecutive code tokens representing a portion of source code.

Motivational Statements

Recent research has successfully applied the statistical ngram language model to show that source code exhibits a good level of repetition. The n-gram model is shown to have good predictability in supporting code suggestion and completion. However, the state-of-the-art n-gram approach to capture source code regularities/patterns is based only on the lexical information in a local context of the code units. Based on SLAMC, the authors developed a new code suggestion method, which is empirically evaluated on several projects to have relatively 18–68% higher accuracy than the state-of-the-art approach.

Hypothesis

In this paper the authors introduce SLAMC a novel statistical Semantic Language Model for source Code that incorporates semantic information into code tokens and models the regularities on their semantic annotations (called sememes), rather than their lexical values (lexemes). Based on SLAMC the authors built a code suggestion engine. The engine suggests a ranked list of sequences of tokens that would complete the current code to form a meaningful code unit and most likely appear next. The top-ranked sequences are unparsed into lexemes and suggested.

Related Work

There are various papers that have solved this problem using various approaches. Predominantly statistical language models have been used before in Software Engineering. A previous paper Hindle et al. [15] use n-gram model with lexical tokens to show that source code has high repetitiveness which forms evidence for development of such prediction engines.

Code repetition which allows code prediction is also a technique observed in Gabel et al. [16]. They reported syntactic redundancy at levels of granularity from 6-40 tokens. Another paper Han et al. [17] have used Hidden Markov Model (HMM) to infer the next token from user-provided abbreviations. Grapacc [18] mines and stores API usage patterns as graphs and matches them against the current code.

Informative visualizations

This paper does not have visualizations since the paper is more of mathematical modeling devoid of any flow that can be represented as a visualization. Although there is a lot of text the authors have used tables to explain certain parts of their model.

Baseline Results

The authors have done a very good job providing base line results for future work. They have used 10-fold cross validation make the results more robust and reliable. For variation of their model the authors evaluated the accuracy of their predictions.

The data set consists of nine systems with a total of more than 2,039KLOCs. For comparison, they collected the same data set of Java projects. To evaluate on C# code, they also collected nine C# projects. This huge and varied data set used also improves the credibility of the results. They have provided the different results that occurred if Top-1 predictions are considered, Top-2 predictions are considered and so on. Finally, their model SLAMC looks to perform the best among models provided in previous research in other papers.

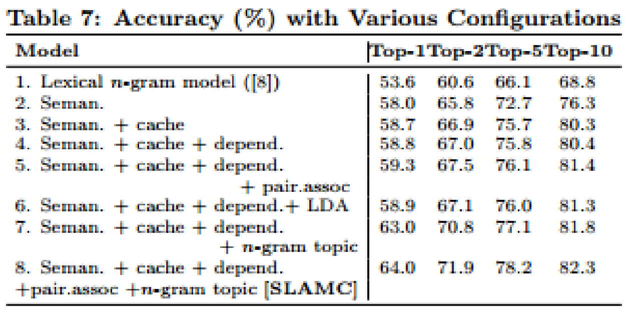


Table 7: Accuracy (%) with Various Configurations

Model	Top-1	Top-2	Top-5	Top-10
1. Lexical n-gram model ([8])	53.6	60.6	66.1	68.8
2. Seman.	58.0	65.8	72.7	76.3
3. Seman. + cache	58.7	66.9	75.7	80.3
4. Seman. + cache + depend.	58.8	67.0	75.8	80.4
5. Seman. + cache + depend. + pair.assoc	59.3	67.5	76.1	81.4
6. Seman. + cache + depend.+ LDA	58.9	67.1	76.0	81.3
7. Seman. + cache + depend. + n-gram topic	63.0	70.8	77.1	81.8
8. Seman. + cache + depend. +pair.assoc +n-gram topic [SLAMC]	64.0	71.9	78.2	82.3

They also made sure to provide their hardware configuration to compare performance such as latency caused due to this model.

Tutorial materials

The authors defined various terms they have used in this model and made sure to provide the basis on which they are building their model. Other than the basic definitions the paper makes a good attempt at making the paper sensible and self-sufficient.

They have also provided pseudo code for training their n-gram topic model and for the code suggestion engine they built. These are the two main components of this paper.

