Empir Software Eng (2015) 20:516–548
DOI 10.1007/s10664-014-9311-2

Modelling the 'hurried' bug report reading process to summarize bug reports

Rafael Lotufo · Zeeshan Malik · Krzysztof Czarnecki

Published online: 11 June 2014

Abstract Although bug reports are frequently consulted project assets, they are communication logs, by-products of bug resolution, and not artifacts created with the intent of being easy to follow. To facilitate bug report digestion, we propose a new, unsupervised, bug report summarization approach that estimates the attention a user would hypothetically give to different sentences in a bug report, when pressed with time. We pose three hypotheses on what makes a sentence relevant: discussing frequently discussed topics, being evaluated or assessed by other sentences, and keeping focused on the bug report's title and description. Our results suggest that our hypotheses are valid, since the summaries have as much as 12 % improvement in standard summarization evaluation metrics compared to the previous approach. Our evaluation also asks developers to assess the quality and usefulness of the summaries created for bug reports they have worked on. Feedback from developers not only shows the summaries are useful, but also points out important requirements for this, and any bug summarization approach, and indicates directions for future work.

Keywords Bug reports · Summarization · Natural language processing · Empirical study · Productivity

1 Introduction

Bug reports are valuable assets in software development projects. They serve not only as a communication medium for bug resolution—an activity that accounts for as much as

Communicated by: Massimiliano Di Penta and Jonathan Maletic

R. Lotufo (✉) · Z. Malik · K. Czarnecki
ECE, University of Waterloo, Waterloo, Canada
e-mail: rlotufo@gsd.uwaterloo.ca

Z. Malik
e-mail: zmalik@gsd.uwaterloo.ca

K. Czarnecki
e-mail: kczarnec@gsd.uwaterloo.ca

40 % of software development efforts (Boehm and Basili 2001)—but they are also often consulted, even after the bug has been resolved, by many different parties. A random sample of 200 bug reports we drew from Mozilla, for example, has 275 references to other bugs, indicating the extent to which developers need to refer to other bug reports. Upstream bugs, which are caused by bugs found in software components from different projects or branches, are another common reason for developers to consult bug reports. Since Webkit is the HTML rendering engine adopted by Chrome, many of the bugs from the Chrome project, for example, refer to Webkit bugs. The same is valid for software components that have been repackaged and ported to Debian, Ubuntu, or other operating systems. Duplicate bug report detection is another reason for developers to consult extraneous bug reports, and at least one in five bug reports from the Mozilla, Launchpad and Chrome bug tracking systems are duplicates (Lotufo et al. 2012b).

We argue, therefore, that it is important for bug reports to be easily *digestible*: readers consulting bug reports should easily be able to find the information they seek for. Bug reports, however, are not created with such intent in mind. Collaboration in bug reports develops as a conversation, similar to email threads: participants post messages—commonly referred to as *comments*—as their contributions. A bug report is, therefore, *the result* of the communication that took place in order to address a bug. Unlike a wiki page, it is not collaboratively constructed *with the intention* of being easy to read and comprehend. Since comments have a context set by their previous comments and useful information is spread out throughout the thread, to comprehend a bug report, it is often necessary to read almost the entire conversation. This problem is compounded in open source projects, in which bug reports receive input from many contributors. The Debian community, for example, recognizes the problem and allows users to set a summary of the bug. They claim: "*This is useful in cases where ... the bug has many comments which make it difficult to identify the actual problem*".[1]

Rastkar et al. (2010) recognized the similarity between bug report messages and email threads and used a preexisting summarization technique created to summarize email threads and conversations (Murray 2008). The approach creates an *extractive summary*, which is built by selecting a set of sentences from the original bug report to compose an informative and cohesive summary. The approach uses a logistic regression classifier that is trained on a corpus of manually created reference bug report summaries, also known as *golden summaries*. The results presented by Rastkar et al., however, show that the quality of the generated summaries is sensitive to the training corpus, suggesting that the approach is mostly applicable when trained on a corpus of golden summaries from the target bug tracking system and that the training corpus should be adjusted to reflect the types and nature of bugs as a project evolves. Since creating golden summaries requires significant manual effort and should be done by experts, creating a reasonably-sized training set of golden summaries could be considered as an impediment for the use of such technique.

The objective of this work is, therefore, to *i*) **develop a deeper understanding of the information exchanged in bug reports** and to use this knowledge to **create a general purpose, unsupervised, summarization approach** that should be readily applicable to

[1] http://www.debian.org/Bugs/server-control#summary

virtually any bug tracking system without need for configuration nor of a corpus of manually created golden summaries and ii) generate summaries at least as good as the previous approach, which, from now on, we shall refer to as *email summarizer*.

The summarization approach we propose is based on a hypothetical model of how someone would read a bug report when pressed with time, assuming the reader will have to overlook many sentences and focus on the ones he finds most important. We use the findings of a qualitative *grounded theory* (Strauss and Corbin 2008) investigation on bug reports, to pose three hypotheses on what kinds of sentences a reader would find relevant: sentences that discuss frequently discussed topics, sentences that are evaluated or assessed by other sentences, and sentences that focus on the topics in the bug report's title and description (Section 2). We use this model to rank sentences by their probability of being read and compose the summary with the sentences with the highest probabilities (Section 3).

We create 4 different summarizers, one to test each of our three hypotheses, and one that combines all three summarization hypotheses. We test these summarizers (Section 4) by generating summaries for the same 36 bug reports used by Rastkar et al. (2010) and comparing them with the summaries generated by the email summarizer, which we have also implemented. By comparing standard summary evaluation measures, our results show that the summarizers for each of the hypotheses create competitive or improved summaries, while the combination of these hypotheses creates summaries with as much as 12 % higher evaluation measures.

We also generate summaries for a random selection of bug reports from four different bug tracking systems and conduct a survey asking the developers who worked on these bug reports to assess the quality and usefulness of the summaries (Section 4). Our study attracts the participation of 58 open source developers, who not only validate the quality and usefulness of the summaries, but also point out the most important use cases for bug report summaries and the improvements that our approach and bug report summarizers in general should focus on.

Finally, we present two heuristics to facilitate the use of our summarizers, helping users define the appropriate summary length that will create good quality summaries (Section 5). The first heuristic eliminates the need for the user to specify the desired summary length by creating an alternative input parameter that should better suit users with specific quality constraints. The second approach is able to suggest either a short or a long summary and performs as much as 80 % better than a random predictor.

This work provides five main contributions: (i) a deeper understanding of the information exchanged in bug reports (Section 2.1); (ii) a novel, unsupervised, approach for creating general purpose bug report summaries, that should be readily applicable for any bug tracking system (Section 2.2); (iii) techniques to identify the most important information in bug reports (Section 3); (iv) a survey with 58 open source developers highlighting the use cases for which bug report summaries are most useful for (Section 4); and (v) four directions for future work motivated by our findings: improving the calculation of a sentence's relevance, moving past extractive summaries, creating personalized summaries, and designing interfaces to facilitate bug report navigation based on a summary (Section 6). An earlier version of this work has appeared in Lotufo et al. (2012a). In this version, we broaden the scope of our study and investigate how the summarizers perform when varying the mandatory length input parameter (Section 4.3.1). To facilitate the choice of such input parameter, we propose an alternative parameter that should be more intuitive for some users (Section 5.1) and propose a heuristic to choose a parameter value that will optimize the summary's quality (Section 5.2).

2 Modelling the Bug Report Reading Process

As with most extractive summarization approaches, we want to rank sentences by relevance and select the n most relevant sentences to compose the summary. For our summarization approach, we estimate the relevance of a sentence based on the probability of a reader focusing his attention on that sentence, if the reader were only allowed to focus his attention on a limited number of sentences while skimming through the bug report and still wanted to maximize his knowledge about the bug.

We consider this should resemble, in fact, how users would read a bug report when in a hurry: they would have to skip less important portions of the bug report, moving back and forth to portions that will complement their current understanding, following a single topic or moving their attention to different topics, until they are satisfied with the knowledge they have acquired.

We can model this process with a Markov chain. A Markov chain is a directed graph where nodes represent states, and edges between the nodes model the transition probability between the states. To model the hurried bug report reading process, we can represent each sentence in a bug report as a node in a Markov chain $\mathcal{M}$ where each edge $m_{i,j}$ represents the probability of transitioning from sentence s_i to sentence s_j. Thus, each sentence i will have outgoing edges to all other sentences j, weighted by the probability of sentence j being the next sentence to be read. The relevance of a sentence can then be approximated by calculating the probability distribution of each state in the Markov chain, i.e., the probability of a reader reaching a state if transitioning through the chain according to the transition probabilities of each edge.

As with most models, the Markov chain is only an approximation of our hypothetical bug report reading process. The approximation is given by the fact that, while intuitively the probability of the next sentence does depend on all the previous sentences that were read, Markov chains are memoryless: the probability of the next sentence to be selected will be given only by the current sentence and will not consider the other sentences that have been read.

To complete this model, however, we must estimate the transition probabilities from one sentence to another. Estimating such probabilities requires us to understand what sentences users find important to read, based on the sentence they have just read, i.e., the *links* users are likely to follow from one sentence to another.

2.1 How Knowledge Evolves in Bug Reports

As most problem-solving tasks, bug resolution is a process of reducing the uncertainty about a software issue, until the knowledge that has been gathered is enough to resolve the issue. Comments are used, therefore, to share information that could be used to improve the current knowledge about a bug. Thus, for readers to understand a bug report, it is important that they are able to follow the threads of evolving knowledge.

To gain an insight into how a user might read a bug report and follow the threads of evolving knowledge, we perform a qualitative investigation of the comments in bug reports using *grounded theory*, as proposed by Strauss and Corbin (2008). The question we ask for this investigation is: "*How does knowledge evolve in bug reports?*".

We start by investigating a random sample of 40 bug reports, with at least 10 comments each, from the Chrome, Launchpad, Mozilla, and Debian bug tracking systems. After going through this random sample, we move to a theoretical sampling approach, as recommended by Strauss and Corbin, and stop sampling when new samples do not deepen the understanding of the problem, but fit into our current theory. The final sample of bugs we have used for this study was 15 from Chrome, 13 from Launchpad, 16 from Mozilla, and 11 from Debian.

We find that a bug report serves as a dump of data for an *ad hoc* problem-resolution process. Such data varies from well formatted and comprehensible text and discourse, such as a detailed description of a scenario; to informal conversations, opinions, and ramblings; to the very technical dumps, stack traces or patches that are pasted into bug reports.

In general, we find that comments revolve around three types of information about a bug: *claims*, *hypotheses*, and *proposals*. A claim is a general affirmation made by a participant, such as "*I can reproduce this on 4.11*", or "*The function returns -1 for me*". Participants post hypotheses about, for example, the cause of the bug or a possible solution: "*since I cannot reproduce this on Wheezy, the problem might be caused by the render_screen function*" or "*I think that removing that call should fix the crash*". As for proposals, they are generally used when discussing different approaches to resolve an issue: "*How about using json instead of xml?*".

The information introduced by claims, hypotheses, and proposals evolve over time. Participants frequently post *evaluation* comments that confirm or dispute previous claims, support or reject previous hypotheses, and evaluate previous proposals. Readers, therefore, need to keep track of each of these threads of context. It is only from understanding these threads that a reader will be able to understand, for example, what the outstanding issues preventing a bug's resolution are; what the different verified solutions or workarounds are; and which environments each of these solutions and workarounds are suited for. Lotufo et al. (2012b), in previous quantitative analysis, found that at least 27 % of comments in bug reports result from the evaluation of other comments. This finding is also supported by Breu et al. (2010). Gasser and Ripoche (2003) performed a related study on bug reports and also found that the bug resolution process is much about the 'stabilization' of the knowledge about a bug, which can only be achieved through the evaluation of claims, hypotheses, and proposals.

