Modelling the 'Hurried' Bug Report Reading Process to Summarize Bug Reports

Rafael Lotufo, Zeeshan Malik, Krzysztof Czarnecki
University of Waterloo
{*rlotufo, zmalik, kczarnec*}*@gsd.uwaterloo.ca*

***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 show the summaries are useful, but also point out important requirements for this, and any bug summarization approach, and indicates directions for future work.**

I. Introduction

Bug reports are valuable assets in software development projects. Not only do they serve as a communication medium for bug resolution—an activity that accounts for as much as 40% of software development efforts [1]—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 [2].

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 occurs 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. [3] recognize the similarity between bug report messages and email threads and use a pre-existing summarization technique created to summarize email threads and conversations [4]. 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—*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 **develop a deeper understanding of the information exchanged in bug reports** and to use this knowledge to **create an unsupervised summarization approach** that should be readily applicable to virtually any bug tracking system without need for configuration nor of a corpus of manually created golden summaries and 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

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

978-1-4673-2312-3/12/$31.00

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* 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 II). We use this model to rank sentences by its probability of being read and compose the summary with the sentences with the highest probabilities (Section III).

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 IV) by generating summaries for the same 36 bug reports used by Rastkar et al. [3] 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 IV). 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.

This work provides five main contributions: (i) a deeper understanding of the information exchanged in bug reports; (ii) techniques to identify the most important information in bug reports; (iii) a novel, unsupervised, approach for summarizing bug reports that should be readily applicable for any bug tracking system; (iv) a survey with 58 open source developers highlighting the use cases for which bug report summaries are most useful for; and (v) three directions for future work motivated by our findings: improving the calculation of a sentence's relevance, moving past extractive summaries, and designing interfaces to facilitate bug report navigation based on a summary.

II. 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 approximate this process by a Markov chain. 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. We can model a Markov chain by a square matrix $\mathcal{M}$ where each element $m_{i,j}$ represents the probability of transitioning from sentence s_i to sentence s_j. This model however, still lacks the probability transitions from one sentence to another. Estimating such probabilities requires us to first understand what sentences a user finds important to read, based on the sentence he has just read—the *links* a user follows from one sentence to another.

A. 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 understanding 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 [5]. 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. Furthermore, keeping track of the evolving knowledge about a bug becomes more difficult with more users discussing a bug report, since the conversation becomes more interwoven

and multi-threaded and demands readers to keep track of additional contexts.

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 for which environments each of these solutions and workarounds are suited for. Lotufo et al. [2], in previous quantitative analysis, find that at least 27% of comments in bug reports result from the evaluation of other comments (C3). This finding is also supported by Breu et al. [6]. Gasser and Ripoche [7] perform a related study on bug reports and also find 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.

B. Hypotheses

The findings from our qualitative investigation suggest that, in order to understand a bug report, it is important that (i) users follow the threads of conversation containing the topics they are interested in, from start to finish, to minimize the probability of missing important information; and that (ii) users give particular attention to sentences that have been evaluated by other sentences, since they set the context for much of the following comments. Furthermore, for users with limited time, in order to focus on the most important points of the bug, (iii) 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. The transition probability from one sentence to another for the Markov chain modelling the bug report reading process should, therefore, be higher the more the two sentences talk about the same topics. Similarly, the transition probability from a sentence to one that it evaluates should also be higher compared to the transition probabilities to other sentences.

From the Markov chain, 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.

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 sentences and the more relevant are the sentences it shares topics with;

Hypothesis 2: the relevance of a sentence is higher the more it is evaluated by other sentences and the more relevant are the sentences that evaluate it;

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

III. Summarization Approach

Brin and Page [8] develop PageRank to rank web pages by relevance using a very similar model. They estimate the relevance of a web page as the probability of a user reaching that page for a user who surfs the web by randomly following hyperlinks from one web page to another—a random surfer. PageRank takes as input a graph G, where web pages are nodes and hyperlinks from one page to another are directed edges. PageRank then calculates the Markov chain $\mathcal{M}$ for the random surfer model 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 similarity of the random surfer model and the bug report reading model we have proposed, applying PageRank to calculate the probabilities of sentences being read requires only that we model the links—the transitions—between sentences in a bug report. We can then derive the Markov chain $\mathcal{M}$ and calculate the probability distribution $\mathbf{R}$, just as in PageRank. We explain this calculation in more detail in Section III-A.

