Automatic Summarization of Bug Reports

Sarah Rastkar, Gail C. Murphy, *Member, IEEE*, and Gabriel Murray

Abstract—Software developers access bug reports in a project's bug repository to help with a number of different tasks, including understanding how previous changes have been made and understanding multiple aspects of particular defects. A developer's interaction with existing bug reports often requires perusing a substantial amount of text. In this article, we investigate whether it is possible to summarize bug reports automatically so that developers can perform their tasks by consulting shorter summaries instead of entire bug reports. We investigated whether existing conversation-based automated summarizers are applicable to bug reports and found that the quality of generated summaries is similar to summaries produced for e-mail threads and other conversations. We also trained a summarizer on a bug report corpus. This summarizer produces summaries that are statistically better than summaries produced by existing conversation-based generators. To determine if automatically produced bug report summaries can help a developer with their work, we conducted a task-based evaluation that considered the use of summaries for bug report duplicate detection tasks. We found that summaries helped the study participants save time, that there was no evidence that accuracy degraded when summaries were used and that most participants preferred working with summaries to working with original bug reports.

Index Terms—Empirical software engineering, summarization of software artifacts, bug report duplicate detection

1 Introduction

A software project's bug repository provides a rich source of information for a software developer working on the project. For instance, the developer may consult the repository to understand reported defects in more details, or to understand how changes were made on the project in the past. When accessing the project's bug repository, a developer often ends up looking through a number of bug reports, either as the result of a search or a recommendation engine (e.g., [1], [2]). Typically, only a few of the bug reports a developer must peruse are relevant to the task at hand. Sometimes a developer can determine relevance based on a quick read of the title of the bug report, other times a developer must read the report, which can be lengthy, involving discussions amongst multiple team members and other stakeholders. For example, a developer using the bug report duplicate recommender built by Sun et al. [2] to get a list of potential duplicates for bug #564243 from the Mozilla system,[1] is presented with a total of 5,125 words (237 sentences) in the top six bug reports on the recommendation list.

In this paper, we investigate whether concise summaries of bug reports, automatically produced from a complete bug report, would allow a developer to more efficiently investigate information in a bug repository as part of a task. Perhaps optimally, when a bug report is closed, its authors would write a concise summary that represents information in the report to help other developers who later access the report. Given the evolving nature of bug repositories and the limited time available to developers, this optimal path is unlikely to occur. As a result, we investigate the automatic production of summaries to enable generation of up-to-date summaries on-demand and at a low cost.

Bug reports vary in length. Some are short, consisting of only a few words. Others are lengthy and include conversations between many developers and users. Fig. 1 displays part of a bug report from the KDE bug repository;[2] the entire report consists of 21 comments from six people. Developers may benefit from summaries of lengthy bug reports but are unlikely to benefit from summaries of short bug reports. If we target summaries of 100 words, a common size requirement for short paragraph-length summaries [3], and we assume a compression rate of 33 percent or less is likely beneficial, then a bug report must be at least 300 words in length to be worth the summarization. Table 1 shows the number and percentage of summary worthy bug reports in three popular and large-scale open source software projects, computed over the two-year period of 2011-2012. In all these projects, almost one third of bug reports created over the 24-month period are longer than 300 words, suggesting that sufficient lengthy bug reports exist to make the automatic production of summaries worthwhile. The length of a bug report is the total number of words in its description and comments.

Many existing text summarizing approaches exist that could be used to generate summaries of bug reports. Given the strong similarity between bug reports and other conversational data (e.g., e-mail and meeting discussions), we chose to investigate whether existing supervised learning approaches for generating extractive summaries for conversations (Section 2) can produce accurate summaries for bug

1. www.mozilla.org, verified 04/04/12.

2. bugs.kde.org, verified 04/04/12.

- *S. Rastkar and G.C. Murphy are with the Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, British Columbia V6T 1Z4, Canada. E-mail: {rastkar, murphy}@cs.ubc.ca*
- *G. Murray is with the Computer Information Systems Department, University of the Fraser Valley, 33844 King Road, Abbotsford, British Columbia V2S 7M8, Canada. E-mail: gabriel.murray@ufv.ca.*

Manuscript received 27 Jan. 2013; revised 29 Oct. 2013; accepted 17 Dec. 2013; date of publication 8 Jan. 2014; date of current version 1 May 2014.
Recommended for acceptance by H. Gall.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2013.2297712

Bug 188311 - The applet panel should not overlap applets

Product: amarok **Version**: unspecified
Component: ContextView **Priority**: NOR
Status: RESOLVED **Severity**: wishlist
Resolution: FIXED
Target: ---

Description From mangus 2009-03-28 11:35:10

```
Version:            svn (using Devel)
OS:                 Linux
Installed from:     Compiled sources

In amarok2-svn I like the the new contextview , but I found the
new bottom bar for managing applets annoying , as it covers parts
of other applets sometimes , like lyrics one , so that you miss a
part of it. Could be handy to have it appear and desappear
onmouseover.
thanks
```

------- *Comment #1 From Dan* *2009-03-28 14:53:55* -------

```
The real solution is to make it not cover applets, not make it
appear/disappear on mouse over.
```

------- *Comment #2 From Leo* *2009-03-29 14:34:53* -------

```
i dont understand your point, dan... how do we make it not cover
applets?
```

------- *Comment #3 From Dan* *2009-03-29 16:32:22* -------

```
Thats your problem to solve :)

The toolbar should be like the panel in kde, it gets it's own area
to draw in (a strut in window manager terms).  The applets should
not consider the space the toolbar takes up to be theirs to play in,
but rather end at the top of it.
```

Fig. 1. An example of the conversational structure of a bug report; the beginning part of bug 188311 from the KDE bug repository.

reports. These classifier-based approaches, assign a zero or one value to each sentence in the conversation (i.e., bug report) based on features of the sentence; sentences assigned a value of one appear in a generated summary. Fig. 2 shows an example 100-word extractive summary for the bug report of Fig. 1 produced by a conversation-based classifier.

We investigated classifiers trained both on general conversation data and on a corpus of bug reports we created. The corpus consists of 36 bug reports from four open source projects. The reports in the corpus were chosen to represent reports for which summaries would be useful and thus are at least 300 words in length. The bug reports in the corpus include both reports about defects and reports about enhancements. We had human annotators create summaries for each of these reports (Section 3). We also produced three summaries for each report in the bug report corpus using three different classifiers (Section 4). The first classifier was trained on e-mail data. The second classifier was trained on a combination of e-mail and meeting data. We created the third classifier by training it on bug report data. The summary in Fig. 2 was produced using this third classifier trained on the bug report corpus.

TABLE 1
Statistics on Summary-Worthy Bug Reports for Several Open Source Projects, Computed for Bug Reports Created over the 24-Month Period of 2011-2012

Project	Number of all bug reports	Number of bug reports longer than 300 words
Eclipse Platform	7,641	2,382 (32%)
Firefox	10,328	3,310 (32%)
Thunderbird	6,225	2,421 (39%)

SUMMARY: The applet panel should not overlap applets

In amarok2-svn I like the the new contextview , but I found the new bottom bar for managing applets annoying , as it covers parts of other applets sometimes , like lyrics one , so that you miss a part of it.

Applets should not be larger than the viewable area, if there's an applet above it, then the lower applet should get a smaller sizehint, and resize if necessary when it's the active applet (and therefore the only one on the screen)

Just thought i should point out that the feature is not yet completed - the polish that's gone into it lately could seem like an indication of feature completion, and as such it would seem the prudent course to inform you that that is not the case :)

Fig. 2. The 100-word summary for bug 188311 from the KDE bug repository generated by the classifier trained on bug report data.

We measured the effectiveness of these classifiers, finding that the first two classifiers produce good enough summaries for bug reports as the quality of bug report summaries is comparable to summaries produced for e-mail and meeting data by the same classifiers. We also found that the bug report classifier (BRC), having a precision of more than 0.66, out-performs the other two classifiers by 20 percent in generating summaries of bug reports. To evaluate whether this level of precision produces summaries useful for developers, we conducted a task-based user study in which we had 12 participants work on eight bug report duplicate detection tasks. We found that participants who used summaries could complete duplicate detection tasks in less time with no evidence of any difference in accuracy, confirming that bug report summaries help software developers in performing software tasks.

This paper makes four contributions.

- It demonstrates that it is possible to generate accurate summaries for bug reports.
- It reports on the creation of an annotated corpus of 36 bug reports chosen from four different systems.
- It demonstrates that while existing classifiers trained for other conversation-based genres can work reasonably well, a classifier trained specifically for bug reports scores the highest on standard measures.
- It reports on a task-based evaluation that showed bug report summaries can help developers working on duplicate detection tasks save time with no evidence that the accuracy has been impaired.

Our goal in this paper is an end-to-end investigation of bug report summarization; starting with investigating whether summaries can be produced automatically for bug reports and ending by investigating whether automatically generated summaries are useful for software developers. This paper extends our previously presented work on the generation of bug report summaries [4] with a task-based study to evaluate the usefulness of bug report summaries in the context of a real software task: bug report duplicate detection. It also clarifies the description of the approach and provides a more thorough review of related work.

2 Related Work

With the growing amount of electronically available information, there is substantial interest and a substantial body

of work about how to automatically generate summaries of information for easier consumption. Two basic approaches have been taken to generating summaries: extractive and abstractive [5]. An extractive approach selects a subset of existing sentences to form the summary. An abstractive approach builds an internal semantic representation of the text and then applies natural-language processing techniques to create a summary. We focus in this paper on extractive techniques as they have been shown to provide value in other domains and can be applied at lower cost than abstractive approaches.

