How Should We Read and Analyze Bug Reports: An Interactive Visualization using Extractive Summaries and Topic Evolution

Shamima Yeasmin Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
shy942@mail.usask.ca, {chanchal.roy, kevin.schneider}@usask.ca

ABSTRACT

Software projects evolve over time as bugs are addressed and new functionalities are added. Managing bugs can be a significant challenge for a project manager especially when the number of reported bugs is large, and the manager needs to consult with them. It is also preferable that developers new to a project first familiarize themselves with the project and the reported bugs before actually working on them. In order to reduce developers' time and efforts for reading a bug report, in this paper, we propose a visualization technique that provides an extractive summary visualization for a given bug report. In addition, our proposed technique assists the developers or managers in reviewing a project's bug reports by interactively visualizing insightful information using topic analysis on the bug reports. In order to validate the effectiveness of our proposed visualization technique, we conducted a task-oriented user study involving six participants and a case study using 3914 bug reports. The findings from both studies show that our visualization technique is promising, and it can assist the comprehension and analysis of bug reports. The results from the user study indicate that visualized summary is relatively preferred to the non-visualized summary for quick comprehension of bug reports.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Techniques—*visualization, topic modeling*

Keywords

Bug Report, Topic Evolution, Summary, Visualization

1. INTRODUCTION

Software bugs are the issues that hinder the work of the users of a software system, and they are reported in the form of software bug reports. A typical bug report contains several pieces of information such as problem description of a software bug, steps to reproduce the bug, relevant source code, and data dumps (i.e., content of the memory at the time of the failure). During the maintenance and evolution of a software system, a project manager often needs to consult with a number of previously filed bug reports. The goal is to determine which parts of a given project are more vulnerable (i.e., affected by bugs) and thus deserve more attention. The task is trivial when there are only a few bugs reported. However, the number of bugs of a software project generally increases over time, and it poses a great challenge for a project manager to analyze such a huge collection of bug reports [26]. On the reporting of a new bug, a triager (i.e., the manager) attempts to classify the bug into existing categories and looks for the duplicate bugs [26]. The goal is to identify not only the similar bugs reported earlier but also the developers who worked on those bugs so that existing knowledge and expertise can be applied when fixing or resolving the new bug. However, both tasks require the manager to read a number of bug reports from the repository [20] [26]. When developers start working on an existing project, it is also preferable that they first familiarize themselves with the project and its reported bugs, which is likely to help them avoid or resolve the similar reported issues.

Given the sheer numbers and size of the bug reports, analyzing a large collection of reports manually is highly unproductive [26], and the managers, the triagers, and the developers spend a significant amount of time and efforts in consulting with the reports [20]. One way to support them in this regard is to offer useful summaries containing less information instead of the original bug reports having lots of text and comments [20] [26]. The summary of a bug report is a condensed form of important information extracted from the report, and it could be a useful mean for the readers to comprehend the reported bug using less amount of time and cognitive effort. There exist several approaches that propose summarization of the bug reports [20] [26] [19]. Most of these techniques generate extractive summaries, where the summary statements are extracted out of their contexts from the bug reports. However, it is not well understood whether those extractive summaries are useful in practice or not, especially for the developers who do not have prior knowledge about the reported bugs. Project managers generally use traditional bug tracking systems (e.g., *BugZilla* [1], *JIRA* [6]) during the maintenance and evolution of a software product. These tracking systems facilitate different basic features such as search, addition, deletion or archiving of a bug report. However, they often fail to deliver useful insights on the bug reports required for the work of the managers.

In this paper, we propose two major visualizations on software bug reports. First, we replicate the *hurried bug summarization* technique of Lotufo *et al.* [19], and visualize the extractive summaries for a given bug report for easier comprehension. This visualization highlights the summary statements within their contexts in the original bug report, and helps a reader comprehend the reported bug with less time and effort.

In our second visualization, we apply topic modeling on a collection of bug reports and visualize the bug topic evolution (i.e., evolution of discussed technical topics) over time. Topic modeling is a statistical probabilistic model that discovers the hidden document structures such as topics from a collection of documents by employing machine learning techniques [21] [25]. This visualization provides an important insight on the different parts of a software system containing bugs, and such information can aid in different project management activities. By inspecting topic evolution over time in a time-windowed manner, developers can make themselves aware of frequently occurring types of bugs in the earlier versions of the project and can take necessary precautions.

In order to explore the capabilities and limitations of our visualization for extractive summaries, we conduct a task-oriented user study. The study shows that the participants found the visualized summary relatively preferable to non-visualized summary for quick comprehension of bug reports. In order to validate the applicability of our topic evolution visualization, we conduct a case study, where we experimented with 3914 bug reports of Eclipse-Ant software system. Several insightful pieces of information about the project were found from the topic evolution visualization. For example, the topic "Tool Launch and Configuration" was found as one of the most frequently occurring topics (i.e., more in Table 1) in the bug reports. Such information assists a developer or a manager in paying more attention on particular parts of the project.

This paper is an extended version of our previous Early Research Achievements paper [29], where we just outlined the idea with limited evaluation. In this work, we improve and explain the techniques in details and conduct extensive experiments.

This paper makes the following contributions:

- An interactive visualization of a bug report summary that conveniently links summary sentences to their context.
- A visualization that shows the topic evolution of bug reports over time.
- A detailed drill-down from a topic's time-segments to its related bug reports.
- A search feature that helps developers explore related issues regarding a given topic-keyword.

The rest of the paper is organized as follows. Section 2 describes an example use case scenario, Section 3 discusses the schematic diagrams of our proposed visualization approach, Section 3.1 discusses bug report extractive summarization technique, Section 3.2 presents visualization based on topic modeling, Section 4.1 discusses the conducted user study and its results. Then Section 4.2 discusses the case study, Section 5 identifies the possible threats to validity, Section 6 discusses the existing studies related to our research, and finally Section 7 concludes the paper with future plan.

Table 1: Example Topics with Keywords

No.	Topic Label	Keywords
1	Plugin Support	require user issu possible feature support realli gener plugin plan
2	Editor Outline	editor xml view outlin content action elem open docum associ
3	Tool Launch and Configuration	launch tool dialog configur view extern config select menu button
4	Version log	reproduce memori version view instal window attach open log time
5	Page Preference	tab page prefer buttoon classpath dialog home runti default pref

2. AN EXAMPLE USE CASE SCENARIO

Let us consider a software maintenance scenario, where a triager assigns software bugs to the developers for fixation. Prior to the assignment of a newly reported bug, the triager attempts to categorize it into an existing category, and then looks for duplicates. She uses an existing bug tracking system such as *Bugzilla* [1], and the system generally returns a number of bug reports based on their severity and recency. She then applies an existing summarization technique (e.g., using [19], [26] or [20]) on those reports for comprehending and analyzing them. However, she notes that the statements of the summary are taken out of context, and they do not express a clear and consistent view about the reported bug. The triager might also be interested in collecting certain other information for her work such as– (1) Did the reported bugs exist in earlier versions?, (2) Are there other bugs related to the reported critical bugs?, (3) When did they first appear?, and (4) What are the technical issues associated with these bugs? However, neither the bug tracking systems nor the existing approaches in the literature answer these questions satisfactorily. In this paper, we thus attempt to solve these two research problems by–(1) analyzing the effectiveness of extractive summaries for bug reports, and (2) mining and visualizing the useful information from bug reports. We decompose the two research problems into the following research questions:

- **RQ$_1$:** Does the visualization of summary statements in their contexts help a reader comprehend the reported bug effectively?
- **RQ$_2$:** Does a newly reported bug associate with a technical issue that is the most frequently reported?
- **RQ$_3$:** Which bug reports in the repository do discuss a frequent technical issue?