The links we have identified for sentences within a bug report, however, are not as concrete as the hyperlinks in web pages. For now, we will consider we can identify sentences with shared topics and sentences that evaluate other sentences—we describe how we identify these relations in Section III-B and Section III-C. We can then define the shared-topic link and evaluation link for Hypotheses 1 and 2 as $\ell_{\text{tp}}(s_i, s_j)$ and $\ell_{\text{ev}}(s_i, s_j)$ for two sentences in a bug report:

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

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

where a sentence s_i is the i-th sentence within a bug report, $\mathrm{topic\text{-}sim}$ measures how much two sentences talk about similar topics and τ is a predefined threshold. Thus, the topic link is a precondition to the evaluation link: if two sentences do not talk about similar topics, one will never evaluate the other. Similarly, sentences in comment i can only be evaluated by sentences from comments posted after comment i.

While ℓ_{tp} and ℓ_{ev} cover Hypotheses 1 and 2, Hypothesis 3 remains unaddressed. The ℓ_{tp} link will, however, already create a link, and thus increase the probability, of any sentence with similar topics to the description. To boost the relevance of sentences beyond the relevance given by ℓ_{tp} to sentences with similar topics to the bug description, we will 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.

We also want to boost the relevance of sentences with similar topics to the bug report title, since previous works has shown that the title can be a very good summary of a bug report [9]. The bug report title is not, however, one of the sentences we are trying to rank, so we cannot add a link from the title to the sentences with similar topics. To circumvent this issue, for every sentence s_i that shares topics with the bug report title, we add a link from every other sentence to s_i. As a result, for Hypothesis 3, we define ℓ_{td} as in (1), where S_D is the set of sentences within the bug description, and $\ell_{\mathrm{ti}}(t, s) = \ell_{\mathrm{tp}}(t, s)$.

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

A. Measuring Sentence Relevance using PageRank

As we have already briefly introduced, the input to PageRank is a matrix $\mathcal{M}_{n\times n}$ representing a Markov chain where each element $m_{i,j}$ represents the probability of transitioning from state i to state j. When using PageRank to rank web pages, each state is a web page. Since the user is navigating the web by selecting links from a web page at random, the probability of transitioning from web page s_i to web page s_j is $1/L$, where L is the number of links to different web pages in s_i. The formula to calculate $m_{i.j}$ in general is shown below, where each $l(i, j)$ is the weight of the link from state i to j:

$$m_{i,j} = \frac{\ell(i, j)}{\sum_{\forall k} \ell(i, k)} \tag{2}$$

For web pages, $l(i, j)$ returns 1 if there is a link from i to j and 0 otherwise.

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.

$$\mathbf{R} = \left[\frac{1-\delta}{n}\mathbf{U} + \delta\mathcal{M}\right]\mathbf{R} = \widehat{\mathcal{M}}\mathbf{R} \tag{3}$$

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 $\epsilon > 0$.

The only change our original model suffers when using PageRank is that it now must consider that, with probability δ, a user might jump to any sentence in a bug report without following our links. We can use (2) 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 $\ell(s_i, s_j) = \ell_{\mathrm{tp}}(s_i, s_j) + \ell_{\mathrm{ev}}(s_i, s_j)$ and when ℓ_{tp} and ℓ_{ev} measure the *strength* of these links.

Illustrative Example: Figure 1 presents an example of a bug report which we will use to explain our approach. Figure 1 shows the bug report title and then the sequence of 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 a sentence belongs to—with the bug description being comment 0.

Figure 2 presents the graph for our running example for ℓ_{tp} and ℓ_{ev}, where darker nodes represent sentences that evaluate other sentences; thin grey edges represent ℓ_{tp} links;

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.

Figure 1: Bug report for running example.

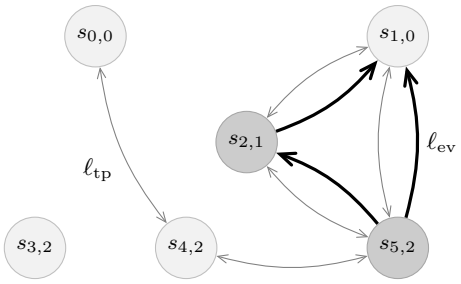

Figure 2: Graph showing ℓ_{tp} and ℓ_{ev} links.

and thick black edges represent ℓ_{ev} links. The figure shows, for example, that sentences $s_{0,0}$ and $s_{4,2}$ have similar topics, since they talk about updating versions, and that sentence $s_{5,2}$ evaluates sentences $s_{1,0}$ and $s_{2,1}$. As can be noted from the definition of ℓ_{tp} and Figure 2, the ℓ_{tp} link is symmetric. Link ℓ_{ev}, on the other hand, is unidirectional: from the evaluator sentence to the evaluated sentence from a previous comment. Thus, although $s_{4,2}$ and $s_{5,2}$ share a same topic and $s_{5,2}$ is an evaluation sentence, $\ell_{\text{ev}}(s_{4,2}, s_{5,2}) = 0$ since they are in the same comment.