For bug reports with more than three contributors (which amount to over 46 % of bug reports, with an average of 13 comments each, in a random set of 10000 bug reports from the Mozilla bug tracking systems), in addition to the previous findings, the topic of comments often changes frequently, since each contributor has a different perspective about the bug or purpose for contributing: some contributors might be concerned about convincing others that the priority of the bug is low, for example; others might be trying to present evidence that the issue also occurs in other environments than those already known; and yet other contributors might be trying to coordinate who will be responsible for resolving the bug. As a result, the more contributors discussing a bug report, the more the conversation becomes interwoven and multi-threaded, demanding readers to keep track of multiple contexts and increasing the difficulty of keeping track of the evolving knowledge about the bug.

Thus, these findings from our qualitative investigation suggest that, in order to understand a bug report, users should follow three general heuristics:

(i) users should follow the threads of conversation containing the topics they are interested in, from start to finish, to minimize the probability of missing important information;

title **Crash when opening preview window in Squeeze**

$s_{0,0}$ I'm running XX on Debian Squeeze, and its been running fine since last update.

$s_{1,0}$ The crash occurs when I open up the preview window.

$s_{2,1}$ I could not reproduce this, I'm running on Debian Wheezy, and I do not face this crash when opening the preview window.

$s_{3,2}$ Hi, thanks for submitting this bug.

$s_{4,2}$ I also updated my system today to version 2.28.6.

$s_{5,2}$ Now, when I open up the preview window on 2.28.6 my system crashes, so I did manage to reproduce this problem.

Fig. 1 Bug report for running example

(ii) users should give particular attention to sentences that have been evaluated by other sentences, since they set the context for much of the following comments;

(iii) for users with limited time, in order to focus on the most important points of the bug, users should focus their attention mostly on comments that discuss the problem that was introduced in the bug's title and description and should not follow into parallel topics—a bug's description is commonly shown as the first comment in a bug report and is the bug reporter's characterization of the problem.

2.2 Modelling the Heuristics

We now present how we propose to model each of these heuristics using a Markov chain. Figure 1 presents an example of a bug report that we will use to explain the approach. The

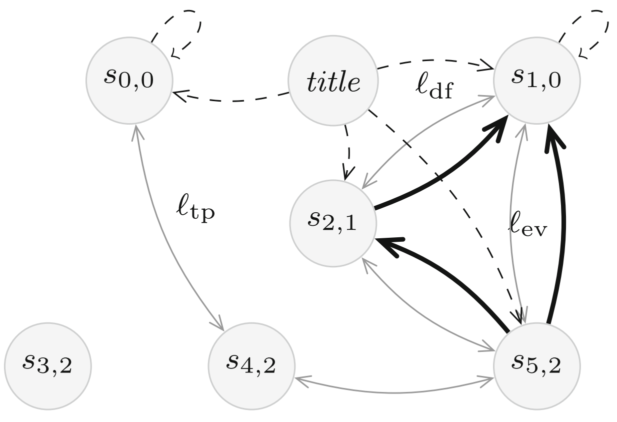


Fig. 2 Graph showing ℓ_{tp} (grey), ℓ_{ev} (*black*), and ℓ_{df} (*dashed*) links for the bug report from the running example

figure shows the bug report title followed by the sentences $s_{i,j}$ in the bug report, where i is the index of the sentence in the bug report and j is the index of the comment the sentence belongs to, considering the bug description as the comment 0. Figure 2 presents the Markov chain for our running example. In the following sections, we will explain the edges shown in the Markov chain.

2.2.1 Topic Similarity

The first heuristic, which assumes that users should follow important threads of conversation from start to finish, implies that after reading a sentence, the most relevant sentences to read next are the sentences that talk about the similar topics as the previous sentence. This can be easily modeled in a Markov chain: the transition probabilities between sentences that talk about the similar topics should be higher than the probability transitions of sentences that do not talk about similar topics. Let us assume that $\ell_{\text{tp}}(s_i, s_j)$ measures how much sentences s_i and s_j talk about the same topics, where i and j are the index of the sentences in the bug report. For now, we consider ℓ_{tp} to be binary: it will return 1 if the two sentences share some topic and 0 otherwise:

$$\ell_{\text{tp}}(s_i, s_j) = \begin{cases} 1 & \text{if topic-sim}(s_i, s_j) > \tau \wedge i \neq j, \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

From the running example, $s_{0,0}$ and $s_{4,2}$ have similar topics, since they talk about updating versions, hence $\ell_{\text{tp}}(s_{0,0}, s_{4,2}) = 1$. In Fig. 2, gray edges represent the symmetric ℓ_{tp} link between two sentences, so we can see a bidirectional edge between sentences $s_{0,0}$ and $s_{4,2}$. Similarly, sentences $s_{1,0}$, $s_{2,1}$, and $s_{5,2}$ all talk about the application crashing when opening the preview window, even if $s_{2,1}$ states that it does not crash when opening the preview window. Therefore, Fig. 2 shows gray bidirectional edges between these sentences.

2.2.2 Evaluation Sentences

The second heuristic suggests that users should pay attention to sentences that have been evaluated by other sentences. For our Markov chain, this means that the transition from a sentence that evaluates another should be higher than from other sentences. Let $\ell_{\text{ev}}(s_i, s_j)$ be a function that indicates if sentence s_i evaluates sentence s_j:

$$\ell_{\text{ev}}(s_i, s_j) = \begin{cases} \ell_{\text{tp}}(s_i, s_j) & \text{if } s_i \text{ evaluates } s_j, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

As it can be seen, we have defined ℓ_{ev} dependent on ℓ_{tp}: if two sentences do not talk about similar topics, one will never evaluate the other. For our example, sentence $s_{2,1}$ and sentence $s_{5,2}$ are evaluation sentences: sentence $s_{2,1}$ evaluates sentence $s_{1,0}$, since the user says that he could not reproduce the bug described in $s_{1,0}$; therefore, $\ell_{\text{ev}}(s_{2,1}, s_{1,0}) = 1$. In Fig. 2, thick black edges represent the ℓ_{ev} relationship, so we can see one of these edges from sentence $s_{2,1}$ to $s_{1,0}$. Similarly, sentence $s_{5,2}$ evaluates sentence $s_{2,1}$, since it disagrees with the statement in $s_{2,1}$ that the bug cannot be reproduced; and evaluates sentence $s_{1,0}$, since it agrees with the statement in $s_{1,0}$ that the bug occurs when opening the preview window. As it can be noted from the definition of ℓ_{tp} and Fig. 2, the ℓ_{tp} link is symmetric. Link ℓ_{ev}, on the other hand, is unidirectional: from the evaluator sentence to the evaluated sentence in a previous comment. Thus, although $s_{4,2}$ and $s_{5,2}$ share a same topic and $s_{5,2}$ is an evaluation sentence, $\ell_{\mathrm{ev}}(s_{4,2}, s_{5,2}) = 0$, since they are in the same comment.

2.2.3 Similarity to Title and Description

In essence we thought that adding the title and removing it later would be as adhoc as the other solution. In addition, we intentionally wanted a strong influence of the title in weighing the other sentences.

Yes, the heuristic we use definitely produces a different result than if we added the title to graph and then removed it at the end. But there are two reasons why we decided on implementing it this way: i)

The final heuristic suggests users to focus on sentences that discuss the problem that was initially reported in the bug title and description. To boost the relevance of sentences with similar topics to the bug description, we can add a link from each sentence in the description to itself. There should be two effects of adding self links to the description: first, the relevance of sentences in the description will be increased; second, as a result of sentences in the description being increased, the relevance of sentences with similar topics to the bug description will also increase. As can be seen in Fig. 2, the two sentences in the description for our example, $s_{0,0}$ and $s_{1,0}$ have edges, of weight 1, to themselves.

We also want to boost the relevance of sentences with similar topics to the bug report title, since previous work has shown that the title can be a very good summary of a bug report (Weiss et al. 2007). To this end, we can artificially include the title as a node in our graph, and add edges from the title t to all sentences s for which $\ell_{\mathrm{tp}}(t, s) > \tau$, for some $\tau > 0$. Figure 2 shows links from the title to sentences $s_{0,0}$, $s_{1,0}$, $s_{2,1}$, and $s_{5,2}$ since they all share some topic with the bug title.

The bug report title is not, however, one of the sentences we are trying to rank. An candidate alternative to workaround this issue would be to include the title as a sentence when computing page rank and then remove the title after computation. Removing the title from the graph after computing page rank, however, would not produce a valid Markov chain, since the sum of the probabilities for each node would not be 1.0. Thus, to add more weight to sentences similar to the bug report's title, instead of adding the title as a node in the Markov chain, we add a link from every sentence s_i to every other sentence s_j that shares topics with the bug report title, such that $i \neq j$.

As a result, to measure the similarity to title and description, we define ℓ_{df} as in (3), where S_D is the set of sentences within the bug description, and t is the bug report title.

$$\ell_{\mathrm{df}}(s_i, s_j) = \begin{cases} 1 & \text{if } i = j \wedge s_i \in S_D, \\ \ell_{\mathrm{tp}}(t, s_j) & \text{otherwise,} \end{cases} \tag{3}$$

2.2.4 Combining Heuristics

Each of our links target different characteristics of sentences in bug reports. Thus, if these heuristics are valid, we hypothesize that the combination of these links should at least produce similar results to the best heuristic and hopefully improve the results of the best heuristic.

Figure 2 shows that we can easily combine ℓ_{tp}, ℓ_{ev}, and ℓ_{df} to rank sentences by summing each of the weights. Since our main interest is to verify if a combination of the links does indeed improve the results of the individual links, we combine them in the most straight-forward way: a self vote of value 1.0 for sentences in the bug report description and a linear combination of the ℓ_{tp}, ℓ_{ev}, and ℓ_{df} for other pairs of sentences:

$$\ell_{\text{all}}(s_i, s_j) = \begin{cases} 1 & \text{if } i = j \wedge s_i \in D, \\ \big[\alpha \ell_{\text{tp}}(s_i, s_j) \\ \quad +\beta \ell_{\text{ev}}(s_i, s_j) \\ \quad +\gamma \ell_{\text{tp}}(t, s_j)\big] \frac{1}{(\alpha+\beta+\gamma)} & \text{otherwise.} \end{cases} \tag{4}$$

Since we want to verify if each of these heuristics are valid and if their combination is also valid, we use the most straightforward parameters for each of the coefficients: 1. We leave the work of optimizing the weights of these parameters for the future. We note that, since our previous definition of ℓ_{df} only depends on the ℓ_{tp} function, which we have already defined, there is no need to redefine it.

2.2.5 Hypotheses

To evaluate if the proposed heuristics are suitable for summarizing bugs reports, we formulate three hypotheses, one for each heuristic.

From Fig. 2, due to the nature of Markov chains, the relevance of each sentence—the probability of a reader reaching each sentence—will be greater *the higher the transition probabilities from other sentences to that sentence weighed by the probabilities of each one of those sentences*. For example, a sentence s_i that has only one topic in common with another sentence s_j, that has a low probability of being read, will also have low probability, since it can only be reached from s_j, even if the probability of transitioning from s_j to s_i is high.