Our technique helps the triager in the above scenario to conveniently dig deeper into a large collection of bug reports containing reports from the previous versions. For example, now, she is able to read the summaries of the bug reports as shown in Fig. 3, which is colour coded and relatively short than the original report. From our technique, she is also able to know that the top most occurring topics are 'Plugin', 'Tools', 'Page' and so on as shown in Table 1. From the visualization of a topic such as in Figure 4, she can explore when a topic peaked. For further information, she can drill-down into the bug reports containing that topic as in Fig. 5. In order to answer our **RQ$_1$**, we conduct a task-oriented

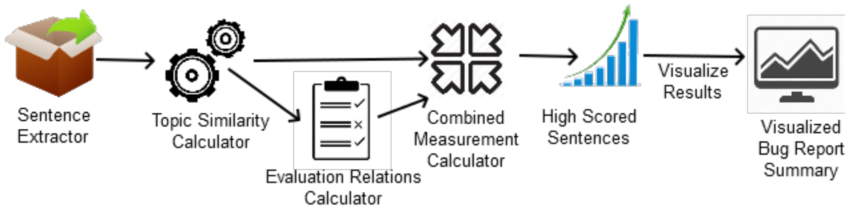

Sentence Extractor
Topic Similarity Calculator
Evaluation Relations Calculator
Combined Measurement Calculator
High Scored Sentences
Visualize Results
Visualized Bug Report Summary

Figure 1: Schematic Diagram of Bug Report Summarizer

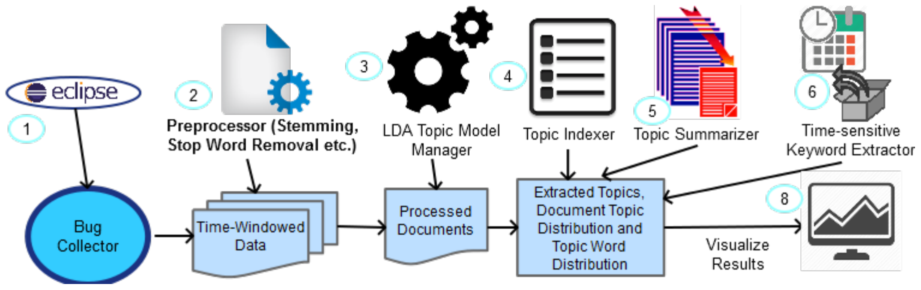

eclipse
1
Bug Collector
2
Preprocessor (Stemming, Stop Word Removal etc.)
Time-Windowed Data
3
LDA Topic Model Manager
Processed Documents
4
Topic Indexer
5
Topic Summarizer
6
Time-sensitive Keyword Extractor
Extracted Topics, Document Topic Distribution and Topic Word Distribution
8
Visualize Results

Figure 2: Schematic Digram of Topic Evolution Visualizer

user study described in Section 4.1. We also conduct a case study for validating the effectiveness of topic evolution of bug reports, which answers our **RQ$_2$** and **RQ$_3$**.

3. PROPOSED APPROACH

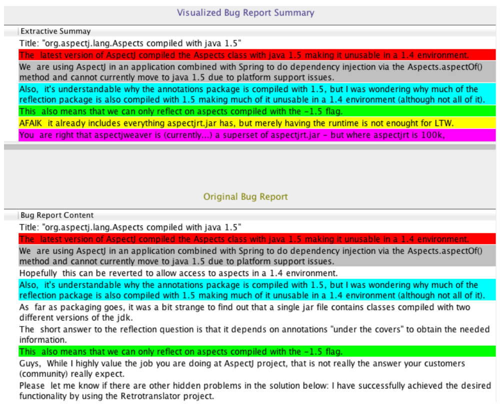
We divide our proposed tool in two different parts: Part I creates extractive summaries of bug reports and visualizes them, and Part II is related to the features that are based on topic modeling. The schematic diagram for extractive summary visualization is shown in Fig. 1. We create extractive summaries of bug reports by applying methods described in Lotufo *et al.* [19], where we use twitter API for sentiment analysis [9]. The goal is to determine the sentiment of each of the sentences. Then, we visualize the summaries following the methodologies as discussed in Section 3.1. Part II consists of two individual phases- analytics and visualization as shown in the schematic diagram of Fig. 2. In the analytics phase, we collect bug reports from a popular bug tracking system *BugZilla*, apply LDA (Latent Dirichlet Allocation) topic modeling on the reports to extract topics as described in Section 3.2. For LDA, we use a Java implementation of LDA called JGibbLDA [5]. In the visualization phase, we visualize topic evolution with the help of a popular Java chart library called JFreeChart [4].

3.1 Part I: Visualization of Bug Report Extractive Summaries

One of the primary objectives of our proposed tool is to provide meaningful information to developers through analysis and visualization of bug reports. Thus, we design our tool to provide a comfortable reading experience with bug reports. In the following, we discuss the detailed design and methodologies for summary visualization of bug reports

3.1.1 Visual Design: Extractive Summary Visualization

The visualized bug report summary is presented in Fig. 3. During designing the visual summary of a bug report, we



Visualized Bug Report Summary

Extractive Summay

Title: "org.aspectj.lang.Aspects compiled with java 1.5"

The latest version of AspectJ compiled the Aspects class with java 1.5 making it unusable in a 1.4 environment.

We are using AspectJ in an application combined with Spring to do dependency injection via the Aspects.aspectOf() method and cannot currently move to java 1.5 due to platform support issues.

Also, it's understandable why the annotations package is compiled with 1.5, but I was wondering why much of the reflection package is also compiled with 1.5 making much of it unusable in a 1.4 environment (although not all of it).

This also means that we can only reflect on aspects compiled with the -1.5 flag.

AFAIK it already includes everything aspectjrt.jar has, but merely having the runtime is not enought for LTW.

You are right that aspectjweaver is (currently...) a superset of aspectjrt.jar - but where aspectjrt is 100k,

Original Bug Report

Bug Report Content

Title: "org.aspectj.lang.Aspects compiled with java 1.5"

The latest version of AspectJ compiled the Aspects class with java 1.5 making it unusable in a 1.4 environment.

We are using AspectJ in an application combined with Spring to do dependency injection via the Aspects.aspectOf() method and cannot currently move to java 1.5 due to platform support issues.

Hopefully this can be reverted to allow access to aspects in a 1.4 environment.

Also, it's understandable why the annotations package is compiled with 1.5, but I was wondering why much of the reflection package is also compiled with 1.5 making much of it unusable in a 1.4 environment (although not all of it).

As far as packaging goes, it was a bit strange to find out that a single jar file contains classes compiled with two different versions of the jdk.

The short answer to the reflection question is that it depends on annotations "under the covers" to obtain the needed information.

This also means that we can only reflect on aspects compiled with the -1.5 flag.