Many generic extractive text summarizing approaches have been proposed in the literature [5]. Generic summarization techniques do not make any assumption about input. However, in cases where data has particular format or content, these characteristics can be utilized by domain or genre specific summarization techniques to more accurately identify important information to be included in the summary. One such example is summarization of conversation-based data which has been the focus of several research studies. Extraction-based summarization approaches have been applied to many kinds of conversations, including meetings [6], telephone conversations [7] and e-mails [8]. Usually these approaches use domain-specific characteristics, such as e-mail header information [9]. Murray and Carenini [10] developed a generic summarizer for conversations in various modalities that uses features inherent to all multi-party conversations. They applied this system to meetings and e-mails and found that the general conversation system was competitive with state-of-the-art domain-specific systems in both cases. In this work, we investigate whether we can use this general conversation system to generate accurate summaries of bug reports.

Although the format of bug reports varies depending upon the system being used to store the reports, much of the information in a bug report resembles a conversation. Beyond the fixed fields with pre-defined values, such as the status field that records whether the bug is open or closed or some other state, a bug report usually involves free-form text, including a title or summary, a description and a series of time-stamped comments that capture a conversation between developers (and sometimes users) related to the bug. In one common system, the Bugzilla bug reporting system,[3] the description and comments may be written but not edited, further adding to the conversational nature of a report. This lack of editable text also means the descriptions do not serve as summaries of the current report contents.

As a bug repository contain substantial knowledge about a software development, there has been substantial recent interest in improving the use and management of this information, particularly as many repositories experience a high rate of change in the information stored [11]. For instance, Anvik et al. [12] have shown how to provide recommendations to help a triager decide to whom a bug report should be assigned. There has been particular interest in duplicate bug reports, the subject of the task-based evaluation we conducted to determine the usefulness of bug report summaries. In a survey of the Eclipse[4] open source project, Bettenburg et al. [13] found out that duplicate bug reports are not considered a serious problem by developers and at times they can even help developers resolve bugs more efficiently by adding additional information. In a different study of twelve open source projects, Davidson et al. [14] argued that duplicate bug reports, being a product of a lack of knowledge of the current state of the project, are considered a problem especially for medium-sized projects which struggle with a large number of submissions without having the resources of large projects. In either case, having mechanisms that can assist developers in determining duplicate bug reports are desirable and as such several studies investigated the problem of detecting duplicate bug reports. Runeson et al. [15] developed a duplicate detector based on information retrieval methods; their detector achieved a recall of approximately 40 percent. Wang et al. [16] introduced an approach to duplicate detection that used both natural language information in the report and execution traces. They showed that, when execution traces are available, the recall of a detector can rise to between 67 and 93 percent. The detector introduced by Sun et al. [2] is based on an extended version of BM25F, a textual similarity measures in information retrieval. Using the extended BM25F to retrieve a list of potential duplicates, their approach outperforms other techniques previously proposed in the literature.

Other efforts aimed at improving the management of bug reports have considered how to improve the content of bug reports. Ko et al. [17] analyzed the titles of bug reports to inform the development of tools for both reporting and analyzing bugs. Bettenburg et al. [18] surveyed a large number of open-source developers to determine what factors constitute a good bug report and developed a tool to assess bug report quality. Some of the information they identified as being helpful in bug reports (e.g, 'steps to reproduce'), could be added to the content of an automatically produced bug report summary to make it more informative.

Several studies explored the conversational nature of bug reports and the role it plays in facilitating communication in software projects. Breu et al. [19] studied bug reports as means of interaction between users and developers, focusing on questions asked in bug reports and their answers. Sandusky and Gasser [20] found that bug repositories used in open source projects are primary and logical locations for much of the distributed negotiation involved in resolving bugs. A qualitative study conducted by Bertram et al. [21] resulted in the argument that bug repositories have evolved to become more central to the software development process, serving as a focal point for communication and coordination for many stakeholders in small collocated teams. It is this conversational nature of reports that makes them valuable and that we hope to exploit to produce summaries.

Various approaches have considered generating summaries of software artifacts. Building on our earlier results [4], Lotufo et al. [22] investigated an unsupervised approach for extractive summarization of bug

3. www.bugzilla.org, verified 04/04/12.

4. www.eclipse.org, verified 04/04/12.

reports. With their approach, sentences in a bug report are ranked based on such features as whether the sentence discusses a frequently discussed topic or whether the sentence is similar to the bug report's title and description. Their paper reports on an analytic evaluation that shows a 12 percent improvement on the supervised approach we have proposed. Mani et al. [23] investigated four existing unsupervised summarization techniques for bug report summarization. They showed that these techniques can achieve an improvement in precision over the supervised approach [4] if they are used in combination with a noise reduction component. None of this work has subjected their approach to a human task-based evaluation.

Other work on summarizing software artifacts has mostly focused on producing summaries of source code. Haiduc et al. [24] generate term-based summaries for methods and classes that contain a set of the most relevant terms to describe the class or method. Sridhara et al. [25] proposed a technique to generate descriptive natural language summary comments for an arbitrary Java method by exploiting structural and natural language clues in the method. Moreno et al. presented a technique to automatically generate a natural language summary for a Java class to understand the main goal and the structure of the class [26]. In a separate work, we developed techniques for automatic generation of natural language summaries for nonlocalized code, namely crosscutting code for a concern [27]. To the best of our knowledge, the work presented in this paper, is the first attempt to generate meaningful summaries of bug reports and to evaluate the usefulness of the generated summaries in the context of a software task.

3 Bug Report Corpus

To be able to train, and judge the effectiveness, of an extractive summarizer on bug reports, we need a corpus of bug reports with good summaries. Optimally, we would have available such a corpus in which the summaries were created by those involved with the bug report, as the knowledge of these individuals in the system and the bug should be the best available. Unfortunately, such a corpus is not available as developers do not spend time writing summaries once a bug is complete, despite the fact that the bug report may be read and referred to in the future.

To provide a suitable corpus, we recruited ten graduate students from the Department of Computer Science at the University of British Columbia to annotate a collection of bug reports. On average, the annotators had seven years of programming experience. Half of the annotators had experience programming in industry and four had some experience working with bug reports.

3.1 Annotation Process

We had each individual annotate a subset of bugs from four different open-source software projects: Eclipse Platform, Gnome,[5] Mozilla and KDE. We chose a diverse set of systems because our goal is to develop a summarization approach that can produce accurate results for a wide range of bug repositories, not just bug reports specific to a single project. There are a total of 2,361 sentences in these 36 bug reports. This corpus size is comparable to the size of corpora in other domains used in training similar classifiers. For example, the Enron e-mail corpus, used to train a classifier to summarize e-mail threads, contains 39 e-mail threads and 1,400 sentences [10].

The 36 bug reports (nine from each project) have been chosen randomly from the pool of all bug reports containing between five and 25 comments with two or more people contributing to the conversation. The bug reports have mostly conversational content. We avoided selecting bug reports consisting mostly of long stack traces and large chunks of code as we are targeting bug reports with mainly natural language content. The reports chosen varied in length: 13 (36 percent) reports had between 300 and 600 words; 14 (39 percent) reports had 600 to 900 words; six (17 percent) reports had between 900 and 1,200 words; the remaining three (8 percent) bugs were between 1,200 and 1,500 words in length. The bug reports also had different numbers of comments: 14 reports (39 percent) had between five and nine comments; 11 (31 percent) reports had 10 to 14 comments; five (14 percent) reports had between 15 to 19 comments; the remaining six (16 percent) reports had 20 to 25 comments each. The reports also varied when it came to the number of people who contributed to the conversation in the bug report: 16 (44 percent) bug reports had between two and four contributors; 16 (44 percent) other had five to seven contributors; the remaining four (12 percent) had eight to 12 contributors. Nine of the 36 bug reports (25 percent) were enhancements to the target system; the other 27 (75 percent) were defects.

Each annotator was assigned a set of bug reports from those chosen from the four systems. For each bug report, we asked the annotator to write an abstractive summary of the report using their own sentences that was a maximum of 250 words. We limited the length of the abstractive summary to motivate the annotator to abstract the given report. The annotator was then asked to specify how each sentence in the abstractive summary maps (links) to one or more sentences from the original bug report by listing the numbers of mapped sentences from the original report. The motivation behind asking annotators to first write an abstractive summary (similar to the technique used to annotate the AMI meeting corpus [28]) was to make sure they had a good understanding of the bug report before mapping sentences. Asking them to directly pick sentences may have had the risk of annotators just selecting sentences that looked important without first reading through the bug report and trying to understand it. Although the annotators did not have experience with these specific systems, we believe their experience in programming allowed them to extract the gist of the discussions; no annotator reported being unable to understand the content of the bug reports. The annotators were compensated for their work.

To aid the annotators with this process, the annotators used a version of BC3 web-based annotation software[6] that made it easier for them to manipulate the sentences of the

5. www.gnome.org, verified 04/04/12.

6. www.cs.ubc.ca/nest/lci/bc3/framework.html, verified 04/04/12.