We can now pose the following hypotheses for how to rank sentences by relevance for an extractive summary:

Hypothesis 1 *the relevance of a sentence is higher the more topics it shares with other relevant sentences;*

Hypothesis 2 *the relevance of a sentence is higher the more it is evaluated by other relevant sentences;*

Hypothesis 3 *the relevance of a sentence is higher the more topics it shares with the bug title and description.*

2.3 Calculating Probability Distribution

The graph in Fig. 2 is not yet a Markov chain, since until now, the weights for the edges are not probabilities, but simple weights measured by the link functions ℓ_{tp}, ℓ_{ev}, and ℓ_{df}. To transform the graph into a proper Markov chain, we must calculate the transition probability for each edge. From the weights for the links $\ell(s_i, s_j)$ between the sentences, calculating the transition probability is trivial: for every outgoing edge in a node, we divide its weight by the sum of the weights for all outgoing edges for that same node, as shown in (5). As a result, the sum of the weights (probabilities) for the outgoing edges of each node will be 1.

$$m_{i,j} = \frac{\ell(i, j)}{\sum_{\forall k} \ell(i, k)} \tag{5}$$

Thus, considering only the ℓ_{tp} links, the adjacency matrix for a graph G for the running example, and the resulting Markov chain $\mathcal{M}$ would be the following:

$$G = \begin{Bmatrix} 0&0&0&0&1&0 \\ 0&0&1&0&0&1 \\ 0&1&0&0&0&1 \\ 0&0&0&0&0&0 \\ 1&0&0&0&0&1 \\ 0&1&1&0&1&0 \end{Bmatrix} \qquad \mathcal{M} = \begin{Bmatrix} 0&0&0&0&1&0 \\ 0&0&0.5&0&0&0.5 \\ 0&0.5&0&0&0&0.5 \\ 0&0&0&0&0&0 \\ 0.5&0&0&0&0&0.5 \\ 0&0.33&0.33&0&0.33&0 \end{Bmatrix} \tag{6}$$

where the order of nodes in the matrix are given by the index of the sentence in the bug report. Elements $g_{4,0}$ and $g_{0,4}$ have value 1, for example, since they represent the ℓ_{tp} links between sentences $s_{0,0}$ to $s_{4,2}$.

2.3.1 *Using PageRank to Calculate Sentence Relevance*

Brin and Page (1998) developed PageRank to rank web pages by relevance using a very similar model. They estimated the relevance of a web page as the probability of a user reaching that page for a user who surfs the web randomly following hyperlinks from one web page to another, like a random surfer. PageRank takes as input a graph G, where web pages are nodes and a link from page i to page j is modeled as a directed edge from i to j with weight $l(i, j) = 1$. PageRank then calculates the Markov chain $\mathcal{M}$ for the random surfer model using (5) and outputs a probability distribution $\mathbf{R}$, where each r_i is the probability of a user eventually reaching web page i after a large number of clicks.

Given the Markov chain $\mathcal{M}$, the probability distribution $\mathbf{R}$ for the elements in $\mathcal{M}$ is its principal eigenvector, such that $\mathbf{R} = \mathcal{M}\mathbf{R}$. By the Perron-Frobenius theorem, if $\mathcal{M}$ is an irreducible and aperiodic stochastic matrix, we can use the iterative power method to compute $\mathbf{R}$, since the Markov chain is guaranteed to converge to a unique stationary distribution.

An irreducible Markov chain is one in which all states are reachable from any other states, e.g., all states have at least one transition to it with probability > 0. An aperiodic Markov chain is one in which all states are aperiodic: the minimum common divisor of the number of transitions required to return to any state is 1.

We cannot guarantee, however, that the Markov chain for the web is irreducible and aperiodic. In fact, it is known that it is not: there are many web pages that are not linked to from any other web pages. Similarly for sentences in a bug report, we cannot guarantee there will not be any sentence that does not share any topic with any other sentence.

To transform $\mathcal{M}$ into an irreducible and aperiodic Markov chain $\widehat{\mathcal{M}}$, Brin and Page consider that with probability δ a user will not follow any hyperlink but will randomly go to *any* web page in the Internet. Since any web page is now reachable by any other web page in one transition with probability δ, the chain is now irreducible and aperiodic. The formula for PageRank, in matrix form, is shown below, where $\mathbf{U}_{n\times n}$ is a square matrix of ones, and $\mathcal{M}$ is the Markov matrix.

$$\mathbf{R} = \left[\frac{1-\delta}{n}\mathbf{U} + \delta\mathcal{M}\right]\mathbf{R} = \widehat{\mathcal{M}}\mathbf{R} \tag{7}$$

Now that $\widehat{\mathcal{M}}$ is aperiodic and irreducible and the convergence of $\mathbf{R}$ is guaranteed, we use the *iterative power method* to calculate the principal eigenvector of $\widehat{\mathcal{M}}$. The power method calculates $\mathbf{R}$ starting as a uniform distribution $\mathbf{R} = \frac{1}{n}\mathbf{1}$, and updates $\mathbf{R}$ at each step as $\mathbf{R}' = \widehat{\mathcal{M}}\mathbf{R}$. The algorithm stops when $|\mathbf{R} - \mathbf{R}'| < \epsilon$, which, for aperiodic and irreducible stochastic matrices, is guaranteed to occur for $\epsilon > 0$.

Given the similarity of the random surfer model and the bug report reading model that we have proposed in Section 2, applying PageRank to calculate the probabilities of sentences being read requires only that we identify the links, i.e, the transitions, between sentences in a bug report. We can then derive the Markov chain $\mathcal{M}$ using (5) and calculate the probability distribution $\mathbf{R}$, just as in PageRank.

The only change our original model suffers when using PageRank is that now we must consider that, with probability δ, a user might jump to any sentence in a bug report without following our links—we use $\delta = 0.85$ just as recommended by Brin and Page in PageRank. We can use (5) directly to calculate the Markov chain, which is applicable even if $\ell(s_i, s_j)$ returns values different from 0 and 1 and instead returns any value ≥ 0 as the weight of links. This will be the case when we rank sentences considering both Hypotheses 1 and 2, for example, by combining ℓ_{tp} and ℓ_{ev} as $\ell(s_i, s_j) = \ell_{\text{tp}}(s_i, s_j) + \ell_{\text{ev}}(s_i, s_j)$ and when ℓ_{tp} and ℓ_{ev} measure the *strength* of these links.

For the running example, considering only the ℓ_{tp} links, transforming G into the stochastic matrix $\mathcal{M}$ by dividing each row by the sum of the row, as shown in (5), results in (6). After making it irreducible and aperiodic by multiplying δ and then adding $\frac{1-\delta}{n}\mathbf{U}$, as shown in (7), and solving for $\mathbf{R}$ using the iterative power method, we get the following probabilities for the sentences in our example: $[0.11, 0.18, 0.18, 0.02, 0.20, 0.27]$, effectively ranking sentences as $[s_{5,2}, s_{4,2}, s_{1,0}, s_{2,1}, s_{0,0}, s_{3,2}]$.

3 Estimating Transition Probabilities Between Sentences

We have shown how to model the hurried bug report reading process and estimate sentence relevance using a Markov chain, assuming we can measure ℓ_{tp}, ℓ_{ev}, and ℓ_{df}. The following sections details how we use natural language processing to measure how much two sentences talk about the same topics and to identify sentences that evaluate other sentences and quantify the weights for links ℓ_{tp}, ℓ_{ev}, and ℓ_{df}.

3.1 Measuring ℓ_{tp}

There exists much work on measuring how much two documents discuss the same topics. Most of these, first identify the topics contained within documents and then measure topic similarity by considering how much topic overlap there exists between the documents. Sun et al. (2007), for example, measured the changes in mutual information from one chunk of a document to another to detect topics. Latent Dirichlet Allocation (LDA) (Blei et al. 2003) and Probabilistic Latent Semantic Analysis (PLSA) (Hofmann 1999), on the other hand, identify topics using word co-occurrence knowledge extracted from documents.

While arguably these approaches are state-of-the-art in identifying topics, they are not lightweight and generally require the tuning of several different parameters, most importantly, the number of topics to be identified. Since we aim for a solution that does not need such parametrization, we choose a more direct and lightweight approach to measure topic similarity: we will consider that sentences that talk about similar topics should have many common words. We will approximate, therefore, topic similarity by lexical similarity.

While there are many textual similarity metrics, such as Levenstein edit distance and Euclidean distance, the *cosine similarity* function is one that has shown consistent results in measuring the similarity of content, and is used, for example, to classify and cluster documents by author, topics, and writing style (Büttcher et al. 2010). The cosine similarity is defined as below, where x and y are the vectors of term frequency for a sentence.

$$\text{cosine} - \text{sim}(x, y) = \frac{x \cdot y}{|x|.|y|} \tag{8}$$

As is commonly done when measuring textual similarity, we scale the term frequency (tf) by the inverse document frequency (idf) of the term, diminishing the importance of terms that occur in most documents, since they do not help in differentiating two documents. The term frequencies in the vector for each sentence, scaled by the inverse document frequency is calculated as shown below, where $n_{t,s}$ is the number of times term t occurs in s, N is the total number of sentences, and n_t is the number of sentences that contain term t.

$$\text{tf} - \text{idf}(t, s) = n_{t,s} \log \frac{N}{n_t} \tag{9}$$

Before building the vectors for the sentences using tf − idf, we must first tokenize the text into its terms. Based on our previous experience with tokenizing text from bug reports, we tokenize the sentences using the following regular expression: '`[\w-]+(\.[\w-]+)*`', which should correctly identify words, while preserving most function and variable names, urls, and software version numbers. After tokenization, we move all characters to lowercase and stem the tokens using the standard Porter stemmer (Porter and et al 1980). We can now redefine ℓ_{tp} as:

$$\ell_{\text{tp}}(s_i, s_j) = \begin{cases} \text{cosine} - \text{sim}(s_i, s_j) & \text{if } i \neq j, \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

3.2 Measuring ℓ_{ev}

The relations of evaluation between sentences we are interested in are those where a sentence evaluates or verifies the validity of the content of another sentence. From our example, $s_{2,1}$ fits this relation, as it suggests that the content of $s_{1,0}$ is not valid, since its author could not reproduce the crash. Similarly, $s_{5,2}$ evaluates both $s_{1,0}$ and $s_{2,1}$, since it says that the bug is reproducible, confirming the content of $s_{1,0}$ and disconfirms the content of $s_{2,1}$.

Although we are not aware of prior work that tries to identify evaluation relations between sentences in a bug report, *polarity detection* through *sentiment analysis* might be a good approximation of this relation. Polarity detection has previously been used to detect the polarity of reviews of movies (Beineke et al. 2004), of products, politicians, and almost anything (Tang et al. 2009). We consider polarity detection might be a reasonable approximation, since, in general, it first filters evaluation sentences, and then tries to detect if the evaluation is positive or negative. For our purposes, we consider a sentence is an evaluation sentence if its polarity is different from neutral.

The best results for polarity detection have been achieved using classifiers such as support vector machines that are trained on a corpus of texts that have had its polarity previously annotated. Since we are looking for an approach that is completely unsupervised, however, having to manually classify the polarity of sentences of a bug report would not suit here. We use, therefore, the same approach as Go and Bhayani (2009) who created a large training corpus of sentences for polarity prediction by using the *emoticons* present in Twitter messages to automatically infer the polarity of each sentence. We use a training set composed of 800,000 Twitter messages with positive polarity, 800,000 with neutral polarity, and 800,000 with negative polarity and use the resulting classifier to predict the polarity of sentences in bug reports—we include into the training set sentences with neutral polarity, since we also need to identify the sentences that are not evaluation sentences. This classifier uses a linear support vector machine in which the feature vector for a sentence represents the presence or absence of a word in the comment. We leave the reader to consult Go and Bhayani (2009) report for further details on this approach.

Simply identifying evaluation sentences is not enough, however, since we want to identify an evaluation *link* from one sentence to another. Our definition of the evaluation link from Section 2.2.2, however, hints to a solution to identify such relation: the evaluation link requires first that two sentences have similar topics. As such, just as how cosine − sim measures not *if*, but *how much*, two sentences share a topic, here we also propose to quantify *the strength* of the evaluation link between two sentences. Furthermore, sentences in comment i can only be evaluated by sentences from comments posted after comment i.

We measure the strength of the evaluation link from s_i to s_j as the strength of the ℓ_{tp} link between the two sentences if s_i is an evaluative sentence or is a sentence within a comment that contains an evaluative sentence. We can now redefine ℓ_{ev} as:

$$\ell_{\text{ev}}(s_i, s_j) = \begin{cases} \ell_{\text{tp}}(s_i, s_j)\frac{\left(p_{\mathcal{S}}(s_i)+p_{\mathcal{C}}(s_i)\right)}{2} & \text{if } \mathrm{c}(s_i) > \mathrm{c}(s_j), \\ 0 & \text{otherwise,} \end{cases} \tag{11}$$

where $p_{\mathcal{S}}(s)$ returns 1 if s has polarity and 0 otherwise, $p_{\mathcal{C}}(s)$ returns 1 if s is contained in a comment that contains a sentence that has polarity and zero otherwise, and $\mathrm{c}(s)$ returns the index of the comment that contains sentence s.