Guys, While I highly value the job you are doing at AspectJ project, that is not really the answer your customers (community) really expect.

Please let me know if there are other hidden problems in the solution below: I have successfully achieved the desired functionality by using the Retrotranslator project.

Figure 3: Bug Report Summary Visualization

kept two things in mind: first, the length of the summary should be significantly smaller than the original bug report so that the developer will have fewer sentences to read; and second, the visualization must be done in a way that can fulfill a developer's intention for reading it effectively. Therefore, to create a good summary we follow the approach proposed by Lotufo *et al.* [19] which is automatic and extractive. We also restrict the number of sentences of our summary to ten.

In our tool, the visualized summary is represented along with the visualized original bug report to the developer. Each sentence in the summary is coloured with a unique colour and the same sentence in the original bug report is also coloured by the same colour. This kind of visualization can help the developer to understand the summary from the context. As our created summary is extractive, sometimes it might be difficult for the developer to gather the desired idea from it. That's why, when the summary sentences are also highlighted in the original bug report using the same colours, the developer is able to study the sentences that precede and follow the summary sentences in original bug report, which can aid her in understanding summary sentences in the context.

3.1.2 *Methodologies: Creation of Bug Report Extractive Summaries*

In order to create extractive summary of a bug report, we first apply the hurried bug summarization technique proposed by Lotufo *et al.*, and visualize the summary using our tool as described in Section 3.1.1. We chose to apply this summarization technique as it is automatic, light-weight and it shows 12% improvement over previous approach [19]. We utilize the following two important hypothesis from Lotufo *et al.* for creating extractive summary of each bug report.

Measuring Topic Similarity: The first hypothesis of Lotufo *et al.* states that in a bug report, the relevance of a sentence is higher if it shares more topics with other sentences. Like Lotufo *et al.*, we use a cosine similarity metric to measure the similarity of the sentences.

Measuring Evaluation Relations: The second hypothesis of Lotufo *et al.* states that the relevance of a sentence is higher the more it is evaluated by other sentences and the more relevant are the sentences that evaluate it. To identify evaluation relations between sentences in a bug report, polarity detection is performed. Sentiment analysis has been used in the polarity detection of movie reviews [10], political reviews [27] and so on. Polarity detection first filters evaluation sentences and then finds out whether the sentence is positive or negative. In order to avoid manual classification of polarity of sentences Go *et al.* [15] use a training set containing 800,000 positive and 800,000 negative Twitter messages which were automatically annotated as negative or positive based on emoticons presented in the comments. We use the same approach for measuring the evaluation relation between two sentences. We compute an overall score for each sentence by combining topic similarity and evaluation relation measures as described above. We rank all sentences of a bug report based on their overall score and finally choose the top ten sentences as an extractive summary as of Lotufo *et al.*.

3.2 Part II: Topic Evolution of a collection of Bug Reports

In Part II, our second visualization technique, (i) generates as well as shows topic evolution of each topic automatically, (ii) then for further inspection it retrieves all software bug reports associated with a given topic along with their Bug Report IDs and titles, and (iii) provides a searching option so that one can search bug reports by keywords associated with a topic. In this subsection we will describe the visual design and methodologies used in Part II of our proposed approach.

3.2.1 *Visual Design: Topic Evolution of a collection of Bug Reports*

Topic Evolution Visualization for Each Topic: Once a developer selects a dataset (i.e., a collection of bug reports) the system automatically applies topic modeling [5] to it. After performing some analytics as described in Section 3.2.2 on the produced topic model, the tool depicts the topic evolution of several topics derived from the dataset. From this visualized output, both experienced and novice developers can analyze which type of topics are evolved most of the time and associated with most of the bugs. By analyzing

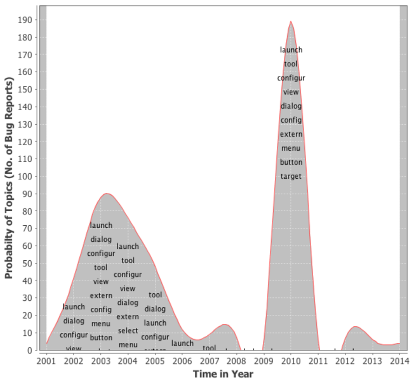
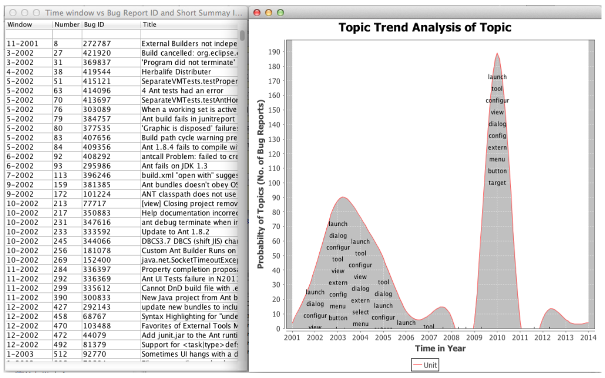


Figure 4: Topic Evolution Example (Topic 3: Tool Launch and Configuration)

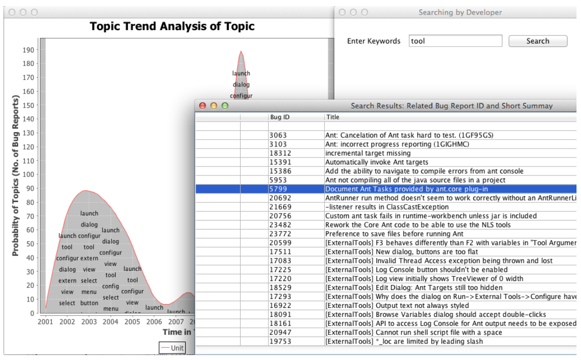

Time window vs Bug Report ID and Short Summay I...
Topic Trend Analysis of Topic
Probabilty of Topics (No. of Bug Reports)
Time in Year
Unit

Window	Number	Bug ID	Title
11-2001	8	272787	External Builders not indepe
3-2002	27	421920	Build cancelled: org.eclipse.
3-2002	31	369837	'Program did not terminate'
4-2002	38	419544	Herbalife Distributer
5-2002	51	415121	SeparateVMTests.testProper
5-2002	63	414096	4 Ant tests had an error
5-2002	70	413697	SeparateVMTests.testAntHor
5-2002	76	303089	When a working set is active
5-2002	79	384757	Ant build fails in junitreport
5-2002	80	377535	'Graphic is disposed' failure
5-2002	83	407656	Build path cycle warning pre
5-2002	84	409356	Ant 1.8.4 fails to compile wi
6-2002	92	408292	antcall Problem: failed to cre
6-2002	93	295986	Ant fails on JDK 1.3
7-2002	113	396246	build.xml "open with" sugges
9-2002	159	381385	Ant bundles doesn't obey OS
9-2002	172	101224	ANT classpath does not use
10-2002	213	77717	[view] Closing project remov
10-2002	217	350883	Help documentation incorrec
10-2002	231	347616	ant debug terminate when ir
10-2002	233	333592	Update to Ant 1.8.2
10-2002	245	344066	DBCS3.7 DBCS (shift JIS) char
10-2002	256	181078	Custom Ant Builder Runs on
10-2002	269	152400	java.net.SocketTimeoutExce
11-2002	284	336397	Property completion proposa
11-2002	292	336369	Ant UI Tests failure in N201
11-2002	299	335612	Cannot DnD build file with .e
11-2002	390	300833	New Java project from Ant b
12-2002	427	292143	update new bundles to inclu
12-2002	458	68767	Syntax Highlighting for "unde
12-2002	470	103488	Favorites of External Tools M
12-2002	472	44079	Add junit.jar to the Ant runti
12-2002	492	81379	Support for <task|type>def
1-2003	512	92770	Sometimes UI hangs with a d