Figure 2 also shows that we can compose ℓ_{tp} and ℓ_{ev} to rank sentences by combining Hypotheses 1 and 2. If we use a linear combination to combine these two links with the linear coefficients being 1, the weight of the edge from $s_{2,1}$ to $s_{1,0}$ would be $\ell_{\text{tp}}(s_{2,1}, s_{1,0}) + \ell_{\text{ev}}(s_{2,1}, s_{1,0}) = 2$. The resulting adjacency matrix for our graph would be G, as shown below, where the order of nodes in the matrix are given by the index of the sentence in the bug report:

$$G = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 2 & 0 & 1 & 0 \end{pmatrix}$$

After transforming G into the stochastic matrix $\mathcal{M}$ by dividing each row by the sum of the row, as shown in (2), making it irreducible and aperiodic by multiplying δ and then adding $\frac{1-\delta}{n}\mathbf{U}$, as shown in (3), and solving for $\mathbf{R}$ using the iterative power method, we get the following probabilities for the sentences in our example: $[0.09, 0.24, 0.22, 0.03, 0.15, 0.26]$, effectively ranking sentences as $[s_{5,2}, s_{1,0}, s_{2,1}, s_{4,2}, s_{0,0}, s_{3,2}]$.

Such a model can only be useful, however, if we are able to measure how much two sentences talk about the same topics, and identify sentences that evaluate other sentences. We now present our approach to measure the links ℓ_{tp}, ℓ_{ev}, and ℓ_{td}.

B. 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 [10], for example, measures the changes in mutual information from one chunk of a document to another to detect topics. Latent Dirichlet Allocation (LDA) and Probabilistic Latent Semantic Analysis (PLSA), on the other hand, identify topics using word co-occurrence knowledge extracted from documents.

While arguably these approaches are state-of-the-art at 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 [11]. The cosine similarity is defined as below, where x and y are the vectors of term frequency for a sentence.

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

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.

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

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, but should preserve 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. We can now redefine ℓ_{tp} as:

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

C. Measuring ℓ_{ev}

The relations of evaluation 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 [12], of products, politicians, and almost anything [13]. 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 Bo and Bhayani [14] of using a training set composed of $800,000$ Twitter messages with positive polarity and $800,000$ with negative polarity, polarity which has been automatically annotated as negative or positive polarity using the *emoticons* present in the comments. 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 Bo and Bhayni's [14] 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 III, 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.

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:

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

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

D. Putting it all Together

Each of our links target different characteristics of sentences in bug reports. If these heuristics are valid, we hypothesize, therefore, 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.

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_{ti} for other pairs of sentences:

$$\ell_{\text{all}}(s_i, s_j) = \begin{cases} 1 & \text{if } i = j \wedge s_i \in D, \\ [\alpha\, \ell_{\text{tp}}(s_i, s_j) & \\ \quad +\beta\, \ell_{\text{ev}}(s_i, s_j) & \\ \quad +\gamma\, \ell_{\text{ti}}(s_i, s_j)]\frac{1}{3} & \text{otherwise.} \end{cases}$$

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_{td} only depends on the ℓ_{tp} function which we have already defined, there is no need to redefine it.

E. 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 Bettunburg et al. [15] parses bug report comments to collect structured information, such as stack traces and patches, they do not work on sentence chunking, but acknowledge that it is a non-trivial problem.

Based on our previous experience, we chunk comments into sentences using the following heuristics: (i) each item from an enumeration is a sentence; (ii) each line in a stack trace or source code snippet is a sentence; (iii) we use '.', ';', '?', '!' as sentence delimiters, except when '.' is used in version strings, urls, code snippets, and abbreviations;

(iv) if a line of text has a line break before 80 characters we consider the line break ends a sentence.

IV. Evaluation

A. 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_{td}, 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 [3], 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. [3], which consists of 36 bug reports, each with three reference *golden summaries* created by humans. We use these reference summaries to compare the generated summaries against. For this evaluation, and as was done by Rastkar et al., we generate summaries by selecting sentences until the summary reaches a length of 25% of the original bug report, in number of words, since this is the average length of the golden summaries.