3.3 Chunking Comments into Sentences

Our extractive summarization approach works with sentences as the minimal chunks of text to be extracted into a summary. Bug reports, however, are not segmented into sentences. In fact, segmenting comments into sentences is a challenging problem itself, since text in bug reports is very informal and often contains source code, stack traces, logs, and enumerations. Using a traditional sentence chunker that uses a '.' character to identify sentence boundaries does not produce good results. While Bettenburg et al. (2008b) parsed bug report comments to collect structured information, such as stack traces and patches, they did not work on sentence chunking, but acknowledged that it is a non-trivial problem.

We chunk comments into sentences using the following heuristics: (i) each item from an enumeration is a sentence; (ii) we use '.', ';', '?', '!' as sentence delimiters, except when '.' is used in version strings, URLs, code snippets, and abbreviations; (iii) if a line of text has a line break before 80 characters, we consider that the line break ends a sentence. As a result of these heuristics, since generally each line in a stack trace or source code snippet has less than 80 characters, each line in a stack trace or source code is often a sentence itself.

4 Evaluation

4.1 Methodology

Our evaluation has two parts. For the first part, we implement three different summarizers, one for each link function ℓ_{tp}, ℓ_{ev}, and ℓ_{df}, to evaluate each one of our hypotheses. We will consider our hypotheses to be valid if our summarizers produce summaries that have competitive or improved evaluation measures compared to the summaries created by the email summarizer (Rastkar et al. 2010), which we have implemented to the best of our knowledge. We also test a summarizer using ℓ_{all} to assess if the combination of the links yields improved summaries.

The corpus we use for this evaluation is the corpus created by Rastkar et al. (2010), which consists of 36 bug reports, from four different open-source projects: Eclipse, Gnome, Mozilla, and KDE, each with three reference *golden summaries* created by humans. We use these reference summaries to compare against the generated summaries. For this evaluation, and as was done by Rastkar et al., we generate summaries by selecting sentences until the summary reaches a predefined percentage of the original bug report's length, in number of words. For completeness, we evaluate the performance of the summarizers when generating summaries of different lengths, from 15 to 70 % of the length of original bug report in number of words.

The bugs chosen for annotation by Rastkar et al. have mostly conversational content. The authors intentionally avoided bug reports containing structured technical data, such as stack traces and code, since they claim that this content is not typically read by developers. For the second part of our evaluation, however, we do not restrict our tests to bug reports with conversational content, and have selected bug reports randomly selected from four other open-source projects: Debian, Launchpad, Mozilla, and Chrome.

For this second part of our evaluation, we use the ℓ_{all} summarizer to generate summaries of length 25 % of the original bug report for a random set of bug reports from the four open-source projects above and invite the developers who contributed to these bug reports to assess the quality and usefulness of the summaries. From the 250 invitations we sent out, we received a response from 58 developers, each one evaluating a different bug report: 22 from Debian, 14 from Mozilla, 13 from Ubuntu, and 9 from Chrome. We present the developers with each summary in two formats: *condensed* and *interlaced*. Figures 3 and 4 show samples of the interlaced and condensed summaries for the bug report in the running example. The interlaced format presents the complete bug report content, with the extracted sentences shown highlighted out from the other sentences. The condensed format shows only the extracted sentences. We ask the developers to: (i) assess the quality of the summary by indicating the mistakes made by the summarizer, i.e., the sentences that should have been extracted but were not and the sentences that were extracted but are not so relevant; (ii) explain if they preferred the condensed or interlaced format for reading a bug report summary; (iii) explain what are the most important types of information that a summary should contain; and (iv) indicate, using a Likert scale, what are the most important use cases for such summaries.

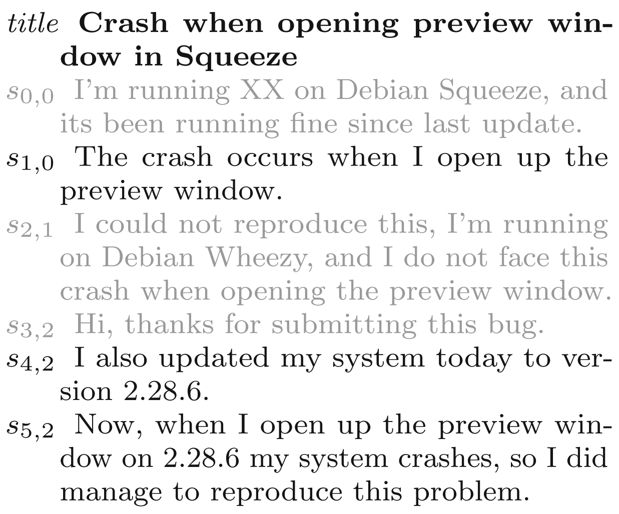
4.2 Evaluation Metrics

To assess our hypotheses, we use the following established metrics for evaluating summaries:

Precision and Recall We measure precision and recall for the summaries, considering a *master golden summary* G^* composed of the sentences that are present in the majority of the golden summaries. For the corpus created by Rastkar, the master golden summary is composed of the sentences that are present in at least two out of three golden summaries.

$$\text{precision}(S) = \frac{|\{S \cap G^*\}|}{|S|} \qquad \text{recall}(S) = \frac{|\{S \cap G^*\}|}{|G^*|} \tag{12}$$

$$\text{f} - \text{score}(S) = \frac{2 * \text{precision}(S) * \text{recall}(S)}{\text{precision}(S) + \text{recall}(S)} \tag{13}$$



title **Crash when opening preview window in Squeeze**

$s_{0,0}$ I'm running XX on Debian Squeeze, and its been running fine since last update.

$s_{1,0}$ The crash occurs when I open up the preview window.

$s_{2,1}$ I could not reproduce this, I'm running on Debian Wheezy, and I do not face this crash when opening the preview window.

$s_{3,2}$ Hi, thanks for submitting this bug.

$s_{4,2}$ I also updated my system today to version 2.28.6.

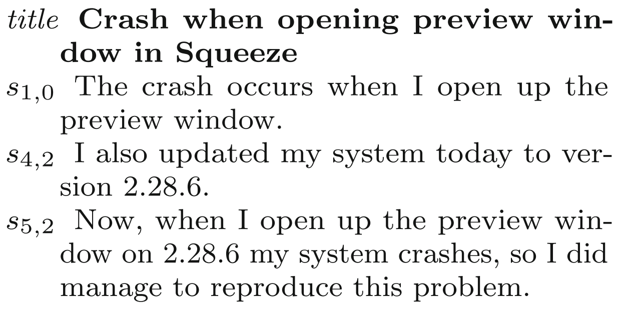
$s_{5,2}$ Now, when I open up the preview window on 2.28.6 my system crashes, so I did manage to reproduce this problem.

Fig. 3 Interlaced summary view for the running example

Precision then measures the percentage of sentences in a summary that is also present in G^*, while recall measures the percentage of sentences in G^* that are present in the summary being evaluated. The f-score is the harmonic mean of the precision and recall rates, given by (13), which, differently from the arithmetic mean, will give a higher weight to the lower values.

Precision and recall, however, measure the quality of a summary against a master golden summary which is artificially composed by the sentences present in at least half of the golden summaries. Thus, it does not mean that such a master golden summary is necessarily a good one, since different golden summaries can provide all the relevant information with different extracted sentences (Nenkova et al. 2007).

Pyramid Score To circumvent the issues of recall and precision, Nenkova et al. (2007) proposed the *pyramid score*, an evaluation metric that should better measure the quality of an extractive summary based on a set of golden summaries created by several annotators. When evaluating a summary composed of *n* sentences, pyramid score is the sum of the number of golden summaries that contain each of the *n* sentences from the evaluated summary, divided



title **Crash when opening preview window in Squeeze**

$s_{1,0}$ The crash occurs when I open up the preview window.

$s_{4,2}$ I also updated my system today to version 2.28.6.

$s_{5,2}$ Now, when I open up the preview window on 2.28.6 my system crashes, so I did manage to reproduce this problem.

Fig. 4 Condensed summary view for the running example

by the sum of the number of golden summaries that contain the n sentences that are present in the highest number of the golden summaries. Pyramid score is, therefore, a recall-related evaluation metric for a summary, that measures the quality of a summary against the best summary of the same length.

The formula for the calculation of pyramid score is shown below, where S is a summary, $s \in S$ are the sentences in summary S, $\mathcal{G}$ is the set of all golden summaries G, and $\mathcal{G}^{\text{top}}_{|S|}$ is the set of size $|S|$ of sentences that are present in highest number of the golden summaries:

$$\text{pyramid}(S) = \frac{\sum_{s \in S} \left|\{G \in \mathcal{G} : s \in G\}\right|}{\sum_{s \in \mathcal{G}^{\text{top}}_{|S|}} \left|\{G \in \mathcal{G} : s \in G\}\right|} \tag{14}$$

Nenkova also defines pyramid precision and pyramid recall. Pyramid precision calculates the percentage of sentences in a summary that are present in at least one golden summary. Pyramid recall calculates the percentage of sentences present in any one of the golden summaries that are present in the summary being evaluated. In effect, these are the precision and recall as defined in (12), with G^* being composed of sentences that are present in any of the golden summaries.

4.3 Hypotheses Tests

To test our hypotheses, we generate summaries for the bug reports in the Rastkar corpus and compare their quality with the summaries generated by the email summarizer. Like Rastkar, we start by generating summaries of 25 % of the length of original bug report in number of words. Figure 5 presents, for each evaluation metric, the averages weighed by the number of sentences in a bug report. For each of the five summarizers we evaluate: ℓ_{tp} summarizer, ℓ_{ev} summarizer, ℓ_{df} summarizer, ℓ_{all} summarizer, and email summarizer. In the spider chart, the maximum value in each of the evaluation metrics' axes is the maximum for that metric, for all of the summarizers and each tick in the axis corresponds to a difference of 0.05. While the spider chart facilitates viewing the differences between the metrics between the

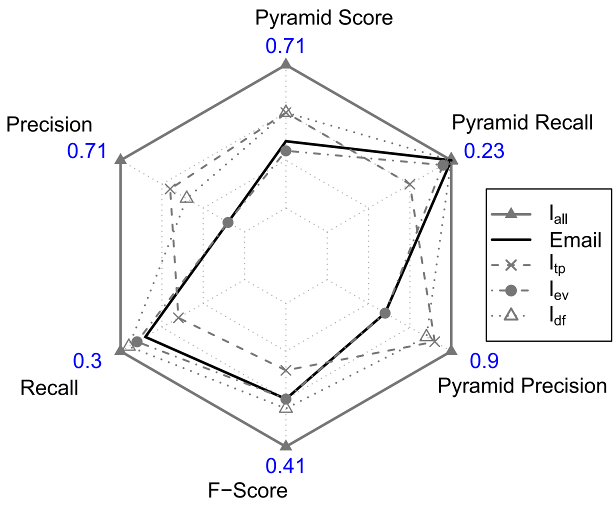


Fig. 5 Evaluation measures for ℓ_{tp}, ℓ_{ev}, ℓ_{df}, ℓ_{all}, and email summarizers for summaries of length 25 %. The maximum value for each axis is the maximum for each metric while each tick in the axis corresponds to 0.05

Table 1 Comparison of evaluation measures for ℓ_{tp}, ℓ_{ev}, ℓ_{df}, ℓ_{all}, and email summarizers for summaries of length 25 %

	ℓ_{tp}	ℓ_{ev}	ℓ_{df}	ℓ_{all}	email
Precision	0.65	0.58	0.63	0.71	0.58
Recall	0.23	0.28	0.29	0.30	0.27
F-Score	0.33	0.36	0.37	0.41	0.36
Pyramid Precision	0.88	0.82	0.87	0.90	0.82
Pyramid Recall	0.18	0.22	0.23	0.23	0.23
Pyramid Score	0.66	0.62	0.66	0.71	0.63

summarizers, we present in Table 1 the same information in tabular format, showing exact values for each of the metrics.