Figure 5: Topic Drill-Down in the Context

bug report topic evolution, a manager can check the year in which a given project contains the highest number of bugs, and which of the topics occur more frequently. Thus, the manager can identify the parts of the project which are affected by the highest number of bugs, and such information can aid the manager in different decisions making activities associated with that software project. We use an area-graph based visual layout to represent topic evolution (i.e., content changes over time as in Fig. 4). Our tool generates the visual summary of each topic individually. However, this area layout is depicted by a set of keyword clouds in order to show the content evolution over time. The height of the area graph at each time-segment (here, a year) encodes the strength of the topic for that point (Fig. 4). Strength is calculated by the number of software bug reports containing that topic at the certain point.

Topic Drill-Down in the Context: If a developer requests for more information regarding a topic, then all bug report IDs and titles associated with that topic will be shown, as is shown in left part of the Fig. 5. In this way, the context of the bug reports will aid the developer in gathering enough knowledge to identify that topic precisely. During resolving a new bug, a developer might be interested to gather knowledge from existing similar bugs so that she can apply existing expertise in order to fix that bug. She can collect important keywords from the provided new bug as well as can check which topic contains those keywords. From topic drill-down feature, she can investigate all bug reports under


Topic Trend Analysis of Topic
Probabilty of Topics (No. of Bug Reports)
Time in
Unit
launch
dialog
configur
tool
extern
view
select
menu
button
config
Searching by Developer
Enter Keywords
tool
Search
Search Results: Related Bug Report ID and Short Summay

		Bug ID	Title
		3063	Ant: Cancelation of Ant task hard to test. (1GF95GS)
		3103	Ant: incorrect progress reporting (1GIGHMC)
		18312	incremental target missing
		15391	Automatically invoke Ant targets
		15386	Add the ability to navigate to compile errors from ant console
		5953	Ant not compiling all of the java source files in a project
		5799	Document Ant Tasks provided by ant.core plug-in
		20692	AntRunner run method doesn't seem to work correctly without an AntRunnerLi
		21669	-listener results in ClassCastException
		20756	Custom ant task fails in runtime-workbench unless jar is included
		23482	Rework the Core Ant code to be able to use the NLS tools
		23772	Preference to save files before running Ant
		20599	[ExternalTools] F3 behaves differently than F2 with variables in "Tool Argumer
		17511	[ExternalTools] New dialog, buttons are too flat
		17083	[ExternalTools] Invalid Thread Access exception being thrown and lost
		17225	[ExternalTools] Log Console button shouldn't be enabled
		17220	[ExternalTools] Log view initially shows TreeViewer of 0 width
		18529	[ExternalTools] Edit Dialog: Ant Targets still too hidden
		17293	[ExternalTools] Why does the dialog on Run->External Tools->Configure have
		16922	[ExternalTools] Output text not always styled
		18091	[ExternalTools] Browse Variables dialog should accept double-clicks
		18161	[ExternalTools] API to access Log Console for Ant output needs to be exposed
		20947	[ExternalTools] Cannot run shell script file with a space
		19753	[ExternalTools] *_loc are limited by leading slash

Figure 6: Search by Keywords

a topic of interest in order to enrich her knowledge and thus, can apply existing expertise on the newly reported bug.

Searching by Keywords: The search feature is presented in Fig. 6. A topic is associated with several keywords. A developer may be interested in inspecting any of them. Consider a scenario where a developer finds a topic described by keywords such as 'launch', 'tool', 'view' and so on as in Fig. 6. She might be curious to know which type of 'tool' related bugs are mentioned by this topic. To help her we provide a search option, where the developer is able to perform a search on the entire bug report collection for a keyword. In Fig. 6, search results are shown containing bug report IDs and titles for the keyword 'tool'.

3.2.2 *Methodologies: Topic Evolution of a collection of Bug Reports*

In the following subsections, we will discuss all analytics we have performed for visualizing topic evolution of bug reports over time.

Pre-processing: Our preprocessor unit performs stemming on each token of the corpus. Here, token means a single word. Stemming is required to keep the root words, which ensures that only the unique root words appear in the extracted topics. We also remove stop words from our dataset. For removing stop words from our corpus we use an existing stop-word list, which is available online [8].

Time-Windowed Data Creation: To visualize the evolution of a topic over time, we split the dataset in a time-windowed manner. We divide the bug report dataset under several years and months according to their reported dates. The reported date of a bug report is collected from the original bug report.

Topic Analysis: A topic implies the thematic content common to a set of text documents. After preprocessing, the collection of bug reports are considered as a collection of text documents. Each document is composed of a sequence of words. Each topic is referred to by a set of keywords that form a topic-keywords distribution and this distribution is independent within all text documents. Each keyword in a topic has a probability that represents its likelihood of appearing in that topic. To retrieve time-sensitive keywords for each topic, our topic model uses two matrices: a document-topic distribution matrix and a topic-word distribution matrix. From the document-topic distribution matrix, we see that each document may be associated with all topics with some probability. Therefore, we use a threshold-based technique to assign a topic to a document. After considering different values we use a threshold of 0.01 (i.e., document-topic distribution probability) in our experiment as using this threshold results reasonable number of topics can be assigned to each document. We also restrict a document from having no more than five topics to ensure that each bug report should be associated with some dominated topics, not all. During topic modeling we restrict the number of iterations to 3000 and generate 20 topics, as we observed too many common keywords among topics can be associated if we consider more than 20 topics for the dataset we have used.

Topic Ranking: LDA topic modeling outputs 20 topics, which are randomly ordered. One of the reasons to apply topic modeling is to discover the topics that appear in most of the bug reports. Therefore, we rank the topics so that the most important topics appear first in the topic list. To rank topics, we consider generic topics that occur in all documents. So, the rank of a topic is measured by a combination of both topic content coverage and topic variance (i.e., how far a topic is spread out). For bug report topic ranking we adapted the topic ranking algorithm of Wei *et al.*, who used a topic ranking algorithm to rank topics in an email-summarizer. We calculate mean, variance, and then rank them as follows:

$$mean, \mu_t = \frac{\sum_{i=1}^{N}(\theta_{ti} \times N_i)}{\sum_{i=1}^{N} N_i} \tag{1}$$

$$variance, \sigma_t^2 = \frac{\sum_{i=1}^{N}((\theta_{ti} - \mu_t)^2 \times N_i)}{\sum_{i=1}^{N} N_i} \tag{2}$$

$$rank_t = \mu_t \sigma_t \tag{3}$$

Given a document-topic distribution θ, the rank of a topic t is computed and N is the total number of documents and θ_{ti} is the probability distribution of topic t in document i.