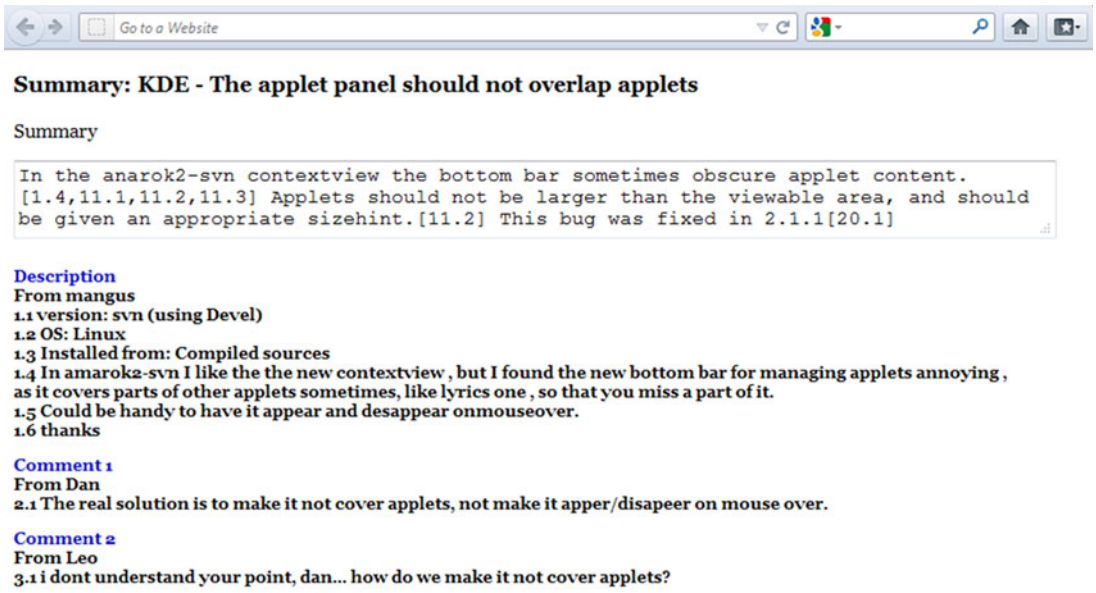

Fig. 3. A screenshot of the annotation software. The bug report has been broken down into labeled sentences. The annotator enters the abstractive summary in the text box. The numbers in the brackets are sentence labels and serve as links between the abstractive summary and the bug report. For example, the first sentence of the abstractive summary has links to sentences 1.4, 11.1, 11.2, and 11.3 from the bug report.

bug report. Fig. 3 shows an example of part of an annotated bug report; the summary at the top is an abstractive summary written by an annotator with the mapping to the sentences from the original bug report marked.

The bug report corpus is publicly available.[7]

3.2 Annotated Bugs

On average, the bug reports being summarized comprised 65 sentences. On average, the abstractive summaries created by the annotators comprised just over five sentences with each sentence in the abstractive summaries linked (on average) to three sentences in the original bug report. Table 2 provides some overall statistics on the summaries produced by the annotators.

A common problem of annotation is that annotators often do not agree on the same summary. This reflects the fact that the summarization is a subjective process and there is no single best summary for a document—a bug report in this paper. To mitigate this problem, we assigned three annotators to each bug report. We use the kappa test to measure the level of agreement amongst the annotators with regards to bug report sentences that they linked in their abstractive summaries [29]. The result of the kappa test (k value) is 0.41 for our bug report annotations, showing a moderate level of agreement.

We asked each annotator, at the end of annotating each bug report, to complete a questionnaire about properties of the report. The annotators, in the answers to the questionnaires, rated (with 1 low and 5 high):

- the level of difficulty of summarizing the bug report to be 2.68 ($\pm$0.86),
- the amount of irrelevant and off-topic discussion in the bug report to be 2.11 ($\pm$0.66), and
- the level of project-specific terminology used in the bug report to be 2.68 ($\pm$0.83).

4 Summarizing Bug Reports

The bug report corpus provides a basis on which to experiment with producing bug report summaries automatically. We produce summaries using binary classifiers that consider 24 sentence features (Section 4.1). It is based on values of these features, computed for each sentence, that it is determined whether the sentence should be included in the summary. To assign a weight to each feature, a classifier first has to be trained on human generated summaries.

We set out to investigate two questions:

1) Can we produce good summaries with existing conversation-based classifiers?
2) Can we do better with a classifier specifically trained on bug reports?

The existing conversation-based classifiers we chose to investigate are trained on conversational data other than bug reports. The first classifier, which we refer to as *EC*, was trained on e-mail threads [10]. We chose this classifier as bug report conversations share similarity with e-mail threads, such as being multi-party and having thread items added at differing intervals of time. This classifier was trained on a subset of the publicly available Enron e-mail

TABLE 2
Abstractive Summaries Generated by Annotators

	mean	stdv
#sentences in the summary	5.36	2.43
#words in the summary	99.2	39.93
#linked sentences from the bug report	16.14	9.73

7. See www.cs.ubc.ca/labs/spl/projects/summarization.html. The corpus contains additional annotations, including an extractive summary for each bug report and labeling of the sentences.

SUMMARY: The applet panel should not overlap applets

In amarok2-svn I like the the new contextview , but I found the new bottom bar for managing applets annoying , as it covers parts of other applets sometimes , like lyrics one , so that you miss a part of it.
Could be handy to have it appear and disappear onmouseover.

The applet should end where the toolbar begins.
Applets should not be larger than the viewable area, if there's an applet above it, then the lower applet should get a smaller sizehint, and resize if necessary when it's the active applet (and therefore the only one on the screen)
Basically, no applet should continue on off the screen, it should end at the panel.

The bug that is being shown here is the fact that you cannot yet resize your applets, and as such we also don't set default sizes sanely.
You are reporting a bug on a non-completed feature ;)

will be fixed in 2.1.1, done locally.

Fig. 4. The gold standard summary for bug 188311 from the KDE bug repository. This is an example of an extractive summary containing sentences that were linked by two or three human annotators.

corpus [30], which consists of 39 annotated e-mail threads (1,400 sentences in total).

The second classifier, which we refer to as *EMC*, was trained on a combination of e-mail threads and meetings [10]. We chose this classifier because some of the characteristics of bug reports might be more similar to meetings, such as having concluding comments at the end of the conversation. The meetings part of the training set for *EMC* is a subset of the publicly available AMI meeting corpus [28], which includes 196 meetings.

The *EC* and *EMC* classifiers are appealing to use because of their generality. If these classifiers work well for bug reports, it offers hope that other general classifiers might be applicable to software project artifacts without training on each specific kind of software artifacts (which can vary between projects) or on project-specific artifacts, lowering the cost of producing summaries.

However, unless these classifiers produce perfect summaries, the question of how good of a summary can be produced for bug reports remains open unless we consider a classifier trained on bug reports. Thus, we also chose to train a third classifier, *BRC*, using the bug report corpus we created. To form the training set for *BRC*, we combined the three human annotations for each bug report by scoring each sentence of a report based on the number of times it has been linked by annotators. For each sentence, the score is between zero, when it has not been linked by any annotator, and three, when all three annotators have a link to the sentence in their abstractive summary. A sentence is considered to be part of the extractive summary if it has a score of two or more. For each bug report, the set of sentences with a score of two or more (a positive sentence) is called the *gold standard summary (GSS)*. For the bug report corpus, gold standard summaries include 465 sentences, which is 19.7 percent of all the sentences in the corpus, and 28.3 percent of all words in the corpus. Fig. 4 shows the gold standard summary for bug 188311 from the KDE bug repository, a portion of the original bug appears earlier in Fig. 1.

As we have only the bug report corpus available for both training and testing the bug report classifier, we use a cross-validation technique when evaluating this classifier. Specifically, we use a leave-one-out procedure so that the classifier used to create a summary for a particular bug report is trained on the remainder of the bug report corpus.

All three classifiers investigated are logistic regression classifiers. Instead of generating an output of zero or one, these classifiers generate the probability of each sentence being part of an extractive summary. To form the summary, we sort the sentences into a list based on their probability values in descending order. Starting from the beginning of this list, we select sentences until we reach 25 percent of the bug report word count.[8] The selected sentences form the generated extractive summary. We chose to target summaries of 25 percent of the bug report word count because this value is close to the word count percentage of gold standard summaries (28.3 percent). All three classifiers were implemented using the Liblinear toolkit [31].[9]

4.1 Conversation Features

The classifier framework used to implement *EM*, *EMC* and *BRC* learn based on the same set of 24 different features. The values of these features for each sentence are used to compute the probability of the sentence being part of the summary.

The 24 features can be categorized into four major groups.

- *Structural* features are related to the conversational structure of the bug reports. Examples include the position of the sentence in the comment and the position of the sentence in the bug report.
- *Participant* features are directly related to the conversation participants. For example if the sentence is made by the same person who filed the bug report.
- *Length* features include the length of the sentence normalized by the length of the longest sentence in the comment and also normalized by the length of the longest sentence in the bug report.
- *Lexical* features are related to the occurrence of unique words in the sentence.

Table 3 provides a short description of the features considered. Some descriptions in the table refer to *Sprob*. Informally, *Sprob* provides the probability of a word being uttered by a particular participant based on the intuition that certain words will tend to be associated with one conversation participant due to interests and expertise. Other descriptions refer to *Tprob*, which is the probability of a turn given a word, reflecting the intuition that certain words will tend to cluster in a small number of turns because of shifting topics in a conversation. Full details on the features are provided in [10].

To see which features are informative for generating summaries of bug reports, we perform a feature selection

8. A sentence is selected as the last sentence of the summary if the 25 percent length threshold is reached in the middle or at the end of it.

9. www.csie.ntu.edu.tw/~cjlin/liblinear/, verified 04/04/12.