The results show that ℓ_{tp} summarizer has a slightly higher precision and pyramid precision than the email summarizer, but has less recall and pyramid recall. The non-parametric Mann-Whitney U test supports that these distributions are indeed different, with p-value $<$ 0.05 for precision and recall, and p-value < 0.01 for pyramid precision and recall. The chart also shows that the ℓ_{ev} summarizer has similar evaluation results compared to the email summarizer, for summaries of length 25 %. For ℓ_{df}, the chart shows that it has slightly better values for all evaluation metrics compared to the email summarizer, except for pyramid recall. The Mann-Whitney U test, however, supports that, for ℓ_{df}, only the pyramid precision measure is better than email summarizer, with p-value < 0.01.

We can also use Fig. 5 to compare each of our three individual summarizers—ℓ_{tp}, ℓ_{ev}, and ℓ_{df}—amongst themselves. This comparison shows that ℓ_{tp} has significantly less recall than the other summarizers, while ℓ_{ev} has significantly less precision than the others. We find that ℓ_{tp} has such low recall because ℓ_{tp} prefers longer sentences than the other summarizers. Thus, since we take sentences until we reach 25 % of the number of words in the original bug report, it extracts less sentences than the other summarizers.

The results we have presented show that all of our individual summarizers are at least competitive with the email summarizer. Furthermore, their combination as ℓ_{all}, has an improvement of 12 % in precision, 8 % in pyramid precision, and 8 % in pyramid score, confirmed by the Mann-Whitney U test with p-value < 0.01. These results indicate that Hypotheses 1, 2, and 3 have a high likelihood to be valid: relevant sentences for a bug report summary are those that discuss topics that are frequently discussed; those that are evaluated by other sentences; and those that do not deviate from the problems as described in the bug report title and description.

It is important to note that all of the summarizers we evaluate here have reasonable precision but quite low recall. One of the main causes of the low recall is that most of expert summaries for the Rastkar corpus are larger than 25 % of the original bug report length. We will discuss how the summarizers perform for different summary lengths next and elaborate on how to aid the user in deciding on a target summary length in Section 5.

4.3.1 Varying Summary Target Length

For most summarization approaches, the length of the resulting summaries are decided beforehand, by the user, usually through an input parameter, either in number or percentage of characters, words, sentences, or paragraphs (Lloret and Palomar 2012).

Rastkar et al. (2010), for example, ranked sentences by their scores and includes the highest scoring sentences until the summary reaches 25 % the original bug report length in words.

We have shown in the previous section that ℓ_{tp}, ℓ_{ev}, ℓ_{df}, and ℓ_{all} produce good quality summaries of length 25 % of the original bug report and that ℓ_{all} produces significant improvements over the email summarization approach. For a more complete evaluation, we now present the accuracy of these summarizers when generating summaries of different lengths.

Figure 6 presents the precision, recall, f-score, pyramid precision, pyramid recall, and pyramid score for ℓ_{tp}, ℓ_{ev}, ℓ_{df}, ℓ_{all}, and the email summarizer, for lengths varying from 15 to 70 % of the original bug reports. Similar to Fig. results-radar, for all metrics, except recall and pyramid recall, the ℓ_{all} summarizer has clearly better values than all other summarizers, for all summary lengths. For recall and pyramid recall, however, the figure shows

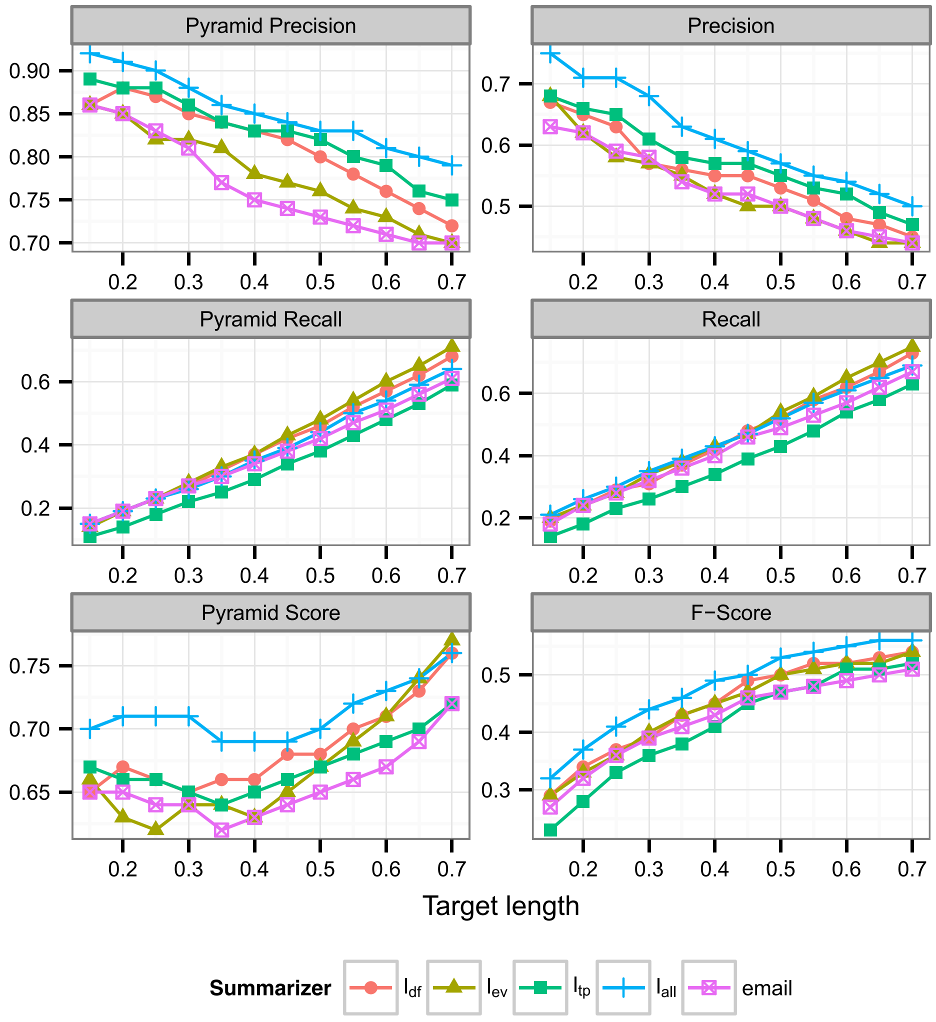


Fig. 6 Evaluation measures for ℓ_{tp}, ℓ_{ev}, ℓ_{df}, ℓ_{all}, and the email summarizer for summary lengths varying from 15 to 70 % of original bug report word count

that ℓ_{all} has similar values to the other summarizers for shorter summary lengths, but the values decrease, compared to the other summarizers, excluding ℓ_{tp}, as the summary length increases. The reason why ℓ_{all} and ℓ_{tp} have lower recall even while having higher precision over the other summarizers is because these two summarizers select longer sentences than the other summarizers. Since we stop including sentences when we reach a predetermined word count, a summarizer that prefers longer sentences will have included fewer sentences than summarizers that prefer shorter sentences.

Figure 6 also shows that, overall, summary quality does not decay as we include more sentences. Although precision and pyramid precision does decrease consistently, the pyramid score remains almost constant and the f-score increases, indicating that the recall and pyramid recall increases at a higher rate than the decrease in precision.

4.4 Evaluation with Developers

The evaluation with the developers from the Debian, Launchpad, Mozilla, and Chrome bug tracking systems, in which we presented developers with a summary of length 25 %, in number of words, of a bug report they had previously work on, was very insightful. From the passion of the responses we received from developers, they seemed genuinely interested in bug report summaries. This feeling is corroborated by the results of our survey, in which more than 80 % of developers stated that bug report summaries would be at least *very* useful—out of a scale of not useful, somewhat useful, useful, very useful, and extremely useful—when (i) looking for a solution or workaround for a bug; (ii) searching for similar or duplicate bugs; (iii) trying to understand the status of the bug and its open issues; and (iv) when consulting bugs for prioritization, triaging, or closing out old bugs.

When asked about the most important kind of information that needs to be present in a bug report summary, developers repeatedly affirmed that summaries should focus on showing information (i) about the current status and the reason for such state; (ii) about solutions or workarounds to the bug and the environments each remedy is applicable to; and (iii) about the consensus on diagnostic information, such as the agreed steps and environment settings to reproduce the bug. Developers also stressed the importance of being able to recognize the different types of structured information in bug reports: stack traces, code snippets, commands and their results, and enumerations such as steps to reproduce. A qualitative analysis of the mistakes made by summarizer and pointed out by the developers, indicates that sentences, as chunked by the procedure explained in Section 3.3, might not be the most appropriate way to chunk content in bug reports, since the resulting summaries often contain, for example, only some of the items of an enumeration and the result of a command but not the command itself.

When evaluating the quality of the summaries presented to the developers, we asked them to mark the mistakes made by the summarizer: sentences that were included in the summary but should not have been (false positives) and sentences that were not included in the summary but should have been (false negatives)—the sentences that were selected by the summarizer to be included in the summary and were not marked by the developers were considered true positives; the sentences that the summarizer did not select to be in the summary and were not marked by the developers were considered to be true negatives. Such marking made by the developers allows us to quantify the quality of the summaries by calculating precision and recall. Since these bug reports only have one reference golden summary, pyramid score is not applicable and the precision and recall, and pyramid precision and pyramid recall will have the same values. The results for the 58 bug reports assessed by developers, shown in Table 2, indicate that, in average, the summaries include half of the relevant information, with 60 % of the sentences in the summaries being relevant ones. These results are promising, since they are substantially better than the results for the Rastkar corpus, showing an improvement 20 % in recall with a decrease in precision of only 10 %. This indicates that the ℓ_{tp}, ℓ_{ev}, ℓ_{df}, and ℓ_{all} summarizers have not been tailored to a particular subset of bug reports and are pretty general, achieving our objective of creating an unsupervised bug report summarization approach that is readily applicable to any bug tracking system.

When developers were asked if they preferred the condensed or interlaced summary formats, responses were mixed: 56 % preferred condensed, while 46 % preferred interlaced, a non-significant difference. We did, however, find a consensus on the advantages of each one, which can be summarized by the two following responses:

> "I would never trust an automated summary, and would always need to refer to the original. By highlighting the important sentences, it allows me to 'speed read' a long report with many comments and status updates."

> "Interlaced works when there aren't pages of irrelevant data. For a bug with *lots* of comments, a condensed view would help. Personally, I use a greasemonkey script that highlights comments from people that are likely to be providing useful information."

Users were, thus, generally skeptical that an automated system would generate a perfect summary, and would need to be able to refer to the non-relevant sentences when needed. An optimal user interface for speed-reading a bug report would be one that would allow users to easily skip irrelevant comments but at the same time be able to quickly scan them looking for relevant sentences that the summarizer missed.

5 What Length Should my Summary be?

Our evaluation has shown how precision and recall vary for different target summary lengths. In practice, however, this and most summarization mechanisms require the user to select the desired target summary length (Lloret and Palomar 2012) as an input parameter. Unfortunately, it might not always be clear what a good target summary length should be for a particular bug report. Should one consider that 25 % of the length of the original bug report is a good summary length for all bugs?