Filtering Keywords for Topics: In LDA topic modeling, some extracted keywords of a given topic may not be ideal for understanding the theme (i.e., name or label or definition) of the topic. There are some keywords that appear in all documents, which are too generic to express any topic definition. Therefore, we summarize each topic by filtering these type of keywords from that topic. The strategy is, a keyword is important for a topic - if it appears frequently in that topic and does not occur frequently in any other topics. This actually measures the TF-IDF (Term Frequency and Inverse Document Frequency) score of each keyword in a bug reports collection. Here, the occurrence of a keyword is calculated by the probability measure of that keyword in a topic. We adapted the approach of Wei *et al.* to topic analysis on a bug reports collection. Given topic-word distribution λ, we compute the weight of each keyword kw_m derived from the original LDA model and then sort them in ascending order to select the top n keywords per topic. we calculate the weight of each keyword using the following equation:

$$weight(kw_m) = \lambda_{tm} \times log\frac{\lambda_{tm}}{(\prod_{j=1}^{T} \lambda_{jm})^{1/T}} \tag{4}$$

Where T is the total number of topics and λ_{tm}, is the probability of keyword kw_m in topic t.

Time-sensitive Keyword Extraction: In our tool, we visualize topic evolution over time, where the most frequently

Table 2: Difficulty and Frequency Scales

Difficulty Levels	Scales	Frequency Levels	Scales
Very hard	5	Very often	5
Hard	4	Often	4
Not hard nor easy	3	Sometimes	3
Easy	2	Hardly	2
Very Easy	1	Never	1

occurring keywords are selected and presented in each time-segment of a topic. At the beginning, we divided the collection of bug reports into subsections, each of them is associated with a particular time such as a month and a year. For each time-segment of a topic, we extract keywords, which appear frequently both in that topic and time-segment. We also consider the highest peak from different values within a time interval. For example, assuming that in 2003 all bug reports from Eclipse-Ant were reported in January, March, June, and December. Also assuming that a given topic is contained in 60, 47, 89 and 56 bug reports respectively during these four months. Then our system will consider the value 89 as the number of bug reports associated with that topic, against the year of 2003. To extract time-sensitive keywords, we adapted the procedure of Wei *et al.*. We collect each word dw_m of sub collection s, compute its weight, and select the top n keywords as time-sensitive keywords for each topic-segment after sorting them in ascending order. Given topic t, term frequency of word dw_m is denoted as TF_{tsm}, and so word-weight is calculated using the following equation:

$$weight(dw_m) = \frac{TF_{tsm}}{\sum_s^S TF_{tsm}} + \lambda_{tm} \times log \frac{\lambda_{tm}}{(\prod_{j=1}^{T} \lambda_{jm})^{1/T}} \quad (5)$$

4. EXPERIMENT AND DISCUSSION

In Part I of this research, we apply visualization to the extractive summaries of bug reports to address a practical problem associated with such summaries. We employ topic modeling on a collection of bug reports to show topic evolution of bug reports over time in Part II. In order to validate the applicability of our bug report summary visualization, we conduct a user study with six participants where we collect feedback from the participants. We also conduct a case study for validating the effectiveness of topic evolution of bug reports. In the following two subsections we discuss conducted user study as well as case study in detail.

4.1 Evaluation of Bug Report Visualization: A Task-Oriented User Study

In this subsection, we discuss different parts of the conducted study such as task design, study participants, questionnaire for data collection, sessions of user study, result collection and data analysis.

4.1.1 Design of User Study

Task Design In order to evaluate the effectiveness of visualized bug report summaries, we design two tasks that involve the identification of duplicate bugs from a collection for a newly reported bug. Each of the tasks is simple enough to be accomplished by any participant, and at the same time sufficient enough for exploring the potential of any of the summary representation techniques.

T1: Identify duplicate bugs from a collection of eight bug reports for a given bug report by consulting plaintext or non-visualized extractive summaries of the bug reports.

Target usage: Evaluation of non-visualized summary.

T2: Identify duplicate bugs from a collection of eight bug reports for a given bug report by consulting visualized extractive summaries of the bug reports.

Target usage: Evaluation of visualized summary.

We choose two *Eclipse* bugs from *Bugzilla* [1] having IDs *62468 [3]* and *14890 [2]* for the study. We denote them as *Bug Report1* and *Bug Report2* respectively in the remaining of the chapter.

Study Participants We choose six graduate research students from Software Research Lab, University of Saskatchewan, as the participants for the study. However, we only select those who have substantial amount of programming experience that includes bug fixation or resolution, and some of them have professional software development experience. Due to the nature and expertise of the participants, we intentionally limit the number of participants to six.

Questionnaire We use a questionnaire to collect feedback from each of the participants during the study sessions, and the questionnaire contains the following questions:

Questionnaire for Non-Visualized summaries:

(1) Consult with plaintext or non-visualized summaries of bug reports, and label each of the eight candidate bug reports for duplicity. Use these labels for labeling: Duplicate (D), Near Duplicate (NRD), Not Duplicate (ND), Related (R) and Not Related (NR).

(2) In extractive summary, sentences are often extracted out of their contexts in the bug report, which might need to be consulted for easier comprehension. How much difficulty did you face in locating the contexts of the summary sentences in the original bug report?

(3) How often did you switch between the summary and the original bug report for context analysis?

(4) Rate the usefulness/effectiveness of a plaintext or non-visualized bug report summary in bug comprehension on the scale from 1 (least useful) to 10 (most useful).

(5) Rate the overall look and feel of the non-visualized summary on the scale from 1 (least helpful) to 10 (most helpful).

Questionnaire for Visualized summaries:

(6) Consult with visualized summaries of bug reports, and label each of the eight candidate bug reports for duplicity. Use these labels for labeling: Duplicate (D), Near Duplicate (NRD), Not Duplicate (ND), Related (R) and Not Related (NR).

(7) Rate the usefulness/effectiveness of visualized bug report summary in bug comprehension on the scale from 1 (least useful) to 10 (most useful)

(8) Rate the overall look and feel of visualized summary on the scale from 1 (least helpful) to 10 (most helpful).

(9) Do you think that the visualized bug report summary is more effective than plaintext or non-visualized bug report summary for quick comprehension of the reported bug?

(10) How to improve the visualized summary for more easier comprehension of bug reports? Please provide your suggestions.

Table 3: Problems of Non-visualized Summary

Motivating Factors	P1	P2	P3	P4	P5	P6	Avg.	Comment
Difficulty- Finding summary sentences in original bug report	5	3	5	5	5	4	4.5	Very hard
Frequency- Context switching between summary and bug report	3	2	3	3	4	4	3.17	Sometimes

Table 4: Rating of Efficiency and Look & Feel by Participants

Features	Approach	P1	P2	P3	P4	P5	P6	U-value	p-value	RD
Efficiency	Non-Visualized Summary	5	6	3	5	6	6	0	0.00512	**S**
	Visualized Summary	7	10	8	8	9	7			
Look & Feel	Non-Visualized Summary	5	7	4	3	6	6	1.5	0.01046	**S**
	Visualized Summary	7	10	8	7	9	7			

4.1.2 Study Session Coordination

We run our user study in two sessions- execution phase and evaluation phase as follows:

Execution Phase At the beginning of the user study, we brief the participants about the tasks to be completed. It usually takes 3-5 minutes. We divide the six participants into two groups- *Group A (P1, P3, P5)* and *Group B (P2, P4, P6)*. Participants in *Group A* perform tasks where they identify duplicate bug reports for *BugReport1* using non-visualized summaries and also do the same for *BugReport2* using visualized summaries. On the other hand, participants from Group B identify duplicate bugs for *BugReport2* using plaintext or non-visualized summaries and for *BugReport1* using visualized summaries. At first, the participants analyze the summaries as well as the bug reports of the candidate bugs for a given bug report, and then, fill up the first and sixth questions of the questionnaire (Section 4.1.1). The session lasts about 15-20 minutes on average.