TABLE 3
Features Key

Feature ID	Description
MXS	max *Sprob* score
MNS	mean *Sprob* score
SMS	sum of *Sprob* scores
MXT	max *Tprob* score
MNT	mean *Tprob* score
SMT	sum of *Tprob* scores
TLOC	position in turn
CLOC	position in conversation
SLEN	word count, globally normalized
SLEN2	word count, locally normalized
TPOS1	time from beginning of conversation to turn
TPOS2	time from turn to end of conversation
DOM	participant dominance in words
COS1	cosine of conversation splits, w/ *Sprob*
COS2	cosine of conversation splits, w/ *Tprob*
PENT	entropy of conversation up to sentence
SENT	entropy of conversation after sentence
THISENT	entropy of current sentence
PPAU	time between current and prior turn
SPAU	time between current and next turn
BEGAUTH	is first participant (0/1)
CWS	rough ClueWordScore
CENT1	cosine of sentence & conversation, w/ *Sprob*
CENT2	cosine of sentence & conversation, w/ *Tprob*

analysis. For this analysis, we compute the F statistics score (introduced by Chen and Lin [32]) for each of the 24 features using the data in the bug report corpus. This score is commonly used to compute the discriminability of features in supervised machine learning. Features with higher F statistics scores are the most informative in discriminating between important sentences, which should be included in the summary, and other sentences, which need not be included in the summary.

Fig. 5 shows the values of F statistics computed for all the features in Table 3. The results show that the length features (SLEN and SLEN2) are among the most helpful features as longer sentences tend to be more descriptive. Several lexical features are also helpful: CWS,[10] CENT1, CENT2,[11] SMS[12] and SMT.[13] Some features have very low F statistics because either each sentence by a participant gets the same feature value (e.g., BEGAUTH) or each sentence in a turn gets the same feature value (e.g., TPOSE1). Although a particular feature may have a low F statistics score because it does not discriminate informative versus non-informative sentences on its own, it may well be useful in conjunction with other features [10].

The distribution of F statistics scores for the bug report corpus is different from those of the meeting and e-mail corpi [10]. For example, MXS and MXT have a relatively high value of F statistics for the e-mail data while both have a relatively low value of F statistics for the bug report data. Similarly SLEN2 has a relatively high F statistics score for the bug report data while it has a low value of F statistics for the meeting data. These differences further motivates training a new classifier using the bug report corpus as it may produce better results for bug reports compared to classifiers trained on meeting and e-mail data.

10. CWS measures the cohesion of the conversation by comparing the sentence to other turns of the conversation.

11. CENT1 and CENT2 measure whether the sentence is similar to the conversation overall.

12. SMS measures whether the sentence is associated with some conversation participants more than the others.

13. SMT measures whether the sentence is associated with a small number of turns more than the others.

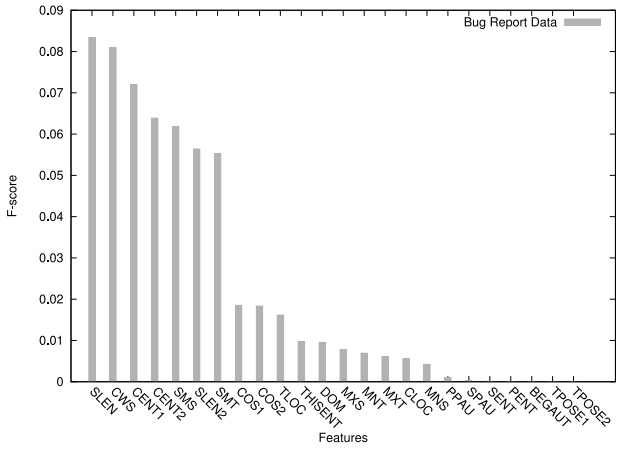

Fig. 5. Features F statistics scores for the bug report corpus.

5 Analytic Evaluation

To compare the EC, EMC and BRC classifiers, we use several measures that compare summaries generated by the classifiers to the gold standard summaries formed from the human annotation of the bug report corpus (Section 4). These measures assess the quality of each classifier and enable the comparison of effectiveness of the different classifiers against each other. In the next section, we report on a task-based human evaluation conducted to investigate the usefulness of summaries generated by a classifier in the context of a software task.

5.1 Comparing Base Effectiveness

The first comparison we consider is whether the EC, EMC and BRC classifiers are producing summaries that are better than a random classifier in which a coin toss is used to decide which sentences to include in a summary. The classifiers are compared to a random classifier to ensure they provide value in producing summaries. We perform this comparison by plotting the receiver operator characteristic (ROC) curve and then computing the area under the curve (AUROC) [33].

For this comparison we investigate different probability thresholds to generate extractive summaries. As described in Section 4, the output of the classifier for each sentence is a value between zero and one showing the probability of the sentence being part of the extractive summary. To plot a point of ROC curve, we first choose a probability threshold. Then we form the extractive summaries by selecting all the sentences with probability values greater than the probability threshold.

For summaries generated in this manner, we compute the false positive rate (FPR) and true positive rate (TPR), which are then plotted as a point in a graph. For each summary, TPR measures how many of the sentences present in gold standard summary are actually chosen by the classifier.

$$TPR = \frac{\#sentences\ selected\ from\ the\ GSS}{\#sentences\ in\ GSS}.$$

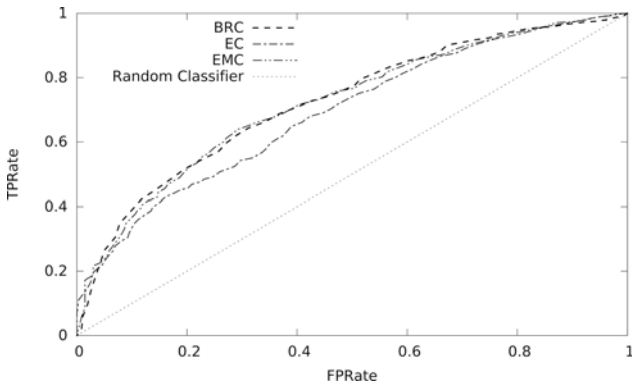

Fig. 6. ROC plots for BRC, EC and EMC classifiers.

FPR computes the opposite.

$$FPR = \frac{\#sentences\ selected\ that\ are\ not\ in\ the\ GSS}{\#sentences\ in\ the\ bug\ report\ that\ are\ not\ in\ the\ GSS}$$

The area under a ROC curve (AUROC) is used as a measure of the quality of a classifier. A random classifier has an AUROC value of 0.5, while a perfect classifier has an AUROC value of 1. Therefore, to be considered effective, a classifier's AUROC value should be greater than 0.5, preferably close to 1.

Fig. 6 shows the ROC curves for all the three classifiers. The diagonal line is representative of a random classifier. The area under the curve for BRC, EC and EMC is equal to 0.723, 0.691 and 0.721, respectively, indicating that all these classifiers provide comparable levels of improvement in efficiency over a random classifier.

5.2 Comparing Classifiers

To investigate whether any of EC, EMC or BRC work better than the other two based on our desired 25 percent word count summaries, we compared them using the standard evaluation measures of precision, recall, and f-score. We also used pyramid precision, which is a normalized evaluation measure taking into account the multiple annotations available for each bug report.

5.2.1 Precision, Recall and F-Score

F-score combines the values of two other evaluation measures: precision and recall. Precision measures how often a classifier chooses a sentence from the gold standard summaries (GSS) and is computed as follows:

$$precision = \frac{\#sentences\ selected\ from\ the\ GSS}{\#selected\ sentences}.$$

Recall measures how many of the sentences present in a gold standard summary are actually chosen by the classifier. For a bug report summary, the recall is the same as the TPR used in plotting ROC curves (Section 5.1).

As there is always a tradeoff between precision and recall, the F-score is used as an overall measure.

TABLE 4
Evaluation Measures

Classifier	Pyramid precision	Precision	Recall	F-score
BRC	.66	.57	.35	.40
EC	.55	.43	.30	.32
EMC	.54	.47	.23	.29

$$F\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

5.2.2 Pyramid Precision

The pyramid evaluation scheme by Nenkova and Passonneau [34] was developed to provide a reliable assessment of content selection quality in summarization where there are multiple annotations available. We used the pyramid precision scheme of Carenini et al. [35] inspired by Nenkova's pyramid scheme.

For each generated summary of a given length, we count the total number of times the sentences in the summary were linked by annotators. Pyramid precision is computed by dividing this number by the maximum possible total for that summary length. For example, if an annotated bug report has four sentences with three links and five sentences with two links, the best possible summary of length 5 has a total number of links equal to $(4 \times 3) + (1 \times 2) = 14$. The pyramid precision of a generated summary of length 5 with a total of eight links is therefore computed as

$$Pyramid\ Precision = \frac{8}{14} \approx 0.57.$$

5.2.3 Results

Table 4 shows the values of precision, recall, F-score, and pyramid precision for each classifier averaged over all the bug reports.

To investigate whether there is any statistically significant difference between the performance of the three classifiers, we performed six paired t-tests.[14] Table 5 shows the p-value for each individual test. These results confirm that the bug report classifier out-performs the other two classifiers (EC and EMC) with statistical significance (where significance occurs with $p < .025$).[15] There is no significant difference when comparing the performance of EC and EMC.

The results obtained for the EC and EMC classifiers are similar to those produced when the same classifiers are applied to meeting and e-mail data [10].

The results demonstrate that based on standard measures, while classifiers trained on other conversation-based data (EC and EMC) can generate reasonably good bug report summaries, a classifier specifically trained on bug report data (BRC) can generate summaries that are better with statistical significance.

14. The data conforms to the t-test normality assumption.

15. Since every two classifiers are compared based on two measures (F-score and pyramid precision), we used the Bonferroni correction [36] to adjust the confidence interval in order to account for the problem of multiple comparisons.