The summaries created by experts for the bug reports in Rastkar's corpus, for example, have a wide variety of lengths, ranging from 18 to 69 %, with a mean and median of approximately 47 %. It seems, therefore, safe to assume that different bugs, with different

Table 2 Precision and recall for developer evaluation

	Debian		Mozilla		Launchpad		Chrome		All	
	ℓ_{all}	email	ℓ_{all}	email	ℓ_{all}	email	ℓ_{all}	email	ℓ_{all}	email
Precision	0.49	0.41	0.77	0.58	0.56	0.62	0.69	0.62	0.59	0.52
Recall	0.45	0.40	0.60	0.34	0.47	0.46	0.61	0.53	0.51	0.42
F-score	0.44	0.34	0.62	0.39	0.50	0.51	0.59	0.50	0.52	0.41

The high quality of summaries for this corpus suggests that the summarizers we propose are able to produce quality summaries for bug reports in any bug tracking system informational content, density and different flow of threads and topics will probably have different optimal summary lengths. The question we now pose is: *how can we assist the user in choosing the length of the target summary when using our summarizers?*

For this part of the work, we focus mainly on the ℓ_{all} summarizer, as we did in our developer evaluation, since this is the best summarizer of all, as our evaluation has shown. We will present, nevertheless, details about summary lengths for the remaining summarizers, mostly for the purpose of understanding how each part of the ℓ_{all} summarizer works, and how each different heuristic contributes to its final result. We hope this will help others in improving each of the heuristics.

5.1 Sentence Relevance Threshold vs. Maximum Length

One way to facilitate the use of the summarizers is to substitute the target summary length parameter with an alternative parameter that would be easier for users to decide on a value for. An alternative parameter to maximum summary length is the *minimum relevance threshold*. Instead of including the top ranking sentences until a certain summary length is achieved, one could include into the summary only the sentences that cross a certain relevance threshold. The length of the summary would be determined solely by how many sentences are found to be relevant, thus allowing different bug reports to have different lengths for the same value of minimum relevance threshold. Creating a summary with only the sentences above a certain relevance threshold is straightforward. Since the sum of the sentence probabilities resulting from PageRank have sum of 1, we normalize the probabilities by dividing the probability of each sentence by the maximum sentence probability in the bug report. Such normalization would transform all sentence relevance scores in values from 0 to 1, now allowing us to specify that we want to include in the summary only sentences with relevance at least $\tau\%$ of the maximum relevance.

When using such scheme, increasing the threshold τ will produce shorter summaries, while decreasing the threshold will produce longer summaries. Figure 7 shows how the distribution of resulting summary lengths per bug report changes as we vary the minimum relevance threshold, for each of the different summarizers for the Rastkar bug corpus as well as the developer bug corpus. What is notable from the figure is that, while there is a large variation in summary lengths for the same relevance threshold, each summarization approach has a characteristic curve for how summary length varies as we increase the relevance threshold. For the ℓ_{df} and ℓ_{all} summarizers, the curve shows a steep decrease in summary lengths for low values of the threshold, whereas the ℓ_{tp} summarizer has a quasi-linear decay in summary length as the threshold increases. This means that the ℓ_{tp} summarizer produces something close to a uniform probability distribution, while the ℓ_{df} and ℓ_{all} summarizers produce a distribution where there are only a few high relevance sentences and a majority of low relevance sentences—it seems likely that the minority of high relevance sentences are the sentences in the bug description and sentences with high similarity to them.

Figure 8 presents the receiver operating characteristic (ROC) curve for both maximum length and minimum relevance selectors, presenting the false positive rates (1-recall) in the x-axis and the true positive rate (precision) on the y-axis. Since the ROC curve effectively shows the trade-off in increasing recall at the expense of decreasing precision, the curve is often used to compare different information retrieval approaches by measuring the Area Under the Curve (AUC). The curve generally starts of at (1,1), where there are few positives, recall is very low and precision high, and progresses to (0,0) where there are many positives, recall is very high but precision is low. As a result, the closer the curve comes to

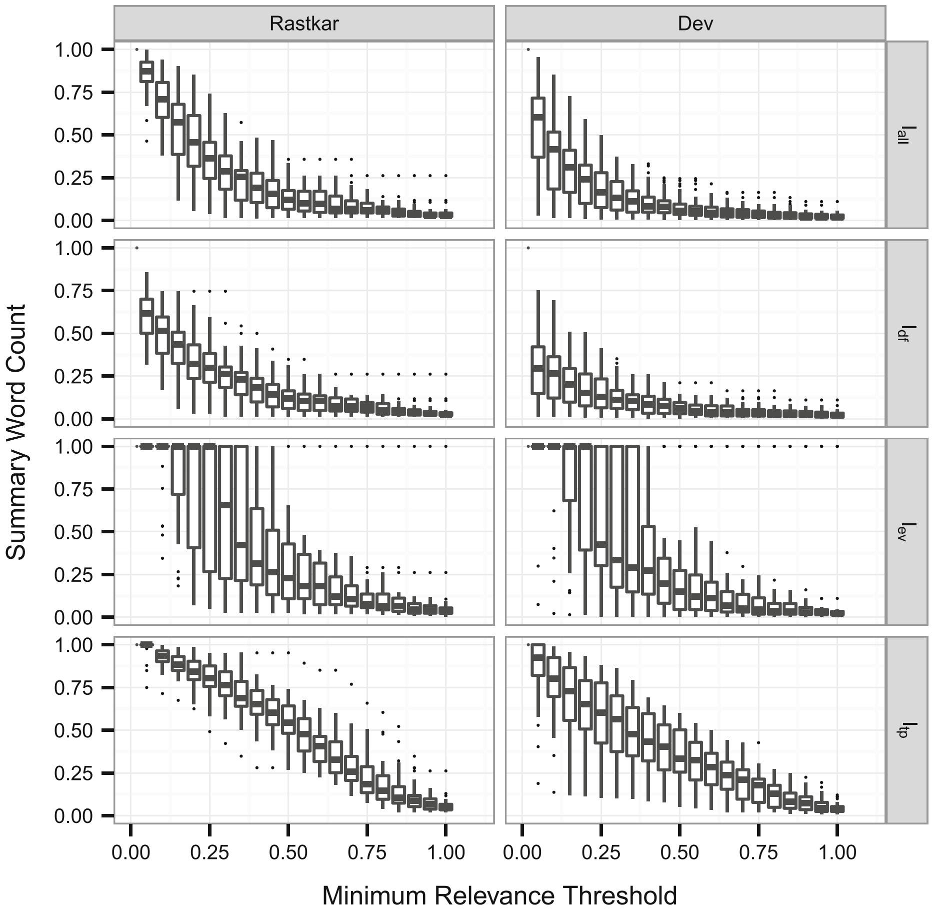


Fig. 7 Boxplots showing how the final summary length decreases as we increase the minimum relevance threshold values and that each summarizer has a characeteristic curve shape

(0,1)—high recall and precision—the higher the AUC. Thus, Fig. 8 shows that, first of all, the AUC for minimum relevance is slightly greater than the AUC for maximum length, for all summarizers. More importantly, particularly for high relevance scores, the minimum relevance selector produces significantly higher precision than maximum length selector for the ℓ_{all} summarizer. Thus, although the user would not be able to predetermine the length of the resulting summaries, it effectively allows the user to better control the *precision* of the resulting summary.

Although the minimal relevance threshold parameter eliminates the need for users to input the desired summary length, it does require users to input the desired minimum sentence relevance. This is beneficial when the user does not know the desired summary length but has a low tolerance for false-positives—irrelevant sentences included in the summary—and only wants the most relevant sentences. Thus, if users have a priori knowledge or constraints on the summary lengths, they should use the maximum summary length parameter; otherwise, if they have constraints on the false-positive rates for the summary, the minimum relevance threshold parameter would be more appropriate.

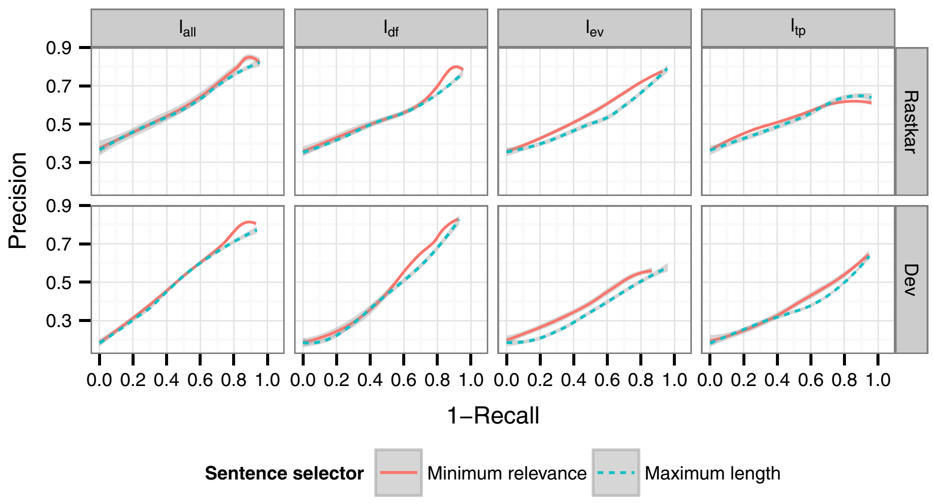


Fig. 8 The ROC curves for the ℓ_{all} summarizer for both the Rastkar and the developer bug corpus shows that the relevance threshold selector has a higher AUC than the maximum length selector, allowing the user to better control the precision of the resulting summary

5.2 Suggesting Summary Length that Optimizes Quality

While we have presented an alternative to the target summary length parameter, we still have not presented a tool to aid the choice of neither parameters. Without such aid, using a target summary length that is shorter than the expert summary length would produce a summary with low recall and using a target summary length that is larger than the expert summary length would produce a summary with low precision, regardless of how well a summarizer performs.

Unfortunately, there are few studies investigating the optimal length for summaries (Lloret and Palomar 2012). We could easily suggest small summary target lengths, 25 % of the original size, for example, for all bug reports. This would lead to summaries with high precision, but low recall. On the other hand, we could suggest large summary target lengths, say 70 % of the original size, leading to lower precision but higher recall rates, or medium-sized summary lengths for a balanced precision and recall. Optimally, we would like to be able to suggest an appropriately-sized summary that balances well precision and recall to the needs of the user.

If we could predict how well a summarizer would perform when summarizing each bug report, then we could aid the choice of the input parameter, suggesting short lengths or high relevance threshold for cases in which the summarizer performed poorly, or long lengths or lower relevance for cases in which the summarizer performed well. We will use the ROC curve, introduced previously, to illustrate this. Since recall and precision are higher the closer a point is to (0,1), the input parameters responsible for the points in the curve that are closest to (0,1) would be good choices. Figure 9 shows the ROC curve for four bug reports in the Rastkar corpus for the ℓ_{all} summarizer, while varying the target length parameter. The curve starts close to (1,1), where low target lengths produce low recall, but generally high precision summaries. The curve then progresses towards (0,0), where high target lengths produce high recall, but low precision summaries.

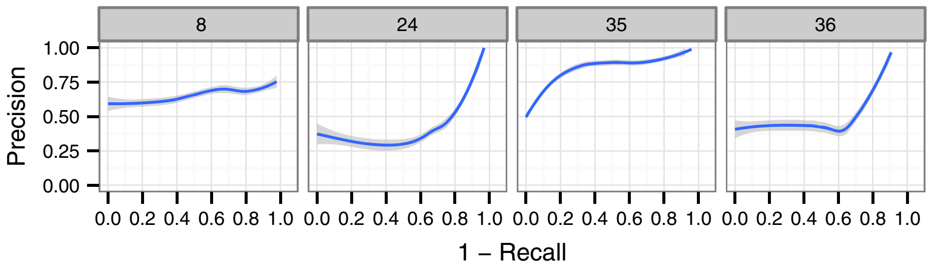


Fig. 9 ROC curve for bugs 8, 24, 35, and 36 in the Rastkar corpus for the ℓ_{all} summarizer

The figure shows, however, that the path from one extreme to the other is different for each of the bug reports. The curves for bugs 24 and 36, for instance, present a sharp decrease in precision while recall varies from 0 to 30 %; for bugs 8 and 35, on the other hand, precision remains constant as the recall increases up to 75 %. It seems, therefore, for bugs 8 and 35, which have high AUC, we could recommend longer summaries, since the result would be high recall with still relatively high precision; for bugs 24 and 36, which have lower AUC, on the other hand, it seems it would be best to recommend shorter summaries so that precision is not diminished at the expense of higher recall. One could, in fact, for bugs 24 and 36, present to the user the choice of a longer summary with high recall and low precision or a shorter summary with low recall but high precision.