Future work:

Scripting languages have fundamentally different structure compared to compiled languages. Given some scripting languages are more complex it would be a good research direction to see how such a prediction tool can be built. The first question would be to see if the same level of high repetitiveness is also observable in scripting languages, given their more complex structure and relatively lesser training data.

This model must be tested in a user oriented real life environment. Accuracy is also not the best evaluator of this prediction engine. It would be very annoying for a user to keep receiving false positive predictions all the time. Hence recall rate would be a better quantifier of such an engine.

Delivery tools

The paper does not provide any interface where a user can use it to test this environment. Although the pseudo code for the model is provided, the best bet would be to email the authors for the source code.

Handling issues of early papers

The n-gram model from previous research has good predictability and is used to support code suggestion. In comparison, SLAMC has key advances. First, its basic units are semantic code tokens, which are incorporated with semantic information, thus providing better predictability. Second, SLAMC's n-grams are also complemented with pairwise association. It allows the representation of co-occurring pairs of tokens that cannot be efficiently captured with n consecutive tokens in n-grams. Finally, a novel n-gram topic model is developed in SLAMC to enhance its predictability via a global view on current technical functionality/concerns.

Han et al. [17] have used Hidden Markov Model (HMM) to infer the next token from user-provided abbreviations. Abbreviated input is expanded into keywords by an HMM learned from a corpus. In comparison, their model has only local contextual information, while SLAMC has also n-gram topic modeling and pairwise association.

2.6 Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis

Keywords

- ***Fault localization:*** Fault localization is a process to find the location of faults. It determines the root cause of the failure. It identifies exactly where the bugs are.
- ***Bug report:*** A bug report is something that stores all information needed to document, report and fix problems occurred in a software or on a website.

Motivational Statements

Given a bug report, it is often painstaking for a developer to manually locate the buggy source code in the code base. Existing approaches still do not deal with the following two issues well, leading to accuracy loss. The two issues are file size consideration (especially large files) and stack traces. According to the results in the paper, their approach is able to significantly improve BugLocator, a representative fault localization approach, on all the three software projects by using these two new issues by including segmentation and stack trace analysis while locating the source of the bug.

Hypothesis

Provide a better algorithm performance than provided by traditional information retrieval methods. The BTracer project developed by the authors of this paper uses BugLocator, a previous model which uses Information Retrieval methods for locating buggy files, along with segmentation and stack trace analysis to provide better localization of buggy files based on the bug reports. The outcome is that the algorithm assigns algorithm assigns a final score to candidate files to detect the bug.

Related Work

There are various parallel approaches to what this paper has described. Mainly the related work is around Information retrieval methods to locate buggy files. Ye et al. [19] defined six features measuring the relationship between bug reports and source files, and defined a ranking model to combine them, using a learning-to-rank technique.

Saha et al. [20] proposed an approach called BLUiR and gained significant improvement over BugLocator by replacing Vector Space Model with structured information retrieval model.

Le et al. [21] applied topic modeling to infer multiple abstraction levels from documents, and extended Vector Space Model to locate code units by computing similarities resulting from several abstraction levels.

Tantithamthavorn et al. [22] proposed an approach to improve BugLocator by leveraging co-change histories, which relied on the assumption that files that have been changed together are prone to be fixed together again in the future.

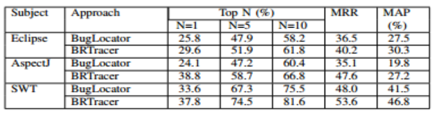
Most of the above approaches are for locating buggy files and for providing new features to existing bug location algorithms. There is significant overlap in the models used in all these papers. While one paper tries to embed dynamic quality of code changes into consideration, the other came up with unique idea of using segmentation and stack trace analysis to make locating buggy files more accurate.

Informative visualizations

This paper does not have any significant flow although the authors could have made a visualization of a high-level flow since there are multiples segregated steps in this algorithm. The paper has a lot of math and examples to help follow the process and since this paper already uses BugLocator, understanding its flow will throw light on how this algorithm has tried to fill in the gaps in the BugLocator and provide better results.