TABLE 5
Paired T-Tests Results

Tested measure	p-value
Pyramid precision	
Comparing BRC and EC	.00815
Comparing BRC and EMC	.00611
Comparing EC and EMC	.89114
F-score	
Comparing BRC and EC	.01842
Comparing BRC and EMC	.00087
Comparing EC and EMC	.28201

6 Task-Based Evaluation

According to the conventional measures of pyramid precision and F-score, bug report summaries generated by the BRC classifier are of reasonable quality. In a human evaluation we previously conducted, human judges agreed that bug report summaries mostly contained important points of original bug reports and excluded unnecessary information (a detailed description of this evaluation can be found in [4]). Yet still the question of whether summaries can help developers in performing software tasks remains to be answered. To investigate this question, we conducted a task-based evaluation of the usefulness of bug report summaries. The particular task we chose to investigate is bug report duplicate detection: determining whether a newly filed bug report is a duplicate of an existing report in a bug repository.

Bug report duplicate detection is performed when a new bug report is filed against a bug repository and has to be triaged and assigned to a developer. One step of triage is deciding whether the new bug report is a duplicate of one or more already in the repository. Early determination of duplicates can add information about the context of a problem and can ensure that the same problem does not end up being assigned to multiple developers to solve. Developers use different techniques to retrieve a list of potential duplicates from the bug repository including their memory of bugs they know about in the repository, keyword searches and machine learning and information retrieval approaches (e.g., [2], [15], [16]). In any approach other than memory-based approaches, a developer is presented a list of potential duplicate reports in the repository based on search or mining results. The developer must go over the list of retrieved potential duplicate bug reports to determine which one is a duplicate of the new report; this may require significant cognitive activity on the part of the developer as it might involve reading a lot of text both in description and comments of bug reports.

Our hypothesis is that concise summaries of original bug reports can help developers save time in performing duplicate detection tasks without compromising accuracy.

Our task-based evaluation of bug report summaries involved having 12 subjects complete eight duplicate detection tasks similar to real-world tasks under two conditions: *originals* and *summaries*. Each task involved a subject reading a new bug report and deciding for each bug report on a presented list of six potential duplicates whether it is a duplicate of the new bug report or not. All the bug reports used in the study were selected from the Mozilla bug repository. We scored the accuracy of a subject's determination of duplicates against information available in the bug repository.

6.1 Experimental Method

Each subject started the study session by working on two training tasks. Then the subject was presented with eight main tasks in random order. Half (four) of these tasks were performed under *originals* condition where the subject had access to potential duplicate bug reports in their original form. The other half were performed under *summaries* condition where the subject had access to 100-word summaries of potential duplicate bug reports, but not to their originals. Summaries were generated by the BRC classifier. As opposed to the 25 percent work count summaries used in the evaluation performed in Section 5, we decided to use fixed-size summaries in the task-based user evaluation to make summaries consistent in terms of size. We produced summaries of 100 words, a common size requirement for short paragraph-length summaries [3].

Fig. 7 shows a screenshot of the tool used by subjects to perform duplicate detection tasks. Based on the condition a task is performed under, clicking on the title of a potential duplicate in the top left window shows its corresponding bug report in original or summarized format in the right window. The new bug report can be viewed in the bottom left window.

For each task, the subject can mark a potential duplicate as *not duplicate*, *maybe duplicate*, or *duplicate*. To complete a task, a subject has to mark all potential duplicates and provide a short explanation for any *duplicate* or *maybe duplicate* marking. At most one bug report on the list of potential duplicates can be labeled as *duplicate*. We put this restriction because, based on information in the bug repository, for each task there is either zero or one actual duplicate on the list of potential duplicates. A subject can mark as many potential duplicate bug reports on the list as *maybe duplicate* or *not duplicate*.

Six out of eight new bug reports have an actual duplicate appearing on their corresponding list of potential duplicates. Subjects were not informed of this ratio and were only told that "There may or may not be a duplicate on the recommendation list."

Subjects were recommended to limit themselves to 10 minutes per task, but the time limit was not enforced. Each study session was concluded with a semi-structured interview.

All 12 users worked on the same set of eight duplicate detection tasks. Each task was performed under each condition (*summaries*, *originals*) by six different users. For each task, the users to whom the task was assigned under a particular condition (e.g., *summaries*) were randomly selected.

A number of questions in the form of a semi-structured interview were asked from each subject at the end of a study session (Table 6). During the interview, the subjects discussed the strategy they used in identifying duplicate bug reports. Subjects were asked to compare working with and without summaries in terms of time, difficulty, and having access to sufficient information.

Computing task completion accuracy: A reference solution for each bug duplicate detection task is available in the

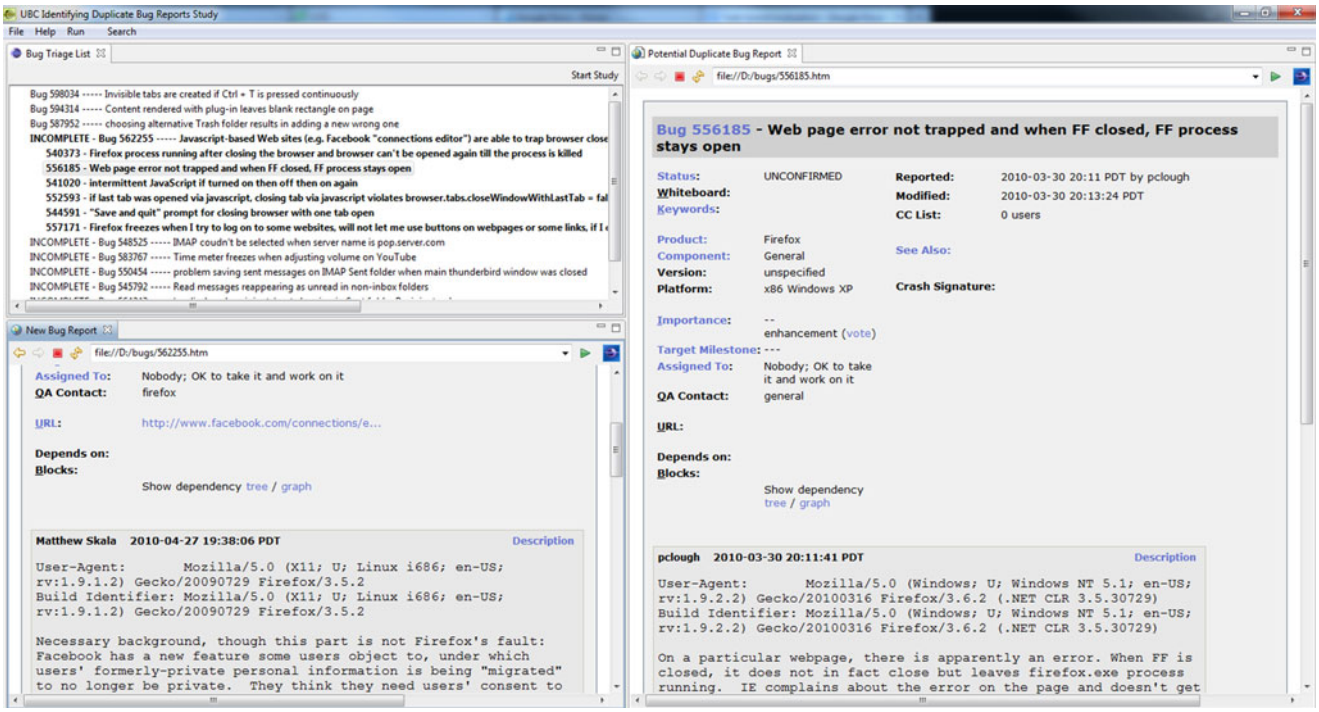

Fig. 7. The tool used by participants in the user study to perform duplicate detection tasks. The top left 'Bug Triage List' window shows the list of tasks, each consisting of a new bug report and six potential duplicate bug reports. The new bug report and the selected potential duplicate can be viewed in the bottom left window and the right window respectively.

project's bug repository. If the actual duplicate of the new bug report is among the six potential duplicates, the solution would be the actual duplicate marked as a *duplicate* and the other five marked as *not duplicate*s. To score a subject's solution for such a task, we compare the marking of each potential duplicate to the marking of the same bug report in the reference solution. If the markings are the same, we give a score of 1. If the potential duplicate is marked as a *maybe duplicate* (indicating insufficient information to make a decision) we give a score of 0.5. Otherwise we give a score of 0. To aggregate these scores to a single score for the task, we give a weight of five to the score of the actual duplicate. We chose to use this weighting scheme because we wanted the score of the actual duplicate to equally contribute to the final score as the scores of the other five potential duplicates. In this case the maximum score (that of the reference solution) would be 10 $((1 \times 5) + 1 + 1 + 1 + 1 + 1)$. The score of a solution in which the actual duplicate and one other bug report on the list are marked as a *maybe duplicate* and everything else is marked as a *not duplicate* is $(0.5 \times 5) + 0.5 + 1 + 1 + 1 + 1 = 7$. Finally the accuracy of each task is computed by dividing its score by the maximum score or the score of the reference solution which is 10. The accuracy of a task with a score of 7 (like the example above) would then be 0.7.

If the actual duplicate is not on the list, the reference solution would be all potential duplicates marked as *not duplicate*s. In this case, because there is no actual duplicate on the list, scores of potential duplicates have all the same weight in computing the total score of the task with 6 being the score of the reference solution.