Evaluation Phase In this phase, participants answer the rest of the questions from questionnaire (Section 4.1.1), where they compare the effectiveness of visualized bug report summaries with non-visualized bug report summaries in identifying duplicate bugs for a given bug report from a list of candidate bugs. We also collect qualitative suggestions from the participants on how to further improve the interactive visualization. This phase takes about 5-10 minutes on average.

4.1.3 Result Analysis and Discussion

We analyze the feedback collected from participants during evaluation phase, and contrast our visualized summaries with non-visualized summaries for determining effectiveness in the identification of duplicate bug reports. We apply two metrics- *Average Rating and Mann-Whitney U-Test [7]* for evaluation. The first metric shows whether two lists of measures are equal or not in terms of their central values, and the second one determines if the lists are significantly different from each other. Here, **Average Rating** averages a list of ratings, where each rating is associated with a certain interval. In our user study, we consider "1" as lowest rating and "10" as the highest rating. We compute average for the ratings on certain features of the bug report summaries such as *Convenience for context analysis*, *Difficulty in context analysis*, *Efficiency in bug comprehension* and so on. **Mann-Whitney U-Test** is a non-parametric statistical test that compares between two sets of ordinal measures. In our user study, this test is used to determine whether the ratings provided by the participants for the visualized summaries are significantly differ from that of non-visualized summaries. This test outputs two measures- U and p-values.

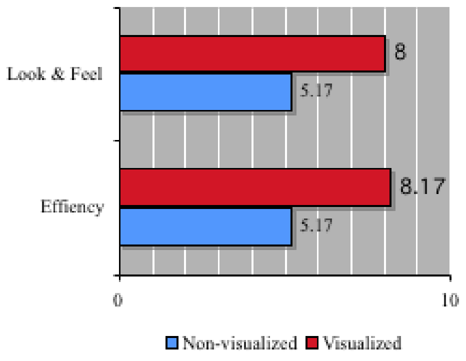


Figure 7: Rating for Summary Visualization

We consider a significance level of 0.05 i.e., if p-value is less than 0.05 for a pair of rating lists, then they are significantly different from each other and vice versa.

Motivating Factors of Visualization to Bug Report Summaries We apply visualization to the extractive summaries of bug reports to aid developers in comprehending bugs conveniently. In order to identify the motivating factors behind our visualization, we collect responses from the participants where we set appropriate scales to certain factors such as *Difficulty (i.e., finding summary sentences in original bug report)* and *frequency (i.e., context switching between summary and bug report)* in (Table 2). We transform the participants' feedback into numerical scales. Table 3 shows individual responses on the motivating factors and their average measures. From Table 3, we note that locating sentences of a non-visualized summary in original bug report is very hard for the participants. It is required when the developer cannot comprehend the summary sentences because of their inconsistencies and moves to the original bug report for contextual help. From Table 3, we note that although the participants attempt to focus on the summary for bug comprehension, sometimes still they need to switch between bug report and its summary. Thus, the responses from the participants indicate that the support for context analysis for the summary sentences of the bug report should be made more accessible and user-friendly to the developers, and we apply interactive visualization for that purpose.

Comparison between Visualized and Non visualized Bug Report Extractive Summaries We identify two features- *Efficiency* and *Look and Feel* to compare the performance between our visualized summaries and the traditional non-visualized summaries. Fig. 7 shows the average ratings for both features, where we note that visualized bug report summaries are highly rated compared to the non-visualized for each feature by the participants. In order to determine whether the ratings for visualized summaries are significantly higher than that of non-visualized ones, we conduct Mann-Whitney U-test. We apply this test on both sets of ratings from the same participants. Table 4 shows that the ratings are significantly different for both features- *Efficiency* and *look and feel* of the bug report summaries.

In our user study, all (i.e., six) participants select the option *Agree* with the fact that visualized bug report summary is more effective than non-visualized summary for comprehending the reported bug quickly. Three out of them (i.e., 50%) report that they *Strongly Agree* with this. Thus, most of the participants in our study highly agreed on the effectiveness of visualized bug report summary over non-visualized summary, which answers our $\mathbf{RQ_1}$.

We also observe the completion time that participants take for both tasks with non-visualized and visualized summaries. According to our experiments, the participants took 10.5 minutes on average in order to complete their tasks with non-visualized summaries, whereas they took 8.8 minutes on average for the same purpose with visualized summaries. Although the timing difference is small, this might be significant when large number of bug reports are handled by the developers. Thus in terms of task completion time, visualized summaries are found more effective than traditional plaintext or non-visualized summaries of bug reports for the comprehension of the bugs

Qualitative Suggestions from Participants Participants pinpoint some useful suggestions that can improve the visualization of the extractive summaries of bug reports, and they are listed as follows:

Keyword highlighting: Important keywords from the given bug report should be highlighted both in the summary and the original bug report of the candidate bugs.

Hyper linking: Summary sentences should be hyper linked to the corresponding in the original bug report.

Mouse hover capabilities: When a developer hovers the mouse on a summary sentence, the same sentence in the original bug report should be highlighted or blinked.

Look and feel: One of the participants suggest lighter colors for highlighting a sentence or keyword.

Text wrapping: The text should be wrapped in order to avoid scrolling for the summaries of the bug reports.

4.2 Evolution: An example case study

In software bug management, an existing bug repository always works as a good source of information. Sometimes a bug which is already fixed can be reopened. Search on existing bug repository is generally performed for finding similar or duplicate bugs. Thus, studying an existing bug database is beneficial even if the bugs are fixed. However, to examine the effectiveness of topic evolution, let us assume a scenario where a novice developer will soon start working on Eclipse-Ant, and at the beginning she wants to give a quick look at its bugs. At present, *Eclipse-Ant* contains 3914 bug reports and definitely studying all of them would take a long time. Therefore, in this situation, we are providing our tool that can aid her to more conveniently and quickly dig deeper into a large collection of bug reports including the bug reports from the previous versions. From our tool we can see 20 topics as output, each of which have 10 keywords. To keep this discussion simple, an example of the top most 5 topics together with their associated keywords

Table 5: Search Results in Terms of Bug Reports Retrieved

Keyword	Bugzilla	Proposed Tool
plugin	58	577
editor	371	702
tool	469	860
log	191	518
tab	130	533

Table 6: # of Bug Reports in Eclipse-Ant from 2001 to 2014

Year	# Bug Reports	Year	# Bug Reports
2001	17	2008	114
2002	484	2009	510
2003	776	2010	90
2004	728	2011	128
2005	479	2012	71
2006	256	2013	91
2007	155	2014	15

are provided in Table 1. We see that the 1st, 2nd and 3rd topics are about ‘Plug in’, ‘Editor’ and ‘Tool’ respectively. During searching these keywords are both in Bugzilla and our tool; a different number of bug reports result as shown in Table 5. Our tool retrieves more bug reports than Bugzilla for each keyword (Table 5). To investigate the reason behind this, we randomly as well as manually check results both from our proposed tool and Bugzilla. In Bugzilla, almost all retrieved bug reports contain searching keyword in their titles, because Bugzilla produces search results based on bug report titles only, where we consider the contents of the bug report in addition to the title during searching. In our scenario, from Table 1, the novice developer can gather an idea regarding the most occurring problems (i.e., bugs) in *Eclipse-Ant*, which are related to ‘plug-in’, ‘editor’, ‘tool’, ‘log’ and so on. To dig deeper she can also search by those keywords as in Fig. 6, and can have a clear idea about how many bug reports are associated with each top topic. Below are some questions we can use our tool to address for the above scenario.