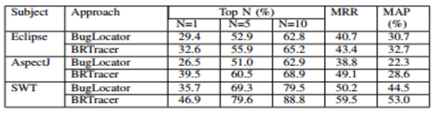
Baseline Results

TABLE II: Overall Effectiveness without Considering SBRs



Subject	Approach	Top N (%)			MRR	MAP
		N=1	N=5	N=10		(%)
Eclipse	BugLocator	25.8	47.9	58.2	36.5	27.5
	BRTracer	29.6	51.9	61.8	40.2	30.3
AspectJ	BugLocator	24.1	47.2	60.4	35.1	19.8
	BRTracer	38.8	58.7	66.8	47.6	27.2
SWT	BugLocator	33.6	67.3	75.5	48.0	41.5
	BRTracer	37.8	74.5	81.6	53.6	46.8

TABLE III: Overall Effectiveness Considering SBRs



Subject	Approach	Top N (%)			MRR	MAP
		N=1	N=5	N=10		(%)
Eclipse	BugLocator	29.4	52.9	62.8	40.7	30.7
	BRTracer	32.6	55.9	65.2	43.4	32.7
AspectJ	BugLocator	26.5	51.0	62.9	38.8	22.3
	BRTracer	39.5	60.5	68.9	49.1	28.6
SWT	BugLocator	35.7	69.3	79.5	50.2	44.5
	BRTracer	46.9	79.6	88.8	59.5	53.0

The authors provide their results across three projects and also compare it to the BugLocator performace to provide evidence of significant performance. They also provided their results pertaining to two environments where in one, they consider the similar bug reports to locate buggy files and two, where they do not consider similar bug reports. This provides a solid foundation for future work to be compared against.

Tutorial materials

The authors have tried two write a self-contained paper with the algorithm defined in the paper itself. The papers description itself works as a tutorial. There are no external tutorial materials provided in the paper, other than related work which helps understand some approaches that the paper has used.

Future work

The authors have been self-critical of their work after completing the paper which is impressive and have provided some sources of limitations and threats to validity, mostly about their experiment design.

The authors want to investigate mechanisms to automatically choose suitable values for the parameters “beta” and “l” based on analyzing the target project. This is to avoid generality and make train their model per the specifics of the project.

The segmentation used in the paper does not consider code structures or other characteristics of the source code. The research would be that if putting same code structures in the same segment help the performance?

Other features like increasing the score for a file based on the files size must be implemented. It is relatively more probable for a file large to contain the bug.

Delivery tools

The project has been delivered as a downloadable package at http://brtracer.sourceforge.net/. They have provided a user interface which explains the tools and provides a link to the tool itself, which is written in Java. This very impressive and ambitious on the part of the authors.

Handling issues of early papers:

The primary motive of this paper was to handle issues which the other papers haven’t and improve the performance results, since there have already been established bug locators before.
Compared to all other papers the authors have their unique two heuristics.
Most papers use Information retrieval methods such as LDA, but fail to address the problem of large files and do not consider stack traces.
To deal with the above two issues this paper has introduced segmentation of large files and includes stack trace analysis, while scoring the candidate bug files

2.7 Auto-completing Bug Reports for Android Applications

Keywords

- ***Android:*** Android is a mobile operating system developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets.
- ***Auto-completion:*** Autocomplete, or word completion, is a feature in which an application predicts the rest of a word a user is typing.
- ***STR (reproduction steps):*** The steps to reproduce (STR) must be as clear as possible, preferably with screenshots and/or test data. The steps should also be definite.

Motivational Statements

The complexity of “apps” has been increasing, making development and maintenance challenging. Additionally, current bug tracking systems are not able to effectively support construction of reports with actionable information that directly lead to a bug's resolution. Software maintenance activities are known to be generally expensive and challenging. The approach that FUSION (model described in the paper) employs is generalizable to other current mobile software platforms, and constitutes a new method by which off-device bug reporting can be conducted for mobile software projects.

Hypothesis

FUSION helps users auto-complete reproduction steps in bug reports for mobile apps. FUSION encourages a new direction of research aimed at improving reporting systems. It reduces the lexical gap between reporters of bugs and developers. FUSION, that relies on a novel Analyze and then Generate paradigm to enable the autocompletion of Android bug reports to provide more actionable information to developers.

Related Work

Bug reporting systems worldwide are the related works that are in context to this paper. Bugzilla, Google Code Issue Tracker, Jira, Mantis etc.