6.2 Tasks

We used the recommender built by Sun et al. [2] to generate the list of potential duplicate bug reports. This recommender compares bug reports based on an extended version of BM25F, an effective textual similarity measure in information retrieval. The top six bug reports retrieved by this recommender for each new bug report form the list of potential duplicates for each task. The list is sorted with the most similar bug report at the top.

The new bug reports for the duplicate detection tasks (listed in Table 7) were randomly selected from the pool of all Thunderbird and Firefox bug reports filed in 2010 and labeled as DUPLICATE with the constraint that at least four out of six corresponding potential duplicate bug reports be longer than 300 words. We made this choice to

TABLE 6
Questions Asked of Each Participant at the End of a Study Session

Question
1. Did you find it easier or harder to detect duplicates given summaries for the potential duplicates? Why?
2. Did summaries change time spent on determining duplicates?
3. What kind of information in general do you think would be helpful in determining duplicates?
4. Did you find summaries to contain enough information to determine if they represented a duplicate?
5. Did you use attributes (e.g., version, platform, etc.)? How?

TABLE 7
New Bug Report Used for Each Task in the User Study

Task	Bug Report	Title	Product	Component
1	545792	Read messages reappearing as unread in non-inbox folders	Thunderbird	Folder and Message Lists
2	546095	Behavior with IMAP attachments utterly broken	Thunderbird	Message Compose Window
3	548525	IMAP coudn't be selected when server name is pop.server.com	Thunderbird	Account Manager
4	550454	problem saving sent messages on IMAP Sent folder when main thunderbird window was closed	Thunderbird	Message Compose Window
5	562255	Javascript-based Web sites (e.g. Facebook "connections editor") are able to trap browser close	Firefox	Security
6	564243	'undisclosed-recipients' not showing in Sent folder Recipient column	Thunderbird	Folder and Message Lists
7	583767	Time meter freezes when adjusting volume on YouTube	Firefox	General
8	587952	choosing alternative Trash folder results in adding a new wrong one	Thunderbird	Account Manager

ensure that the 100-word summaries are visibly shorter than most of the potential duplicate bug reports. The new bug reports have been drawn from different components of the projects. The diagram in Fig. 8 shows the distribution of length (in words) of potential duplicates (total number of potential duplicates: $8 \times 6 = 48$). The average length of a potential duplicate in our study is 650 words leading to an average compression rate of 15 percent for a summary length of 100 words.

Table 8 shows the list of potential duplicates for each task where '+' indicates an actual duplicate. Tasks 2 and 5 do not have an actual duplicate among the list of potential duplicates. The length of each potential duplicate, in the number of words, is shown next to it. The data in the table exhibits a recall rate of 75 percent (6/8: six out of eight tasks have the actual duplicate on the recommendation list) which exceeds the recall rate of all existing duplicate detection techniques that retrieve potential duplicates for a bug report based on its natural language content.

To generate a realistic task setting, for each bug report in Table 7, we used the date the bug report was created (filed) as a reference and reverted the bug report and all the six duplicate bug reports to their older versions on that date. All the comments that had been made after that date were removed. The attribute values (e.g., Status, Product, Component) were all reverted to the values on that date.

6.3 Participants

All 12 people recruited to participate in the study had at least 5 years (average: 9.9 ± 4) of experience in programming. This amount of programming experience helped them easily read and understand bug reports which often contain programming terms and references to code.

Participants had different amounts of programming experience in an industrial context. Five participants had 0-2 years, while the other seven had an average of 7.1 years of experience in programming in industry. The second group had an average of 5.8 years of experience working with issue tracking systems.

We decided to choose participants with different backgrounds because although people with more industrial experience may have better performance working on duplicate detection tasks, it is often novice people who are assigned as triagers for open bug repositories like Mozilla.

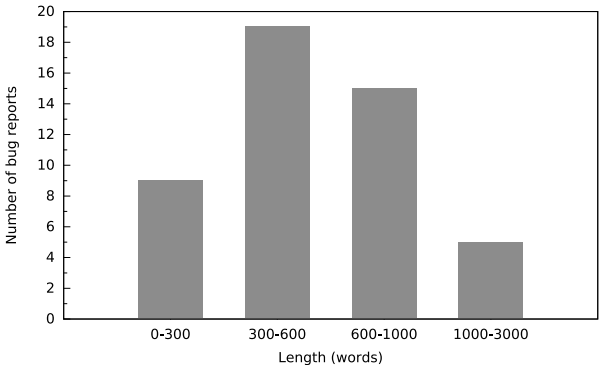

Fig. 8. The distribution of potential duplicates based on their length.

6.4 Results

Bug report summaries are intended to help a subject save time performing a bug report duplicate detection task by not having to interact with bug reports in their original format. At the same time it is expected that summaries contain enough information so that the accuracy of duplicate detection is not compromised. We investigated the following three questions using the data collected during the user study:

1) Do summaries, compared to original bug reports, provide enough information to help users accurately identify an actual duplicate?
2) Do summaries help developers save time working on duplicate detection tasks?
3) Do developers prefer working with summaries over original bug reports in duplicate detection tasks?

6.4.1 Accuracy

Table 9 shows the accuracy of performing each of the eight tasks by each of the 12 participants. The accuracy scores for tasks performed under the *summaries* condition have been marked with a '*'. The top diagram in Fig. 9 plots the accuracy of performing each task under each condition. In this figure, each accuracy value is the average of six individual accuracy scores corresponding to six different users performing the task under the same condition. On average (computed over all 48 corresponding non-starred accuracy scores in Table 9), the accuracy of performing a task under the *originals* condition is 0.752 (± 0.24) while the accuracy of performing a task under the *summaries* condition (computed over all 48 corresponding starred accuracy scores in Table 9) is 0.766 (± 0.23).

TABLE 8
List of Potential Duplicates per Task, Retrieved by Extended BM25F, '+' Indicates an Actual Duplicate

Task	1	2	3	4	5	6	7	8
Potential Duplicates	539035+ (612)	538803 (309)	538121 (452)	538340+ (209)	540373 (790)	540841 (381)	580795 (581)	558659 (1539)
	538756 (275)	544748 (432)	547812+ (465)	543508 (603)	556185 (204)	562782 (801)	571000 (537)	547682 (335)
	540846 (264)	543399 (723)	547530 (663)	543746 (640)	541020 (692))	549931+ (942)	578804+ (319)	542760+ (1071)
	544655 (1908)	545650 (523)	538115 (415)	549274 (871)	552593 (263)	542261 (167)	542639 (40)	547455 (2615)
	540289 (933)	539494 (827)	541256 (307)	544837 (543)	544591 (338)	550573 (2588)	571422 (656)	564173 (68)
	540914 (569)	541014 (408)	538125 (576)	540158 (997)	557171 (516)	541650 (246)	577645 (456)	539233 (849)

Numbers in parentheses show the length of each bug report.

TABLE 9
Accuracy of Performing Each Task by Each Participant, '*' Indicates *Summaries* Condition

	Participant											
Task	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}
1	0.45*	0.35*	0.45*	0.45	0.4	0.4*	0.95	0.5	0.4*	0.55	0.45*	0.4
2	0.92*	0.83	1*	1	1*	0.92	0.92	0.75*	0.83*	0.83	0.83	1*
3	0.75*	0.7	0.95	0.95*	1	0.45*	1*	0.95*	1*	0.75	1	1
4	0.95	1*	0.95	0.9*	1*	1	0.95	0.85	0.95	0.85*	1*	1*
5	1*	0.92	1*	1	1	1*	1	0.75*	0.75	0.83*	1	1*
6	0.45	0.5*	0.75*	0.4	0.45*	0.75	0.5*	0.45	0.7	0.45*	0.7*	0.5
7	0.5	0.5	0.5	0.5*	0.5	0.85*	0.75*	0.9*	0.75*	0.2	0.5*	0.5
8	0.95	1*	0.4	0.7*	0.75*	1	0.95*	0.75	1	0.35*	0.75	1*

We used a linear mixed effects regression model to model accuracy (dependent variable) as a function of the set of three independent variables (summary condition, task and participant). This model accounts for repeated measures in our study as we took multiple measurements per task and per participant. In the mixed effects model, summary condition is a fixed effects variable while task and participant are random effects variables. Wald chi-square test was applied to the mixed effects model to compute the p-value associated with the fixed effects variable (summary condition). The results ($\chi^2(1) = .2193$, $p > .63$) do not provide any evidence that there is a statistically significant change in accuracy when bug report summaries are used.

6.4.2 Time to Completion

In each study session, we measured the time to complete a task as the difference between the time the task is marked as completed and the time the previous task on the task list was marked as completed. Table 10 shows the time each participant took to complete each task with a '*' indicating the task was performed under the *summaries* condition. The bottom diagram in Fig. 9 plots the amount of time for each task under the two conditions averaged over the users. This plot shows that the 10-minute non-enforced time limit was enough for performing most of the bug report duplicate detection tasks in the study.

By comparing time to completion across the two conditions we were interested to know if summaries, by being shorter in length, help users to perform duplicate detection tasks faster. Based on our data, on average it took 8.21 (± 4.52) minutes to complete a task under the *originals* condition while it took 5.90 (± 2.27) minutes to perform a task under the *summaries* condition. Similar to our earlier analysis regarding accuracy, we used a linear mixed effects regression model to model time as a function of summary condition (fixed effects variable), task (random effects variable) and participant (random effects variable).[16] The result of Wald chi-square test ($\chi^2(1) = 14.629, p < .0005$) shows that the improvement in time under the *summaries* condition is statistically significant.