- Which month/year was the most crucial period for Eclipse-Ant bugs?
- What was the most active topic in a given year such as 2003 or 2009?
- What was discussed in the most active topic?
- Which bugs are associated with the most active topic?

The developers as well as managers might ask these questions for several reasons. For example, during resource allocation the manager of a software project tries to determine which part of a given project is more problematic and thus needs more attention. By investigating the most active topics and their associated bug reports, the manager can identify the components or parts of a software project, which are largely affected by those bugs. The idea is to pay more attention on those components and allocate more time and effort for them while working with the next version of the project.

To answer the first question we need to investigate the top most topics, where they are in their peak. We can see that the year 2009 is the most active year for Eclipse-Ant bug reports for the top most five topics, two of them, topic-3 and

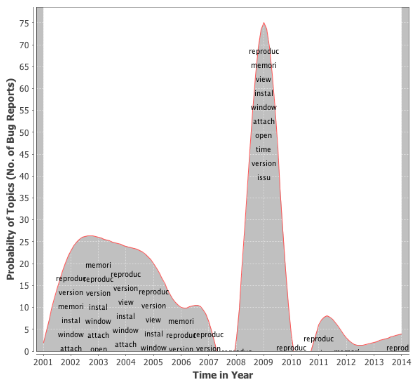


Figure 8: Topic-4 (Version Log)

topic-4, are depicted in Fig. 4 and Fig. 8 respectively. Then 2003 is the second most active year for this bug dataset. We also can relate it to the number of bug reports of Eclipse-Ant in each year from 2001 to 2014 as presented in Table 6. Here, although years 2003 and 2004 have the highest number of bug reports, the top-most 5 topics are not that active in these two year, compared with year 2009.

To address the second question, we notice that in 2003 the top 5 topics (Table 1) have the following number of bug reports, respectively: 63, 11, 88, 26, and 25. That means topic-3, "Tool Launch and Configuration", is the most active topic during 2003. However, in 2009 the number of bug reports for the top 5 topics were 173, 28, 189, 75 and 65, i.e. topic-3 is associated with the most bug reports in 2009 also. Now, in order to investigate the relevant bug reports under the most active topic (i.e., Tool Launch and Configuration), our detail drilldown feature can aid developers by showing all bug report IDs and titles associated with that topic as depicted in Fig. 5. This also answers our $\mathbf{RQ_3}$.

It is also observed from Table 1 that the most crucial topic, i.e., topic-3 contains keywords such as 'launch, 'tool', 'dialog', 'configur', 'view' and so on. We verify it with search results presented in Table 5, where we can see that the highest number of bug reports are retrieved against keyword 'tool' both from Bugzilla and our proposed tool. If a newly reported bug contains keywords from the most crucial topic, i.e., topic-3, the developer will pay more attention addressing that issue. This also answers $\mathbf{RQ_2}$.

5. THREATS TO VALIDITY

We identify a few potential threats to the validity of our findings. One of them is the lack of professional experience of the participants in bug report management. In practice, the task of identifying duplicate bugs from a bug report collection is performed by a triager who generally has some prior knowledge on the reported bugs. In order to mitigate the threat, we thus choose two frequent issues related to *memory leak* and *build dependency* which are often faced by the developers (i.e., participants). These issues also suit for our user study since we intended to evaluate our visualized bug report summary involving average developers rather than experts.

Second, the participants are chosen from among the peers for our user study, and some of them are from Software Research lab. Thus one can argue about the potential bias in the ratings by the participants for our system. While we cannot rule out the possibility of such bias, we adopted a careful technique in order to mitigate such bias in the evaluation. The study sessions with each of the participants were conducted in isolation and the evaluation was based on their instant working experience as well as their best judgment.

Third, the number of participants involved in our conducted user study is not enough. In order to mitigate this threat, we involve such graduate students who have substantial amount of programming experience that involves problem solving and bug fixation. Some of them also have professional software development experience.

Forth, the number of sample bug reports for the study is also limited. However, since our focus is on interactive summary visualization of an individual bug report, the volume of bug reports does not actually affect the studies much.

Fifth, in this paper, we experimented and reported results only for Eclipse bug reports, and the results may vary for bug reports from other application domains. However, Eclipse bug repository is a widely used dataset that has been used in several other research studies. Thus, the findings with this dataset are reliable and possibly comparable with other systems.

6. RELATED WORK

Analysis and visualization of bug reports are not new, and there have been a great many studies. To represent the evolution of bugs, D'Ambros *et al.* [13] propose a visualization technique called **System Radiography** that indicates which parts are the most problematic parts in a system. They also provide useful insight on the life cycle of a bug by another visualization technique- **Bug Watch**. Another technique is proposed by Dal Sassc & Lanza [11] for representing a fine-grained view of a bug report. To analyze bug tracking system, they also propose a visual analytic platform called **in*Bug**. Hora *et al.* [17] present a tool, **BugMaps,** to map the reported bugs to the defects in object oriented systems, and provide several interactive visualizations for decision support. To uncover the relationship of how an evolving software is affected by software bugs, D'Ambros & Lanza [12] propose a visual approach that shows the evolution of software entities at different levels of granularity. The main differences between those work and our technique are that (i) for visualization of bugs, we use topic evolution over time, but D'Ambros *et al.* use a matrix-based representation, Dal Sassc & Lanza provide a web-based visual analytics platform, and Hora *et al.* utilize Distribution Map and (ii) none of these existing studies visualizes the extractive summary of a bug report which we do.

PageRank algorithm [18] models the probability of reaching a web page from another page by estimating the relevance of both pages. Lotufo *et al.* [19] first use PageRank for unsupervised bug report summarization to develop a deeper understanding of the information exchanged in a bug report. Rastkar *et al.* [26] investigate the possibility of summarizing a bug report automatically and effectively so that the user can benefit from the smaller version of the entire artifact. In our research, besides showing topic evolution of bug reports, we also apply *hurried bug summarization* by Lotufo *et al.* [19] to create summaries and then visualize them in a convenient way which improves their understandability for the developer.