For the second part of our evaluation, we use the ℓ_{all} summarizer to generate summaries for a random set of bug reports from the Debian, Launchpad, Mozilla, and Chrome bug tracking systems, and ask 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*. The condensed format shows only the extracted sentences. The interlaced format presents the complete bug report content, with the extracted sentences shown highlighted out from the other sentences. We ask the developers to: (i) assess the quality of the summary by indicating the mistakes made by the summarizer: sentences that should have been extracted but weren't 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 a summary should contain; and (iv) indicate, using a Likert scale, what are the most important use cases for such summaries.

B. 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 at least half 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 golden summaries.

$$\text{precision}(S) = \frac{|\{S \cap G^*\}|}{|S|} \quad \text{recall}(S) = \frac{|\{S \cap G^*\}|}{|G^*|} \quad (4)$$

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. 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 [16].

Pyramid Score: To circumvent the issues of recall and precision, Nenkova et al. [16] proposes 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 by the sum of the number of golden summaries that contain the n sentences that are most frequently present in 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 most frequently present in golden summaries:

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

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, as the name suggests, 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 (4), with G^* being composed of sentences that are present in any of the golden summaries.

C. Hypotheses Tests

Figure 3 presents for each evaluation metric, the averages—weighted by number of sentences in a bug report—for each of the 5 summarizers we evaluate: ℓ_{tp} summarizer, ℓ_{ev} summarizer, ℓ_{td} summarizer, ℓ_{all} summarizer, and email summarizer. 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. The chart shows, for example, that the

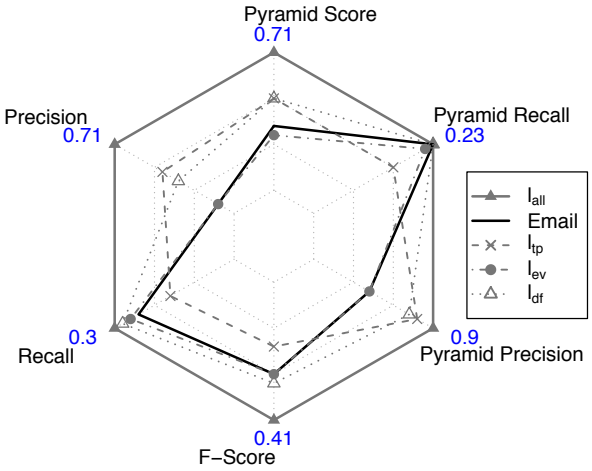

Figure 3: Comparison of evaluation measures for ℓ_{tp}, ℓ_{ev}, ℓ_{td}, ℓ_{all}, and email summarizers for summaries of length 25%.

pyramid score for ℓ_{all} is 0.71, while for email summarizer it is around 0.63, and for ℓ_{tp} it is 0.66.

The results show that ℓ_{tp} summarizer has slightly more 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 almost identical evaluation results compared to the email summarizer. For ℓ_{td}, 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_{td}, only the pyramid precision measure is better than email summarizer, with p-value < 0.01.

We can also use Figure 3 to compare each of our three individual summarizers—ℓ_{tp}, ℓ_{ev}, and ℓ_{td}—amongst themselves. This comparison shows that 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.

Interestingly, all of the summarizers we evaluate here have reasonable precision but quite low recall. Pyramid precision rates of 90% show that the summarizers extract sentences that are present in at least one golden summary with very high precision. The low recall, however, suggests that the summarizers extract sentences that are redundant: sentences that are present in different golden summaries, but that convey similar information. To improve on these results and avoid extracting redundant sentences, one could incorporate one of the many techniques from summarization, such as removing sentences subsumed by other sentences [17].

D. Evaluation with Developers

The evaluation with the developers from the Debian, Launchpad, Mozilla, and Chrome bug tracking systems 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, pointed out by the developers, indicates that sentences, as chunked by the procedure explained in Section III-E, 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.

To evaluate the quality of the summaries, as judged by the developers, we use the mistakes indicated by the developers—sentences that they marked that should have or shouldn't have been present in the summaries—to create golden summaries and calculate 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 recall will have the same values. The results for the 58 bug reports assessed by developers, shown in Table I, indicate that, in average, the summaries include half of the

relevant information, but that around 40% of the sentences in the summaries are not so relevant—the mean and median number of sentences for each of the 58 bug reports are 65.77 and 56. We note that the precision and recall rates are higher for the Chrome and Mozilla projects. We hypothesize this difference might come from the fact that Chrome and Mozilla, from our experience, contain fewer patches and stack traces pasted into bug reports than Debian and Launchpad.