The comparison of accuracy scores and time to completion across the two conditions show that using summaries in duplicate detection tasks can help developers save time. It also shows that there is no evidence of any difference in accuracy of performing tasks across the two conditions.

6.4.3 Participant Satisfaction

Nine out of 12 (75 percent) participants (all except p_3, p_6, p_9) preferred working with summaries mentioning that it 'was less intimidating/daunting,' 'seemed more clear than full bug reports,' 'less noisy because they didn't have the header info,' 'made them relieved,' 'easier to read,' 'seemed to preserve what was needed' and 'had less cognitive overload.' The remaining three thought they needed to have originals to be sure of the decision they made, mentioning the ideal setting would be to have both summaries and originals available ('wanted to know what was left out from the summaries,' 'had problem with loss of context,' 'wanted to know who was talking').

Ten out of 12 (83 percent) participants thought they were faster while working with summaries, the other two (p_6 and p_{12}) mentioned that they felt time depended more on task difficulty than the task condition (*summaries* vs. *originals*).

Seven out of 12 (58 percent) participants (p_1, p_3, p_6, p_8, p_{10}, p_{11}, p_{12}) mentioned that the 'steps to reproduce,' was probably the most important piece of information in comparing two bug reports and deciding if they were duplicates. One way to further improve the quality of produced summaries is by including in the summary an indication of the the

16. For both accuracy and time data, visual inspection of residual plots did not reveal any obvious deviations from homoscedasticity or normality.

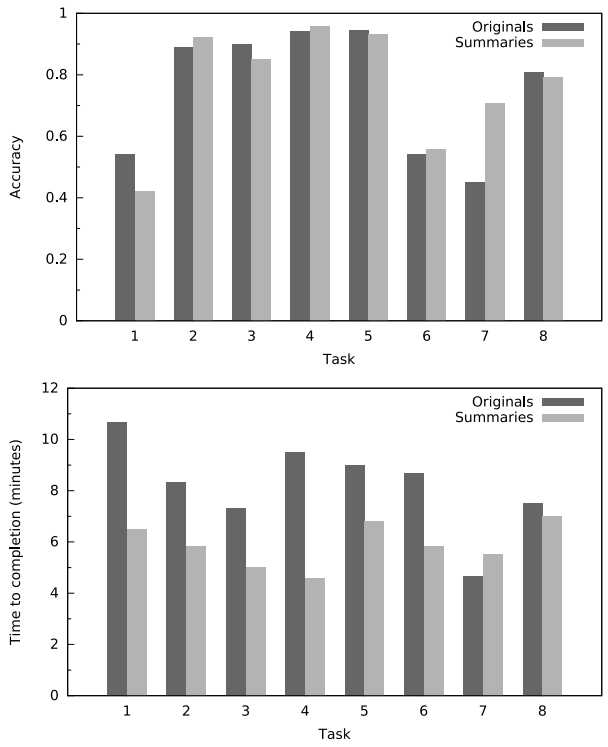

Fig. 9. The average accuracy and time of performing each duplicate detection task under each condition (original bug reports, bug report summaries).

availability of this information in the original bug report (see Section 7).

6.5 Threats

One of the primary threats to the internal validity of the evaluations we have conducted is the annotation of the bug report corpus by non-experts in the projects. Optimally, we would have had summaries created for the reports by experts in the projects. Summaries created by experts might capture the meaning of the bug reports better than was possible by non-experts. On the other hand, summaries created by experts might rely on knowledge that was not in the bug reports, potentially creating a standard that would be difficult for a classifier to match. By assigning three annotators to each bug report and by using agreement between two to form the gold standard summaries, we have attempted to mitigate the risk of non-expert annotators.

The use of non-project experts in the task-based evaluation is a threat to the external validity of our results. This threat has been mitigated by using systems (e.g., Firefox and Thunderbird) that participants were familiar with at least from a user perspective. The other main threat is the use of general purpose summaries for a particular task. Optimally, different summaries should exist for different types of tasks involving bug reports. For instance, a bug report duplicate detection task more likely needs more information on the symptoms of the problem rather than how it should be fixed.

The bug reports in the corpus and the bug reports used in the task-based user study have been chosen to be representative of the intended target of the approach (i.e., lengthy bug reports). There is an external threat that the approach does not apply to bug repositories with mostly shorter reports.

The other threat is the use of different summary lengths for the evaluation of summaries. Summaries as long as 25% of original bug reports were used for the analytic evaluation because we wanted to match the length of gold standard summaries. We generated paragraph-length summaries (100 words) for the task-based evaluation to make them visibly shorter compared to the original bug reports and easier to interact with for a participant that has to go over a lot of bug report summaries in the course of a study session.

7 Discussion

With this work, we have shown that it is possible to generate summaries for a diverse set of bug reports with reasonable accuracy. We also showed that summaries were helpful in the context of duplicate detection tasks. In this section, we discuss possible ways to improve the summaries produced and to evaluate their usefulness.

7.1 Improving the Bug Report Summarizer

In this paper, we used a summarization technique developed for generic conversations to automatically summarize bug reports. We also showed that bug reports summaries generated by this technique help developers. Given this initial evidence, future efforts may focus more on domain-specific features of bug reports to improve generated summaries.

For example, the accuracy of the bug report summarizer may be improved by augmenting the set of generic features with domain-specific features. For instance, comments made by people who are more active in the project might be more important, and thus should be more likely included in the summary. As another example, it was noted by many of the participants in our task-based evaluation that 'steps to

TABLE 10
Time (in Minutes) to Complete Each Task by Each Participant, '*' Indicates *Summaries* Condition

Task	Participant											
	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}
1	7*	8*	8*	13	11	4*	15	12	5*	6	7*	7
2	8*	12	7*	12	4*	3	9	4*	6*	3	11	6*
3	5*	8	9	6*	10	4*	4*	4*	7*	4	4	9
4	11	4*	21	9*	4*	2	7	6	10	5*	2*	4*
5	9*	14	7*	13	6	4*	7	11*	9	6*	5	4*
6	6	9*	5*	24	8*	6	6*	3	10	3*	4*	3
7	7	4	7	11*	3	6*	5*	6*	2*	5	3*	2
8	5	6*	10	12*	8*	7	5*	10	7	5*	6	6*

reproduce' in a bug report description helped them in determining duplicates. This information has also been identified as important by developers in a separate study [18]. The usefulness of bug report summaries might be improved if an indication of the availability of 'steps to reproduce' information be included in the summary so that a developer can refer to the original bug report in cases she needs this information in addition to the summary.

As another example, as part of the annotation process, we also gathered information about the intent of sentences, such as whether a sentence indicated a 'problem,' 'suggestion,' 'fix,' 'agreement,' or 'disagreement.' Similar to the approach proposed by Murray et al. [37] in using speech acts, this information can be used to train classifiers to map sentences of a bug report to appropriate labels. Then an abstractive summary can be generated by identifying patterns that abstract over multiple sentences. We leave the investigation of generating such abstractive summaries to future research.

7.2 Task-Based Evaluation of Bug Report Summaries

The ultimate goal in development of any summarization system is to help the end user perform a task better. We used a duplicate detection task setting to evaluate the usefulness of bug report summaries in this paper. Other settings are possible and can be evaluated in the future. For example, previous work in our research group developed a recommender to suggest to a developer performing an evolution task on a system, change tasks (described by bug reports) that have been previously completed which might guide the developer [1]. As with the duplicate bug case, the developer must again wade through a substantial amount of text to determine which bug report might be relevant. Summaries could also ease this process.

There are a number of other alternatives that could be considered in the task-based evaluation of bug report summaries. One option that was suggested by a few participants in our study was to make both summaries and original available to them. The other option is to allow participants to adjust the length of summaries so that they could add more content to a summary if they felt they needed more information to make a decision. In the future studies, it would also be useful to compare the performance of developers on a task with access only to bug report descriptions to their performance while having access to automatically generated summaries.

7.3 Other Forms of Evaluation

Other forms of evaluation than software tasks are also possible that are less expensive to use to evaluate bug report summaries. For example, bug reports summaries could be evaluated in the context of comprehension or question answering activities. This style of evaluation is very common in the natural language processing community where numerous task-based evaluations have been performed to establish the effectiveness of summarization systems in a variety of tasks. The tasks range from judging if a particular document is relevant to a topic of interest [38], writing reports on a specific topic [39], finding scientific articles related to a research topic [40] and decision audit (determining how and why a given decision was made) [41]. While using a comprehension activity to evaluate bug report summaries is less expensive, comprehension is not the end goal of software developers interacting with bug reports.

8 Summary

Researchers rely on good summaries to be available for papers in the literature. These summaries are used for several purposes, including providing a quick review of a topic within the literature and selecting the most relevant papers for a topic to peruse in greater depth. Software developers must preform similar activities, such as understanding what bugs have been filed against a particular component of a system. However, developers must perform these activities without the benefit of summaries, leading them to either expend substantial effort to perform the activity thoroughly or resulting in missed information.

In this paper, we have investigated the automatic generation of one kind of software artifact, bug reports, to provide developers with the benefits others experience daily in other domains. We found that existing conversation-based extractive summary generators can produce summaries for reports that are better than a random classifier. We also found that an extractive summary generator trained on bug reports produces the best results. We showed that generated bug report summaries could help developers perform duplicate detection tasks in less time with no indication of accuracy degradation, confirming that bug report summaries help software developers in performing software tasks.

Acknowledgments

The authors would like to thank Chengnian Sun for providing them with the code for the BM25F-based duplicate bug report recommender. This work was funded by NSERC.