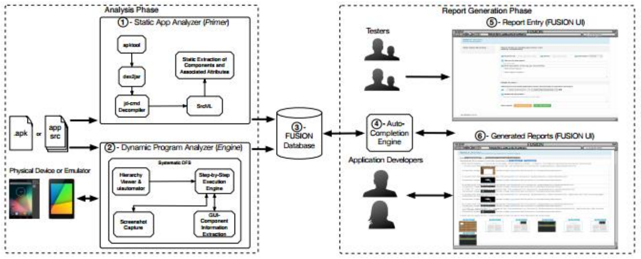
Most current issue tracking systems rely upon unstructured natural language descriptions in their reports. However, some systems do offer more functionality. For instance, the Google Code Issue Tracker (GCIT) offers a semi structured area where reporters can enter reproduction steps and expected input/output in natural language form (i.e. the online form asks: "What steps will reproduce the problem?"). Nearly all current issue trackers offer structured fields to enter information such as tags, severity level, assignee, fix time, and product/program specifications. Some web-based bug reporting systems (e.g. Bugzilla, Jira, Mantis, UserSnap, BugDigger) facilitate reporters including screenshots.

Study instruments

They have used participants to evaluate their performance against other bug reporting systems. The paper asked the users to conduct two maintenance activities involving reporting and reproduction of real bugs in open source apps. Later they interviewed the users with a standard set of questions and noted the time users took to complete the exercise, for the bug reporting and the reproduction part.

They were trying to answer five questions around the usefulness of using FUSION by inexperienced and experienced users. The authors have done a thorough job of designing the experiments and collecting the results. Although FUSION did not do well on all fronts they have reported that there are some areas which could mean extremely good news for the tool.

Informative visualizations


Analysis Phase
① - Static App Analyzer (Primer)
apktool
dex2jar
jd-cmd Decompiler
SrcML
Static Extraction of Components and Associated Attributes
.apk
or
app src
② - Dynamic Program Analyzer (Engine)
Systematic DFS
Hierarchy Viewer & uiautomator
Step-by-Step Execution Engine
Screenshot Capture
GUI-Component Information Extraction
Physical Device or Emulator
③ - FUSION Database
Report Generation Phase
Testers
⑤ - Report Entry (FUSION UI)
④ - Auto-Completion Engine
⑥ - Generated Reports (FUSION UI)
Application Developers

FUSION's Analyze and Generate workflow corresponds to two major phases. In the Analysis Phase FUSION collects information related to the GUI components and event flow of an app through a combination of static and dynamic analysis. Then in the Report Generation Phase FUSION takes advantage of the GUI centric nature of mobile apps to both auto-complete the steps to reproduce the bug and augment each step with contextual application information.

Baseline Results:

The paper designed an experiment to evaluate the results. They used real life participants who are bug reporters to evaluate results. They found out that the experienced users appreciated the tool much more than inexperienced users. The experienced users preferred using FUSION based tool rather than a simpler web UI which was preferred by inexperienced users.

The design of experiments was primarily to understand the real-time performance of FUSION. The results quantitative and qualitative from this paper are very subjective and are not to be taken as a strong basis for comparing with new work.

Developers using FUSION can reproduce more bugs compared to traditional bug tracking systems such as the GCIT is one of the key takeaway from the results.

Tutorial materials

One of the most impressive and practical tutorial material I have found in the papers. Although most of the algorithm is available in the thorough paper itself. Intuitive visualizations have been provided to understand the algorithm.

The videos provided at http://android-dev-tools.com/ are primarily for users to get used to the tool by viewing video material. Such an approach to get real life users to evaluate the tool is very important for the research to thrive.

The paper provides how to get key insights into the paper by researching the tools used to implement FUSION: *APKTool, Dex2jar, jd-cmd, Android Debug Bridge, Hierarchy Viewer* and *UIAutomator*.

Future work

One key component of this paper is the DFS engine which for fault localization. Improving the DFS engine by supporting more gestures which and add program specific gestures is a research direction to consider.

Using FUSION for reporting feature requests is another direction that the tool can be useful in. There will be specific shortcomings for this, which must be solved.