Table I: Precision and recall for developer evaluation.

	Debian	Mozilla	Launchpad	Chrome	All
Precision	0.49	0.77	0.56	0.69	0.59
Recall	0.45	0.60	0.47	0.61	0.51

Responses about the condensed and interlaced summary formats 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."

V. Future Directions

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

A. 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. [18], for example, shows a promising 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.

B. 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.

C. 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.

VI. Related Work

PageRank has been used for text summarization before. TextRank [19] and Lexrank [20] also calculate sentence probability using textual similarity. Lexrank adds a post-processing 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 title and description similarity.

Rastkar et al. [3] summarizes bug reports using a summarization approach for email threads. The bug reports they test on are manually selected to resemble email messages, since they exclude bug reports with stack traces, logs, and patches. Although we also use this corpus for tests, we also sample 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. asks graduate students to evaluate the summaries they create. We ask the developers who actually worked on the bugs to evaluate the summaries.

Other work aimed at facilitating bug report digestion comes from Ankolekar et al. [21], who identifies and automatically links bug reports to important information about bugs. Such links, they claim, should answer questions of 'what', 'why', and 'who', such as who created a function and what the function is for.

Bettenburg et al. [22] perform 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

agree that steps to reproduce and stack traces are the most important information.

VII. Threats to Validity

The grounded theory investigation in Section II-A does not triangulate with developer interviews or other sources of data besides bug reports. We do, however, look into 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.

We use a comparison of the evaluation metrics with the summaries created by the email summarizer to test our hypotheses. 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. Another threat comes from our evaluation with developers. Since only 25% of invited developers agreed to participate, this might be a subset of developers that is already biased about the utility of bug report summaries.

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.

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

References

[1] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34, 2001.

[2] R. Lotufo, L. Passos, and K. Czarnecki, "Towards improving bug tracking systems with game mechanisms," *MSR*, 2012.

[3] S. Rastkar and G. Murphy, "Summarizing software artifacts: a case study of bug reports," *ICSE*, 2010.

[4] G. Murray, "Summarizing spoken and written conversations," *EMNLP*, 2008.

[5] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 2008.

[6] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," *Computer Supported Cooperative Work*, 2010.

[7] L. Gasser and G. Ripoche, "Distributed collective practices and free/open-source software problem management: perspectives and methods," *CITE*, 2003.

[8] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *WWW*, 1998.

[9] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" *MSR*, 2007.

[10] B. Sun, P. Mitra, C. Giles, and J. Yen, "Topic segmentation with shared topic detection and alignment of multiple documents," *SIGIR*, 2007.

[11] S. Büttcher, C. Clarke, and G. Cormack, *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[12] P. Beineke, T. Hastie, and C. Manning, "Exploring sentiment summarization," *AAAI*, 2004.

[13] H. Tang, S. Tan, and X. Cheng, "A survey on sentiment detection of reviews," *Expert Systems with Applications*, 2009.

[14] A. Go and R. Bhayani, "Twitter sentiment classification using distant supervision," *CS224N Project Report, Stanford*, 2009.

[15] N. Bettenburg, R. Premraj, and T. Zimmermann, "Extracting structural information from bug reports," *MSR*, 2008.

[16] A. Nenkova and R. Passonneau, "The pyramid method: Incorporating human content selection variation in summarization evaluation," *ACM Transactions on Computational Logic*, 2007.

[17] E. Lloret and M. Palomar, "Text summarisation in progress: a literature review," *Artificial Intelligence Review*, 2011.

[18] X. Quan, G. Liu, Z. Lu, X. Ni, and L. Wenyin, "Short text similarity based on probabilistic topics," *Knowledge and Information Systems*, 2009.

[19] R. Mihalcea and P. Tarau, "Textrank: Bringing order into texts," *EMNLP*, 2004.

[20] D. R. Radev, "Lexrank : Graph-based lexical centrality as salience in text summarization," *Artificial Intelligence*, 2004.

[21] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty, "Supporting online problem-solving communities with the semantic web," *WWW*, 2006.

[22] N. Bettenburg, S. Just, A. Schrter, C. Weiss, and R, "What makes a good bug report?" *SIGSOFT*, 2008.