In software bug management, topic modeling has been used to classify bug reports from non-bugs [24], detect duplicate bug reports [23], recommend buggy source code [22], and so on. Martie *et al.* [21] apply LDA topic modeling on a large collection of documents containing discussion data from Android developers. A limitation of their work is that although they consider discussion trends over time, they use the same associated keywords throughout the discussion trends. Topic evolution seems to be meaningless to developers if a topic's associated keywords do not change over time. To mitigate this problem, in our work, we extract frequently used keywords associated with each topic for each time window as a topic summary, and refer them as time-sensitives keywords. Thus, our approach is more time specific than theirs. To investigate the impact of changes of technologies on Software Engineering research field, Demeyer *et al.* [14] apply N-gram analysis on the complete corpus of ten years of MSR (Mining Software Repositories) papers, and compute the normalized frequency of keywords for measuring their occurrence over time. Their work inspired us to apply a mining technique on large dataset of bug reports. However, none of the previous work interactively visualizes topic evolution over time, which we do.

In information visualization, researchers have deployed two types of visualization techniques: one is metadata based and the other is content-based. Havre *et al.* [16] use a symmetric river metaphor to represent the thematic variations over time in the context of a time-line and corresponding external events. TIARA [28] conveys far more complex text analysis results than Havre *et al.* by showing detailed thematic content in keywords. Wei *et al.*extract topics from email data and patient records, and generate time-sensitive keywords to represent topic evolution. A part of our work is similar to them [28] in the sense that we adapted their topic evolution approach in this work. However, we have applied this in a new domain, the software bug reports whereas they applied in email datda. Furthermore, our primary focus was visualizing the extractive summaries interactively within the context of the original bug reports, which they did not do.

7. CONCLUSION AND FUTURE WORK

In this paper, we not only visualize the extractive summaries of bug reports but also show the evolution of technical topics over time discussed in those reports. In summary visualization, we apply different colour coding and link the summary sentences to their contexts in the original bug reports interactively so that one can easily analyze the contexts of those statements while reading the summaries. In the visualization of topic evolution for bug reports, we apply LDA, a topic modeling tool, and adapt an existing approach associated with the topic evolution in email documents. In order to evaluate our topic evolution technique, we apply topic modeling on 3914 bug reports collected from *Eclipse-Ant*, and visualize the evolution of several important topics such as plugin support, editor outline, tool launch and configuration, version log, page reference and so on. In order to determine how well the visualized extractive summaries of the bug reports can aid developers in comprehending bug reports, we conduct a task-oriented user study. According to the findings from the study, participants rated relatively higher for visualized summaries than plaintext or non-visualized summaries of bug reports. In future, in order to improve the visualization of bug report summary we plan to add some of the features suggested by the participants such as important keyword highlighting, tool tip option on summary sentences, and so on. Currently, we are working on only software bug reports, but in future we have a plan to visualize not only information extracted from bug reports but also from source code of a software system.

References

[1] *Bugzilla.* http://www.bugzilla.org.

[2] *Eclipse Bug 14890.* https://bugs.eclipse.org/bugs/show_bug.cgi?id=14890.

[3] *Eclipse Bug 62468.* https://bugs.eclipse.org/bugs/show_bug.cgi?id=62468.

[4] *JFreeChart.* http://www.jfree.org/jfreechart/.

[5] *JGibbLDA.* https://jgibblda.sourceforge.net.

[6] *JIRA.* https://www.atlassian.com/software/jira.

[7] *Mann-Whitney U-Test.* http://www.socscistatistics.com/tests/mannwhitney/.

[8] *Stop Words List.* https://code.google.com/p/stop-words/.

[9] *Twitter Sentiment Analysis.* http://help.sentiment140.com/api.

[10] Beineke, P., Hastie, T., Manning, C., & Vaithyanathan, S. 2004. Exploring Sentiment Summarization. *In: Proc. AAAI tech report SS-04-07.*

[11] Dal Sassc, T., & Lanza, M. 2013. A Closer Look at Bugs. *Pages 1–4 of: Proc. VISSOFT.*

[12] D'Ambros, M., & Lanza, M. 2006. Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship. *Pages 10 pp.–238 of: Proc. CSMR.*

[13] D'Ambros, M., Lanza, M., & Pinzger, M. 2007. "A Bug's Life" Visualizing a Bug Database. *Pages 113–120 of: Proc. VISSOFT.*

[14] Demeyer, S., Murgia, A., Wyckmans, K., & Lamkanfi, A. 2013. Happy Birthday! A Trend Analysis on Past MSR Papers. *Pages 353–362 of: Proc. MSR.*

[15] Go, A., Bhayani, R., & Huang, L. 2009. Twitter Sentiment Classification using Distant Supervision. *CS224N Project Report, Stanford*, 1–12.

[16] Havre, S., Hetzler, E., Whitney, P., & Nowell, L. 2002. ThemeRiver: Visualizing Thematic Changes in Large Document Collections. *IEEE Transactions on Visualization and Computer Graphics*, **8**, 9–20.

[17] Hora, A, Anquetil, N., Ducasse, S., Bhatti, M., Couto, C., Valente, M. T., & Martins, J. 2012. Bug Maps: A Tool for the Visual Exploration and Analysis of Bugs. *Pages 523–526 of: Proc. CSMR.*

[18] Lawrence, P., Sergey, B., Rajeev, M., & Terry, W. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66.

[19] Lotufo, R., Malik, Z., & Czarnecki, K. 2012. Modelling the Hurried Bug Report Reading Process to Summarize Bug Reports. *Pages 430–439 of: Proc. ICSM.*

[20] Mani, S., Catherine, R., Sinha, S. V., & Dubey, A. 2012. AUSUM: Approach for Unsupervised Bug Report Summarization. *Pages 11:1–11:11 of: Proc. FSE.*

[21] Martie, L., Palepu, V.K., Sajnani, H., & Lopes, C. 2012. Trendy Bugs: Topic Trends in the Android Bug Reports. *Pages 120–123 of: Proc. MSR.*

[22] Nguyen, A. T., Nguyen, T. T., Al-Kofahi, J., Nguyen, H. V., & Nguyen, T. N. 2011. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. *Pages 263–272 of: Proc. ASE.*

[23] Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., Lo, D., & Sun, C. 2012. Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling. *Pages 70–79 of: Proc. ASE.*

[24] Pingclasai, N., Hata, H., & Matsumoto, K.-I. 2013. Classifying Bug Reports to Bugs and Other Requests Using Topic Modeling. *Pages 13–18 of: Proc. APSEC.*

[25] Rahman, M. M., & Roy, C. K. 2014. An Insight into the Pull Requests of GitHub. *Pages 364–367 of: MSR.*

[26] Rastkar, S, Murphy, C. G., & Murray, G. 2010. Summarizing Software Artifacts: a Case Study of Bug Reports. *Pages 505–514 of: Proc. ICSE.*

[27] Tang, H., Tan, S., & Cheng, X. 2009. A Survey on Sentiment Detection of Reviews. *Expert Systems with Applications*, **36**(7).

[28] Wei, F., Liu, S., Song, Y., Pan, S., Zhou, M. X., Qian, W., Shi, L., Tan, L., & Zhang, Q. 2010. TIARA: A Visual Exploratory Text Analytic System. *Pages 153–162 of: Proc. KDD.*

[29] Yeasmin, S., Roy, C.K., & Schneider, K.A. 2014 (Sept). Interactive Visualization of Bug Reports Using Topic Evolution and Extractive Summaries. *Pages 421–425 of: Proc. ICSME.*