Delivery tools

"THE FUSION REPLICATION PACKAGE" is a replication package containing a live instance of FUSION running on the web, and the full dataset of all results obtained during our comprehensive empirical evaluation. The package can also be accessed at http://android-dev-tools.com/

Handling issues of early papers

This paper has used techniques used in other papers to its advantage, although the paper does not attempt to handle any previously researched techniques so that it can better them by fixing issues.

Overall this paper tries to tackle the problem of making bugs more reproducible by adding some intelligence in the bug reporting system. Most related work does not assist the user in reporting bugs. While FUSION is more complex to use, it can make the most inexperienced of users write better bug reports which help reproduction of the bug and help fault localization.

2.8 Automated triaging of very large bug repositories

Keywords

- ***Triaging bugs:*** Assign degrees of urgency to bugs in bug repositories.
- ***Tokenization:*** Is the process by which each bug report is reduced to a simpler form by removing punctuation marks and converting all uppercase characters to lowercase characters.

Motivational Statements

Limited research has been done in developing a fully automated triager, one that can first ascertain if a problem report is original or duplicate, and then provide a list of 20 potential matches for a duplicate report. Manual triaging of bug reports is both challenging and time consuming. The very nature of the English language entails that two people can use vastly different language to describe the same issue.

Hypothesis

The motive is to develop an automated triaging system that can be used to assist human triagers in bug tracking systems. One that can first ascertain if a problem report is original or duplicate, and then provide a list of 20 potential matches for a duplicate report.

Related Work

Research dating back as far as 2006, Hiew [23] applied a practical threshold based method on similarity scores for identifying duplicate reports in four open source repositories: Firefox, Eclipse, Apache 2.0, and Fedora Core.

Wang et al. [24] use natural language processing along with execution information to build a suggested list of 20 originals with recall rates between 67% and 93%. BM-25F [25] is a factor based extension of BM-25 text similarity algorithm along with a gradient descent function.

Informative visualizations

There is nothing much in this paper to provide intuitive visualizations over. The paper discusses document pre-processing and then evaluating the similarity score over various parameters after preprocessing.

Baseline Results

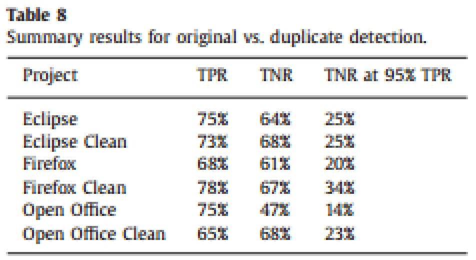


Table 8
Summary results for original vs. duplicate detection.

Project	TPR	TNR	TNR at 95% TPR
Eclipse	75%	64%	25%
Eclipse Clean	73%	68%	25%
Firefox	68%	61%	20%
Firefox Clean	78%	67%	34%
Open Office	75%	47%	14%
Open Office Clean	65%	68%	23%

The is table provides a summary of experiments conducted by the paper. There is no discussion in the paper regarding computational performance of this algorithm which is a very important aspect for real time systems. Although the true positive and true negative rates are given, there is much more to observe to concretely evaluate the results.

Tutorial materials

The algorithm uses the document preprocessing to evaluate the 24-similarity metrics that the paper has defined. The pseudo code for part of the algorithm is provided in the paper, although it was not necessary to provide the pseudocode for finding the longest common subsequence in this paper.

The model in the paper has evaluated many classification algorithms and implemented the one with best results. One can find documentation online regarding Random forest classifiers and other Machine Learning classifiers.

Future work

The assumption in this paper is that all the 24 metrics have significant importance in deciding which bugs are duplicate and which aren't. The author must get some insights into how exactly these metrics are causing the classification to better the performance or reduce it.

Investigate the effects of feature selection, as well as the inclusion of additional features that can assist in differentiating between originals and duplicates.

This paper needs to look at more sophisticated classifiers than Random Forests which might be overfitting the results. The specificity of the use case defines what classifier is best for providing results.

Delivery tools

There is no package or any particular website provided for users to interact with. However most of the algorithm is either provided in the paper itself or provided in the form of Weka Machine learning tool's library for various classification techniques.

Handling issues of early papers

Single similarity measures used in earlier papers fail to adequately differentiate between original and duplicate reports due to the diversity of human language. By applying a suite of 24 similarity measures and similarity summary statistics, this paper reduces the false matches caused by the diversity in language.