References

[1] D. Čubranić and G.C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," *Proc. 25th Int'l Conf. Software Eng. (ICSE '03)*, pp. 408-418, 2003.

[2] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards More Accurate Retrieval of Duplicate Bug Reports," *Proc. 26th Int'l Conf. Automated Software Eng. (ASE '11)*, pp. 253-262, 2011.

[3] P. Over, H. Dang, and D. Harman, "DUC in Context," *Information Processing and Management: An Int'l J.*, vol. 43, no. 6, pp. 1506-1520, 2007.

[4] S. Rastkar, G.C. Murphy, and G. Murray, "Summarizing Software Artifacts: A Case Study of Bug Reports," *Proc. 32nd Int'l Conf. Software Eng. (ICSE '10)*, vol. 1, pp. 505-514, 2010.

[5] A. Nenkova and K. McKeown, "Automatic Summarization," *Foundations and Trends in Information Retrieval*, vol. 5, no. 2/3, pp. 103-233, 2011.

[6] K. Zechner, "Automatic Summarization of Open-Domain Multiparty Dialogues in Diverse Genres," *Computational Linguistics*, vol. 28, no. 4, pp. 447-485, 2002.

[7] X. Zhu and G. Penn, "Summarization of Spontaneous Conversations," *Proc. Ninth Int'l Conf. Spoken Language Processing (Interspeech '06-ICSLP)*, pp. 1531-1534, 2006.

[8] O. Rambow, L. Shrestha, J. Chen, and C. Lauridsen, "Summarizing Email Threads," *Proc. Human Language Technology Conf. North Am. Chapter of the Assoc. for Computational Linguistics (HLT-NAACL '04)*, 2004.

[9] S. Wan and K. McKeown, "Generating Overview Summaries of Ongoing Email Thread Discussions," *Proc. 20th Int'l Conf. Computational Linguistics (COLING '04)*, pp. 549-556, 2004.

[10] G. Murray and G. Carenini, "Summarizing Spoken and Written Conversations," *Proc. Conf. Empirical Methods on Natural Language Processing (EMNLP '08)*, 2008.

[11] J. Anvik, L. Hiew, and G.C. Murphy, "Coping with an Open Bug Repository," *Proc. OOPSLA Workshop Eclipse Technology eXchange*, pp. 35-39, 2005.
[12] J. Anvik, L. Hiew, and G.C. Murphy, "Who Should Fix This Bug?" *Proc. 28th Int'l Conf. Software Eng. (ICSE '06)*, pp. 361-370, 2006.
[13] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate Bug Reports Considered Harmful; Really?" *Proc. IEEE 24th Int'l Conf. Software Maintenance (ICSM '08)*, pp. 337-345, 2008.
[14] J. Davidson, N. Mohan, and C. Jensen, "Coping with Duplicate Bug Reports in Free/Open Source Software Projects," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '11)*, pp. 101-108, 2011.
[15] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," *Proc. 29th Int'l Conf. Software Eng. (ICSE '07)*, pp. 499-510, 2007.
[16] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," *Proc. 30th Int'l Conf. Software Eng. (ICSE '08)*, pp. 461-470, 2008.
[17] A.J. Ko, B.A. Myers, and D.H. Chau, "A Linguistic Analysis of How People Describe Software Problems," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing (VL-HCC '06)*, pp. 127-134, 2006.
[18] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What Makes a Good Bug Report?" *Proc. 16th Int'l Symp. Foundations of Software Eng. (FSE '08)*, pp. 308-318, 2008.
[19] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information Needs in Bug Reports: Improving Cooperation between Developers and Users," *Proc. ACM Conf. Computer Supported Cooperative Work (CSCW '10)*, pp. 301-310, 2010.
[20] R.J. Sandusky and L. Gasser, "Negotiation and the Coordination of Information and Activity in Distributed Software Problem Management," *Proc. Int'l ACM SIGGROUP Conf. Supporting Group Work (GROUP '05)*, pp. 187-196, 2005.
[21] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, Collaboration, and Bugs: The Social Nature of Issue Tracking in Small, Collocated Teams," *Proc. ACM Conf. Computer Supported Cooperative Work (CSCW '10)*, pp. 291-300, 2010.
[22] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'Hurried' Bug Report Reading Process to Summarize Bug Reports," *Proc. IEEE 28th Int'l Conf. Software Maintenance (ICSM '12)*, pp. 430-439, 2012.
[23] S. Mani, R. Catherine, V.S. Sinha, and A. Dubey, "AUSUM: Approach for Unsupervised Bug Report Summarization," *Proc. ACM SIGSOFT 20th Int'l Symp. the Foundations of Software Eng. (FSE '12)*, article 11, 2012.
[24] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," *Proc. 17th Working Conf. Reverse Eng. (WCRE '10)*, pp. 35-44, 2010.
[25] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," *Proc. 25th Int'l Conf. Automated Software Eng. (ASE '10)*, pp. 43-52, 2010.
[26] L. Moreno, J. Aponte, G. Sridhara, A Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," *Proc. IEEE 21st Int'l Conf. Program Comprehension (ICPC '13)*, 2013.
[27] S. Rastkar, G.C. Murphy, and A.W.J. Bradley, "Generating Natural Language Summaries for Crosscutting Source Code Concerns," *Proc. IEEE 27th Int'l Conf. Software Maintenance (ICSM '11)*, pp. 103-112, 2011.
[28] J. Carletta, S. Ashby, S. Bourban, M. Flynn, M. Guillemot, T. Hain, J. Kadlec, V. Karaiskos, W. Kraaij, M. Kronenthal, G. Lathoud, M. Lincoln, A. Lisowska, I. McCowan, W Post, D. Reidsma, and P. Wellner, "The AMI Meeting Corpus: A Pre-Announcement," *Proc. Second Int'l Workshop Machine Learning for Multimodal Interaction (MLMI '05)*, pp. 28-39, 2005.
[29] J. Fleiss, "Measuring Nominal Scale Agreement Among Many Raters," *Psychological Bull.*, vol. 76, no. 5, pp. 378-382, 1971.
[30] B. Klimt and Y. Yang, "Introducing the Enron Corpus," *Proc. First Conf. E-mail and Anti-Spam (CEAS '04)*, 2004.
[31] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A Library for Large Linear Classification," *J. Machine Learning Research*, vol. 9, pp. 1871-1874, 2008.
[32] Y.-W. Chen and C.-J. Lin, "Combining SVMs with Various Feature Selection Strategies," *Feature Extraction, Foundations and Applications*, pp. 315-324, Springer, 2006.
[33] T. Fawcett, "ROC Graphs: Notes and Practical Considerations for Researchers," technical report HP Laboratories, 2004.
[34] A. Nenkova and R. Passonneau, "Evaluating Content Selection in Summarization: The Pyramid Method," *Proc. Human Language Technology Conf. North Am. Chapter of the Assoc. for Computational Linguistics (HLT-NAACL '04)*, pp. 145-152, 2004.
[35] G. Carenini, R.T. Ng, and X. Zhou, "Summarizing Emails with Conversational Cohesion and Subjectivity," *Proc. 46th Ann. Meeting of the Assoc. for Computational Linguistics: Human Language Technologies (ACL-08-HLT)*, pp. 353-361, 2008.
[36] H. Abdi, "Bonferroni and Sidak Corrections for Multiple Comparisons," *Encyclopedia of Measurement and Statistics*, N.J. Salkind, ed., Sage Publications, 2007.
[37] G. Murray, G. Carenini, and R. Ng, "Generating and Validating Abstracts of Meeting Conversations: A User Study," *Proc. Sixth Int'l Natural Language Generation Conf. (INLG '10)*, pp. 105-113, 2010.
[38] I. Mani, D. House, G. Klein, L. Hirschman, T. Firmin, and B. Sundheim, "The TIPSTER SUMMAC Text Summarization Evaluation," *Proc. Ninth Conf. European Chapter of the Assoc. for Computational Linguistics (EACL '99)*, pp. 77-85, 1999.
[39] K. McKeown, R.J. Passonneau, D.K. Elson, A. Nenkova, and J. Hirschberg, "Do Summaries Help?" *Proc. 28th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR '05)*, pp. 210-217, 2005.
[40] S. Teufel, "Task-Based Evaluation of Summary Quality: Describing Relationships between Scientific Papers," *Proc. Automatic Summarization Workshop North Am. Chapter of the Assoc. for Computational Linguistics (NAACL-AutoSum '01)*, pp. 12-21, 2001.
[41] G. Murray, T. Kleinbauer, P. Poller, T. Becker, S. Renals, and J. Kilgour, "Extrinsic Summarization Evaluation: A Decision Audit Task," *ACM Trans. Speech and Language Processing*, vol. 6, no. 2, article 2, Oct. 2009.

Sarah Rastkar received the BSc and MSc degrees from Sharif University of Technology in 2001 and 2003, and the PhD degree form the University of British Columbia in 2013. She is a postdoctoral fellow in the Department of Computer Science at the University of British Columbia. Her main research interests include human-centric software engineering with an emphasis on ways software developers interact with data to address their information needs.

Gail C. Murphy received the BSc degree in computing science from the University of Alberta in 1987, and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she was a software designer in industry. She is currently a professor in the Department of Computer Science at the University of British Columbia. Her research interests include software developer productivity and software evolution. She is a member of the IEEE Computer Society.

Gabriel Murray is an assistant professor at the University of the Fraser Valley, Chilliwack, BC, Canada. He teaches programming and artificial intelligence, and researches in the area of natural language processing. Gabriel is co-author of the book Methods for Mining and Summarizing Text Conversations.