It seems, therefore, that we would be able to suggest good summary lengths if we were able to predict the AUC for each bug report, i.e., the difficulty the ℓ_{all} summarizer had when summarizing each bug report. Nenkova and Louis (2008) found that the difficulty of summarizing a text, for humans or machines, can be predicted by the textual Shannon entropy (15), word count, and vocabulary size. Particularly, they suggest that low entropy values for a text generally indicates that the text is more cohesive and, thus, easier to summarize.

$$\text{entropy}(B) = -\sum_{x \in \text{vocab}(B)} P_B(x) * \log(P_B(x)) \tag{15}$$

$$P_B(x) = \frac{\#\text{ occurences of } x \text{ in } B}{\text{wordcount}(B)} \tag{16}$$

With the insight from Nenkova, we use the Rastkar corpus as a training set and find that ϕ, defined in (17), a relation between the three measurable properties of texts identified by Nenkova—entropy, word count, and vocabulary size—has a small, but significant correlation with the AUC for each bug in the Rastkar corpus—Spearman rank correlation of 0.20 and p-value < 0.03.

$$\phi(B) = \text{entropy}(B) \div \text{wordcount}(B) \div |\text{vocab}(B)| \tag{17}$$

To evaluate if ϕ can be used to suggest appropriate summary lengths for bug reports when using the ℓ_{all} summarizer, we use $\bar{\phi}$, the mean of ϕ, as a threshold to decide between two predefined summary lengths: a short summary if $\phi > \bar{\phi}$ or a long summary otherwise. We then measure the recall/precision ratio for the summaries for which the predictor decided for a longer summary, effectively measuring how much recall we can gain by every afforded 1 % precision. Finally, we compare the recall/precision ratio resulting from using ϕ as a predictor for summary lengths to the recall/precision rates resulting from using a random predictor—we repeat the random experiment 100 times and take the average precision and recall.

Figure 10 shows the difference between the recall/precision ratio in percentage points, when using 25 % as the short summary length and varying the long summary lengths from 30 to 70 %, with increases of 5 %. This allows us to note the difference in recall/precision ratio when increasing the summary length from 25 to 30 %, from 25 to 35 %, etc. The figure shows that, for the Rastkar corpus, when choosing between a 25 % and a 30 % long summary, the ϕ predictor creates summaries with 110 % better recall/precision ratio than when randomly choosing between a 25 or 30 % long summary. When deciding between a 25 and a 50 % long summary, the ϕ predictor creates summaries with 25 % improvement over the random predictor. As can be seen, although the improvements for the developer corpus are not as large as for the Rastkar corpus when the long summary varies from 30 to 40 %, it does produce improvements of at least 8 % over the random predictor, effectively showing that the ϕ predictor, which was originally tailored to the Rastkar corpus, is more general and can be applicable to other corpora when using the ℓ_{all} summarizer.

6 Future Directions

The results of our work indicate three important future directions, which we discuss in the following sections.

6.1 Improving Sentence Relevance Estimates

Improving the estimates on sentence relevance should significantly improve summary quality, and involves improving the techniques to estimate ℓ_{tp} and ℓ_{ev}. To improve ℓ_{tp}, one could study the use of LDA, PLSA, or other natural language processing technique in which similarity is measured using topics. Quan et al. (2009), for example, showed a promising

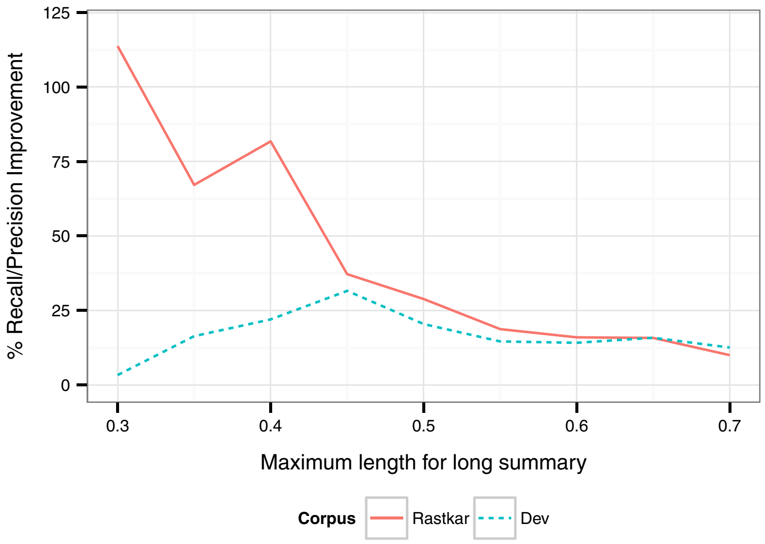


Fig. 10 Precision/recall improvement over random predictor when using ϕ to suggest a shorter or longer summary. The chart shows that ϕ is a good predictor for the difficulty of summarizing a bug report

approach to use topic models to estimate the similarity of very short texts. Additionally, once topics are identified, it might be relevant to give topics different weights: the topic of solutions to a bug, as stated by developers, could be considered more relevant than the topic of who is fixing the bug, for example. To improve ℓ_{ev}, the use of a corpus trained on sentences from bug reports should be promising, and could use the characteristics of communication in bug reports, beyond emoticons, to automatically annotate sentence polarity. Another line of investigation should look for other characteristics in bug report comments that increase the relevance of a sentence. Contributors who are well-known to post important information, for example, could be one such characteristic. Feedback from developers also indicate the need to improve the recognition and handling of structured information, such as code snippets, diagnostic information, and enumerations.

6.2 Moving Past Extractive Summaries

Once one has a better understanding of the topics being discussed in bug reports, one can not only give different weights to these topics, as explained in the previous section, but also move past summaries that are composed solely of extractive sentences. Such a summary could also *restructure* the information in different ways, as indicated by our developer evaluation, moving all the information on solutions to a particular section and provide information such as the number of users for which the solution was applicable for, to indicate if there is consensus.

6.3 Creating Personalized Summaries

Our summarization approach generates a general purpose bug report summary. Our heuristics do not take into account the specific objective of a developer or user when reading a bug report. It is quite reasonable that a developer triaging a bug report has different information needs than a developer seeking a workaround for a bug, or a developer looking for duplicate bug reports. An important future direction to improve bug report summaries would be to first understand the tasks that such a general purpose summary is most helpful for and to adapt the summarization approach to create personalized summaries suited to task at hand and to the reading preferences of the reader.

The proposed approach could be extended to support personalized summaries. First one would need to determine what are the topics or terms that are most relevant for a user's particular purpose. One option for determining relevant terms is to allow users to explicitly input the terms they are interested in. Another option would be to find the most relevant terms for specific tasks. Given these relevant terms, we need to simply boost relevance for the sentences in the Markov chain that contain them.

6.4 Creating Interfaces to Support Bug Report Navigation

Designing optimal interfaces to facilitate bug report navigation based on summaries and the links we have identified is another important direction of work. Results from our developer evaluation indicate that condensed and interlaced views have their own advantages and are suitable for different scenarios and user preferences. This suggests, for example, that users should be able to switch between the two formats. Additionally, a dynamic version of the interlaced format could highlight the links from the current sentence to the sentences it has stronger links with, as the user navigates through the bug report and focuses on different sentences.

7 Related Work

7.1 Textual Summarization

There is a large body of research on textual summarization (Lloret and Palomar 2012). While the approach we use is based on graph theory, there are summarizers that are based on statistical reasoning, others that are topic-based and identify relevant sentences from words that are likely to indicate importance, and others that are discourse-based and identify relevant sentences based on discourse relations. Edmundson (1969), for example, in a topic-based summarization approach, determined the relevance of a sentence by means of the phrases or words it contained. They considered, for example, that sentences containing words like "*in conclusion*" or "*the aim of this paper*" might contain relevant topics. Mann and Thompson (1988), in a discourse-based summarization approach, used Rhetorical Structure Theory (RST) to structure the discourse into nucleus and satellite relations and determine the most important textual units in a document.

The graph-based summarization approach that we use has been previously used to summarize text. TextRank (Mihalcea and Textrank 2004) and Lexrank (Radev 2004) also calculate sentence probability using textual similarity. Lexrank adds a post-processing step after sentence ranking to avoid redundancy by excluding sentences that *subsume* other sentences. We are the first, however, to propose the use of PageRank for bug report summarization and to adapt it to the bug report domain, considering the importance of evaluation links and the similarity of sentences to the bug's title and description.

7.2 Summarizing Bug Reports

There have been a couple of works on summarizing bug reports. Rastkar et al. (2010) summarized bug reports using a summarization approach for email threads. The bug reports they tested their approach on were manually selected to resemble email messages, since they exclude bug reports with stack traces, logs, and patches and preferred bug reports with conversational content. Although we also used this corpus for evaluation, we additionally sampled bug reports from the wild: our only restriction is that a bug report has at least 10 comments. To evaluate their summaries, Rastkar et al. asked graduate students to evaluate the summaries they created. Alternatively, we asked the developers who actually worked on the bugs to evaluate the summaries.

A second attempt at summarizing bug reports was performed by Mani et al. (2012). To overcome the difficulties of using the supervised summarization approach proposed by Rastkar et al. (2010) and eliminate the overhead of creating a supervised predictor, they evaluated the quality of summaries generated by four well-known general purpose unsupervised textual summarizers. They found, however, that these approaches only converge after removing noise from bug reports. They propose a heuristic-based noise reduction approach that tries to automatically classify sentences as either being a question, an investigative sentence, or a code snippet. Anything that is not one of these is discarded. When using this noise reduction heuristic they found that each of the four well-known textual summarizers produced a summary of at least equal quality compared to the supervised approach.

Other work aimed at facilitating bug report digestion comes from Ankolekar et al. (2006), who identified and automatically linked bug reports to important information about bugs. Such links, they claimed, should answer questions of 'what', 'why', and 'who', such as who created a function and what the function is for. Dit and Marcus (2008) proposed a recommendation engine to find the comments that a comment is related to, to improve the readability of bug reports. They do not implement such a system, only acknowledging that detecting these links is a difficult problem.

Bettenburg et al. (2008a) performed a survey with developers from Apache, Mozilla, and Eclipse, asking them what information is important in a bug report description, and what information is frequently absent from bug reports. Developers agreed that the description of the steps to reproduce the bug and stack traces are the most important information.

7.3 Other Applications of Summarization for SE

Sridhara et al. (2010) generated textual summaries to describe Java methods. They used a number of heuristics on a method's signature and a method's code to find the main intent of the code. The heuristics work on breaking the source code down into smaller units, e.g. statements, and then on selecting the most relevant units to base the summary on. After selecting the units, they used natural language generation techniques to convert each unit into a natural language phrase. Developers who judged the quality of the summaries found them to be accurate, concise, and complete. Haiduc et al. (2010), with the same objectives, used well-known information retrieval methods to generate term-based summaries of methods and classes. These term-based summaries do not compose a grammatically valid phrase, but are intended to be the set of most relevant terms that together should describe an entity. The proposed approach first creates a corpus, by breaking down the source code tokens into terms; finally they used information retrieval methods to select the most relevant terms to describe the class or method. They experimented with a number of techniques for corpus creation and term selection and found that a particular combination of techniques identifies the intent of methods and classes more accurately than other previous approaches.

Hamou-Lhadj and Lethbridge (2006) summarized large execution traces to make it easier for software engineers to understand a programs behavior. Their approach assumed that developers, when trying to understand the content of large traces, will not be interested in calls to utility routines. They defined a metric to estimate the utility of a routine, based on the number of other routines that call the routine and on the number of routines that the routine calls. The metric assumes that utility routines are called by a large number of unrelated routines and will be largely self-contained, i.e., will not call out to a large number of other routines. The evaluation showed that this approach can be used to create adequate summaries of execution traces with over 100,000 method calls.