3. Conclusion

This paper discusses the impressive algorithms being researched to improve the quality of bug handling systems. While some of them have used non-traditional ways such as using Topic modeling. Predominantly papers have been using Information Retrieval methods at the heart of Duplicate Bug detections systems. One of the papers has also forced bug reporters to provide better description of bugs in the reports, especially when they are inexperienced.

To start their own research in Duplicate bug detection systems one has to start with the traditional Information Retrieval papers and move into researching Machine Learning models, since the Information Retrieval have been used predominantly and can be considered standard baselines for evaluation of future work.

BTracer[29] has compared itself to BugLocator and has provided evidence of better performance. FUSION [30] was compared against Google code Issue Tracker, Bugzilla and various other bug reporting systems, the paper has provided significant evidence as to why FUSION can be the system for the future and for the experts in the field. The design of experiments in the FUSION was impressive and showed real ambition to make users adapt to the new FUSION environment. While another paper solves the problem of labeling topic models which must evaluated in a very subjective matter.

A combination of all these methods would form the next level of intelligent bug handling tool. A tool which can resolve duplication and triaging, a tool which can give auto-completion recommendations and can immediately locate the buggy file based on the bug report.

4. REFERENCES

[1] L. Hiew. Assisted detection of duplicate bug reports. Master's thesis, University of British Columbia, 2006. (14)

[2] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In ICSE'07. IEEE CS, 2007. (23)

[3] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In ICSE'08, pages 461–470. ACM, 2008. (26)

[4] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In Int. Conf. on Dependable Systems and Networks, pp. 52–61. 2008(16)

[5] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In ICSE'10. ACM, 2010. (25)

[6] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In Proceedings of the 2012 IEEE/ACM International Conference on Automated Software Engineering, ASE'12, pages 70–79, ACM Press, 2012.

[7] 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.

[8] 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

[9] 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.

[10] 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.

[11] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In ICSE '10. ACM Press, 2010.

[12] H. Jiang, T. Nguyen, I. Chen, H. Jaygarl, and C. Chang. Incremental latent semantic indexing for effective, automatic traceability link evolution management. In Proc of Automated Software Engineering (ASE), 2008.

[13] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In Proc Int'l Conf on Program Comprehension, 2006.

[14] X. Wang and A. McCallum. A note on topical n-grams. In University of Massachusetts Technical Report UM-CS-2005-071, 2005

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In Proceedings of the 2012 International Conference on Software Engineering, ICSE'12, pages 837–847, IEEE CS, 2012.

[16] M. Gabel and Z. Su. A study of the uniqueness of source code. In Proceedings of the 2010 ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 147–156. ACM, 2010

[17] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pages 332–343. IEEE CS, 2009.

[18] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In Proceedings of the 2012 International Conference on Software Engineering, ICSE'12, pages 69–79. IEEE Press, 2012.

[19] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in Proc. FSE, 2014, pp. 66–76.

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

[21] T.-D. B. Le, S. Wang, and D. Lo, "Multi-abstraction concern localization," in Proc. ICSM, 2013, pp. 364–367

[22] C. Tantithamthavorn, A. Ihara, and K. ichi Matsumoto, "Using cochange histories to improve bug localization performance," in Proc. SNPD, 2013, pp. 543–548.

[23] L. Hiew, Assisted detection of duplicate bug reports, The University Of British Columbia, 2006 Ph.D. thesis.

[24] X. Wang, L. Zhang, T. Xie, J. Anvik, J. Sun, An approach to detecting duplicate bug reports using natural language and execution information, in: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 461–470.

[25] C. Sun, D. Lo, S.-C. Khoo, J. Jiang, Towards more accurate retrieval of duplicate bug reports, in: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2011, pp. 253–262.

[26] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In ASE'11, pp. 263-272. IEEE CS, 2011.

[27] Qiaozhu Mei, Xuehua Shen, and ChengXiang Zhai. 2007. Automatic labeling of multinomial topic models. In Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining

[28] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013.

[29] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In ICSME'14. IEEE, 2014.

[30] K. Moran, M. Linares-V´asquez, C. Bernal-C´ardenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15), to appear, 2015

[31] S. Banerjee et al., Automated triaging of very large bug repositories, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.09.006