7.4 Using Bug Report Content to Automate SE Tasks

There have been many approaches of using textual analysis on bug report content to automate software engineering tasks. Wang et al. (2008), for example, presented the current, most effective, approach to identify duplicate bug reports. They built on the previous duplicate bug report detection attempts that used only the textual bug description (Runeson et al. 2007; Hiew 2006) to disambiguate bugs and instead used both textual descriptions and bug execution traces to disambiguate bug reports. The approach is reported to detect as much as 90 % of duplicate bug reports.

Automated textual analysis in bug reports has also been used to estimate how long it will take to fix a bug (Weiss et al. 2007), which developer has the most experience to fix a particular bug report (Anvik et al. 2006), and predicting the severity and category of bug reports (Tian et al. 2012; Menzies and Marcus 2008; Thung et al. 2012). All these approaches used very similar textual analysis techniques to the ones we have used, representing bug reports as a vector where features are the number of occurrences of each word, using cosine similarity to measure the similarity of bug reports and to group similar bug reports, and using support vector machines for machine learning.

8 Threats to Validity

8.1 Internal

Although Strauss and Corbin (2008) suggest the use of more than one data source for grounded theory, our investigation in Section 2.1 does not triangulate bug reports with developer interviews or other sources of data. We do, however, compare bug reports from four distinct bug tracking systems and, since we use this investigation to pose hypotheses that were later tested in two distinct ways, we consider the methodology for the investigation appropriate.

To test the three hypotheses presented in Section 2.2.5, we use a comparison of the evaluation metrics with the summaries created by the email summarizer. We consider this is a reasonable first test, since Rastkar et al. claim the email summarizer produces good-quality summaries. To mitigate this threat, we also ask expert developers to assess our summaries and find that they consider our summaries useful.

We consider that the threats to replicate the results of the first evaluation, which compares the quality metrics between the proposed summarizers and the email summarizer are considerably low. Since the algorithm for implementing the email summarizer and the three summarizers we propose have been precisely explained, a well-developed implementation of these summarizers should reliably reproduce our results. While our own tests for assuring that our implementation of the email summarizer did not compare individual summaries with the summaries created by Rastkar's own implementation, we compared it with Rastkar's reported average pyramid score, precision, recall, and f-score using the non-parametric Mann-Whitney U test, but found no significant difference (p-value > 0.01). Another replication threat comes from the polarity detection mechanism. Different implementations of the ℓ_{ev} summarizer might produce slightly different results when training the sentiment analysis classifier with different samples from Twitter. Since we have taken a random sample of 2,400,000 messages from, we consider this threat should not be significant.

Our evaluation of the two input parameters we propose (summary percentage length and minimum relevance threshold) is significantly limited. Although we present the quality metrics (precision and recall) for summaries of all 94 bug reports (36 from Rastkar corpus and 58 from our developer corpus) for a wide range of both input parameters, we do not provide an evaluation showing how users would choose such input parameters and for which tasks they would find each input parameter most suitable for. Similarly, we do not provide an evaluation showing how users would use the tool suggesting an optimal summary length and if they would find such a tool indeed helpful.

8.2 External

We claim our summarizer should be widely applicable to any bug tracking system. We consider this is a reasonable claim, since we pose our hypotheses from an analysis of a set of bug tracking systems, but test our hypotheses on a different set of bug tracking systems, the ones from the corpus created by Rastkar et al. Furthermore, the risk of our summarization approach being *over-fitted* is small, since we do not use machine learning, but heuristics that should be valid in most bug tracking systems.

Our evaluation with open source developers, however, has more significant threats due to difficulty of obtaining a larger number of participants. Although we consider 58 developers to be a good number of participants, and that we invited a random set of developers to participate in the study, it is possible that the sample of developers who participated is biased, i.e, only the developers who already were interested in bug report summaries participated in our evaluation. Since ultimately developers have a choice to participate or not, this is a threat that is difficult to avoid. As an attempt to overcome this threat, we invited 3 developers to work on each bug report. Unfortunately, we could not attract the participation of more than one developer per bug report for the majority of bug reports. Another bias factor for the developers' responses arises from the fact that we did not collect the task or responsibility of the developer that worked with each bug report. It is possible that developers performing triaging are more inclined to favor bug report summaries than developers who have actively worked on fixing the bug and accessed it on a daily basis.

The evaluation of the two proposed input parameters and the tool to suggest an optimal summary length have similar threats to generalizability. Although all our tests were validated on 94 bug reports, the potential of a biased sample set might influence our results.

9 Conclusion

We have created an automatic, unsupervised, bug report summarization approach that should be widely applicable to different bug tracking systems. The summarization approach we propose models how a user, with limited time, would read a bug report. We hypothesize that such a reader would navigate through a bug report following sentences that talk about the same topics, that have been evaluated by other sentences, and that are close to the bug report title and description. Our evaluation, performed by comparing evaluation metrics against a previous summarization approach and by a quality assessment made by developers, shows not only that this summarization approach produces good quality summaries, but also validates the importance of solving such a problem for developers.

The summarizer requires, as input, only the bug report to be summarized and a parameter specifying the target summary length for the original bug report. We propose, nevertheless, an alternative input parameter that allows the user to better control the quality of the resulting summary. We consider that the two input parameters might be valuable to facilitate generating summaries that meet a user's quality expectation.

Still with the objective of facilitating the choice of the input parameter, we use the entropy of the bug report to calculate ϕ, an estimate of the difficulty that the summarizer will have in summarizing a bug report. We show that ϕ can be used to suggest a target summary length for a particular bug report to optimize the precision of a bug report summary, given a particular expectation of recall: if ϕ is greater than a certain threshold, we can safely ask for a longer summary and produce a summary with higher recall and precision; if ϕ is lower than the threshold, we suggest the use of a shorter summary, reducing the recall but maintaining a higher precision.

Finally, our work suggests four future directions of work: improving the estimation on sentence relevance, moving past extractive summaries, creating personalized summaries, and creating interfaces to support navigation based on summaries.

References

Ankolekar A, Sycara K, Herbsleb J, Kraut R, Welty C (2006) Supporting online problem-solving communities with the semantic web. WWW

Anvik J, Hiew L, C Murphy G (2006) Who should fix this bug? In: Proceedings of the 28th international conference on software engineering. ACM

Beineke P, Hastie T, Manning C (2004) Exploring sentiment summarization. AAAI

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008a) What makes a good bug report? SIGSOFT

Bettenburg N, Premraj R, Zimmermann T (2008b) Extracting structural information from bug reports. MSR

Blei DM, Ng Y, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res

Boehm B, Basili VR (2001) Software defect reduction top 10 list. IEEE Comput:34

Breu S, Premraj R, Sillito J, Zimmermann T (2010) Information needs in bug reports: improving cooperation between developers and users. Comput Supported Coop Work

Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. WWW

Büttcher S, Clarke C, Cormack G (2010) Information retrieval: implementing and evaluating search engines. MIT Press

Dit B, Marcus A (2008) Improving the readability of defect reports. RSSE

Edmundson HP (1969) New methods in automatic extracting. J ACM (JACM) 16(2)

Gasser L, Ripoche G (2003) Distributed collective practices and free/open-source software problem management: perspectives and methods. CITE

Go A, Bhayani R (2009) Twitter sentiment classification using distant supervision. CS224N Project Report, Stanford

Haiduc S, Aponte J, Moreno L, Marcus A (2010) On the use of automated text summarization techniques for summarizing source code. In: 2010 17th working conference on reverse engineering (WCRE). IEEE

Hamou-Lhadj A, Lethbridge T (2006) Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: 14th IEEE international conference on program comprehension, 2006. ICPC 2006. IEEE

Hiew L (2006) Assisted detection of duplicate bug reports, Master's thesis, The University of British Columbia

Hofmann T (1999) Probabilistic latent semantic indexing. In: SIGIR. ACM

Lloret E, Palomar M (2012) Text summarisation in progress: a literature review. Artif Intell Rev 37(1)

Lotufo R, Malik Z, Czarnecki K (2012a) Modelling the 'hurried' bug report reading process for bug report summarization. ICSM

Lotufo R, Passos L, Czarnecki K (2012b) Towards improving bug tracking systems with game mechanisms. MSR

Mani S, Catherine R, Sinha VS, Dubey A (2012) Ausum: approach for unsupervised bug report summarization. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. ACM

Mann WC, Thompson SA (1988) Rhetorical structure theory: toward a functional theory of text organization. Text 8(3)

Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. In: IEEE international conference on software maintenance, 2008. ICSM 2008. IEEE

Mihalcea R, Textrank PT (2004) Bringing order into texts. EMNLP

Murray G (2008) Summarizing spoken and written conversations. EMNLP

Nenkova A, Louis Ae (2008) Can you summarize this? Identifying correlates of input difficulty for generic multi-document summarization

Nenkova A, Passonneau R, McKeown K (2007) The pyramid method: Incorporating human content selection variation in summarization evaluation. ACM Trans Comput Logic

Porter MF et al (1980) An algorithm for suffix stripping

Quan X, Liu G, Lu Z, Ni X, Wenyin L (2009) Short text similarity based on probabilistic topics. Knowl Inf Syst

Radev DR (2004) Lexrank: graph-based lexical centrality as salience in text summarization. Artif Int

Rastkar S, Murphy GC, Murray G (2010) Summarizing software artifacts: a case study of bug reports. ICSE

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: Proceedings of the 29th international conference on software engineering

Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM international conference on automated software engineering. ACM

Strauss A, Corbin J (2008) Basics of qualitative research: techniques and procedures for developing grounded theory. Sage Publications

Sun B, Mitra P, Giles CL, Yen J, Zha H (2007) Topic segmentation with shared topic detection and alignment of multiple documents. SIGIR

Tang H, Tan S, Cheng X (2009) A survey on sentiment detection of reviews. Exp Syst Appl

Thung F, Lo D, Jiang L (2012) Automatic defect categorization. In: 2012 19th working conference on reverse engineering (WCRE). IEEE

Tian Y, Lo D, Sun C (2012) Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 2012 19th working conference on reverse engineering (WCRE). IEEE

Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th international conference on software engineering. ACM

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: Proceedings of the 4th international workshop on mining software repositories. IEEE Computer Society

Rafael Lotufo graduated as a Ph.D student in the Department of Electrical and Computer Engineering at the University of Waterloo, under the supervision of Krzysztof Czarnecki. Before starting his Ph.D, he worked for 6 years leading software development teams build enterprise software solutions, and has extensive knowledge of the issues that arise in practice in software engineering. His Ph.D thesis tackles one of these biggest problems: improving bug tracking systems to help users work with bug reports and collaborate to fix bugs faster. His work includes motivational mechanisms to engage users in productive collaboration, as well as automated natural processing techniques to summarize bug reports, and a paradigm breaking interface for bug tracking systems that removes comments as the most important content in bug reports.

Zeeshan Malik is a graduate student in Mathematics at the University of Waterloo, pursing a Master's degree in Health Informatics. He works with automated human activity recognition for remote patient monitoring. Before starting the graduate degree, Zeeshan received his Bachelor's degree in Computer Science from Lahore University of Management and Sciences (LUMS) and worked as software engineer for four years.

Krzysztof Czarnecki is a Professor of Electrical and Computer Engineering at the University of Waterloo. Before coming to Waterloo, he was a researcher at DaimlerChrysler Research (1995-2002), Germany, focusing on improving software development practices and technologies in enterprise, automotive, space and aerospace domains. He co-authored the book on ”Generative Programming” (Addison- Wesley, 2000), which deals with automating software component assembly based on domain-specific languages. While at Waterloo, he held the NSERC/Bank of Nova Scotia Industrial Research Chair in Requirements Engineering of Service-oriented Software Systems (2008-2013) and has worked on a range of topics in model-driven software engineering, including software-product lines and variability modeling, consistency management and bi-directional transformations, and example-driven modeling. He received the Premier’s Research Excellence Award in 2004 and the British Computing Society in Upper Canada Award for Outstanding Contributions to IT Industry in 2008.