Summarizing Software Artifacts: A Case Study of Bug Reports

Sarah Rastkar, Gail C. Murphy and Gabriel Murray
Department of Computer Science
University of British Columbia
{rastkar,murphy,gabrielm}@cs.ubc.ca

ABSTRACT

Many software artifacts are created, maintained and evolved as part of a software development project. As software developers work on a project, they interact with existing project artifacts, performing such activities as reading previously filed bug reports in search of duplicate reports. These activities often require a developer to peruse a substantial amount of text. In this paper, we investigate whether it is possible to summarize software artifacts automatically and effectively so that developers could consult smaller summaries instead of entire artifacts. To provide focus to our investigation, we consider the generation of summaries for bug reports. We found that existing conversation-based generators can produce better results than random generators and that a generator trained specifically on bug reports can perform statistically better than existing conversation-based generators. We demonstrate that humans also find these generated summaries reasonable indicating that summaries might be used effectively for many tasks.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]

General Terms

Experimentation

Keywords

Machine Learning, Human-centric Software Engineering

1. INTRODUCTION

Individuals outside the profession of software development sometimes incorrectly believe that the profession is all about programming. Those involved in software development know that the profession has a strong component of information management. Any successful large and complex software system requires the creation and management of many artifacts: requirements, designs, bug reports, and source code with embedded documentation to name just a few.

To perform work on the system, a software developer must often read and understand artifacts associated with the system development. For example, a developer attempting to fix a performance bug on a system may be told that a similar bug was solved six months ago. Finding the bug report that captured the knowledge about what was fixed will likely require the developer to perform searches and read several bug reports in search of the report of interest. Each report read may contain several sentences of description as well as tens of sentences representing discussion amongst team members. For example, bug #491925 from the Mozilla system[1] comprises 91 sentences: 7 in the description and 84 sentences from 24 comments. If solving the bug requires an understanding of the Java library `WeakHashMap` class, another 21 sentences from documentation about that class will need to be read. Wading through text comprising system artifacts to determine which ones matter can be time consuming and frustrating for the developer. Sometimes, the amount of information may be overwhelming, causing searches to be abandoned and duplicate or non-optimized work to be performed, all because the previous history of the project has been ignored.

One way to reduce the time a developer spends getting to the right artifacts to perform their work is to provide a summary of each artifact. An accurate summary can enable a developer to reduce the time spent perusing artifacts that have been returned from searches, found through browsing or recommended by team members or tools. Perhaps optimally, the authors of system artifacts would write a suitable abstract to help other developers working on the system. Given the evolving nature of artifacts and the limited time available to developers, this optimal path is not likely to occur.

Alternatively, it might be possible to generate summaries of project artifacts, saving developers effort and enabling up-to-date summaries on-demand. In this paper, we investigate the possibility of automatic summary generation, focusing on one kind of project artifact, bug reports, to make the investigation tractable. We chose to focus on these reports as there are a number of cases in which developers may make use of existing bug reports, such as when triaging bugs (e.g., [3]) or when performing change tasks (e.g., [18]), and these reports can often be lengthy, involving discussions amongst multiple team members.

[1] `mozilla.org`, verified 05/09/05

ICSE '10, May 2-8 2010, Cape Town, South Africa

The problem of summarizing discussions amongst multiple people is similar to the problem of summarizing email and meeting discussions (Section 2). We build on existing work in this area, investigating whether existing machine-learning based approaches (classifiers) for generating extractive summaries can produce accurate summaries for bug reports. These approaches assign a zero or one value to each sentence in the bug report based on features of the sentence; sentences assigned a one value appear in a generated summary.

To enable this investigation, we had human annotators create summaries for 36 bug reports, creating a corpus of bug report data (Section 3). These bug reports included both reports about defects and reports about enhancements. We then applied existing classifiers trained on email, and email and meeting data to produce summaries for reports in the bug report corpus. We also trained a classifier specific to bug reports and applied it to the bug report corpus (Section 4).Our bug report corpus serves as the training set for the bug report classifier and the testing set for all three classifiers.

We measured the effectiveness of these classifiers based on several computed measures (Section 5). We found that all three classifiers perform well above the chance level (a random classifier). We also found that the bug report classifier, having a precision of more than 62%, out-performs the other two classifiers in generating summaries of bug reports. To evaluate whether a measure of 62% produces summaries useful for developers, we had human judges evaluate the goodness of a subset of the summaries produced by the bug report classifier. On a scale of 1 (low) to 5 (high), the arithmetic mean quality ranking of the generated summaries by the human judges is 3.69 ($\pm$ 1.17). This average rating suggests that the generated summaries can provide a sufficiently accurate summary for developers, saving them the need in all cases to read lengthy bug reports.

This paper makes three contributions.

- It demonstrates that it is possible to generate accurate summaries for one kind of project artifact, bug reports.
- It reports on the creation of a corpus of 36 human annotated bug report data 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.

2. RELATED WORK

With the growing number of electronically available information, there is substantial interest and a substantial body of work in the automated generation of summaries for that information. Two basic approaches have been taken to generating summaries: extractive and abstractive [10]. 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. As the current state-of-the-art in abstractive techniques has not yet supported meaningful application, we focus in this paper on extractive techniques.

Extractive-based summarization approaches have been applied to many kinds of conversations, including meetings [21], telephone conversations [22] and emails [16]. Some of these approaches have used domain-specific characteristics, such as email header information [19]. Murray and Carenini [13] developed a summarizer for conversations in various modalities that uses features inherent to all multi-party conversations. They applied this system to meetings and emails 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 (or good) summaries of bug reports.

While the format of bug reports vary 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[2], the description and comments may be written but not edited, further adding to the conversation nature of a report. This lack of editable text also means the descriptions do not serve as summaries of the current report contents. Figure 1 displays part of a bug report from the KDE Bugzilla bug repository[3] with 21 comments from 6 people; the description and each comment can be considered a turn in a conversation.

As bug repositories 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 [1]. For instance, Anvik and colleagues [2] have shown how to provide recommenders for whom to assign a report whereas Runeson and colleagues [17] and Wang and colleagues [20] have demonstrated how to detect duplicate reports. While some of this work has applied classifiers to the problems they intend to solve, none has attempted to extract a meaningful summary for developers.

Other efforts have considered how to improve the content of bug reports. Ko and colleagues analyzed the titles of bug reports to inform the development of tools for both reporting and analyzing bugs [12]. Bettenburg and colleagues 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 [4]. Developers in their study noted that bug reports "are often used to debate the relative importance of various issues"; it is this conversational nature of reports that makes summaries valuable and that we hope to exploit to produce the summaries.

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

[2] `www.bugzilla.org`, verified 05/09/09
[3] `www.kde.org`, verified 05/09/05

Bug 188311 - The applet panel should not overlap applets

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

Votes: 0

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.
```

Figure 1: An example of the conversational structure of a bug report (The beginning part of bug 188311 from the KDE bug repository).

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,[4], Gnome,[5], Mozilla and KDE. We chose a diverse set of systems because our goal is to develop a summarization framework that can produce accurate results for a wide range of bug repositories, not just bug reports specific to a single project. 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.

The 36 bugs reports (nine from each project) chosen for annotation have mostly conversational content. We avoided selecting bug reports consisting mostly of long stack traces

[4]`www.eclipse.org`, verified 04/09/09
[5]`www.gnome.org`, verified 04/09/09

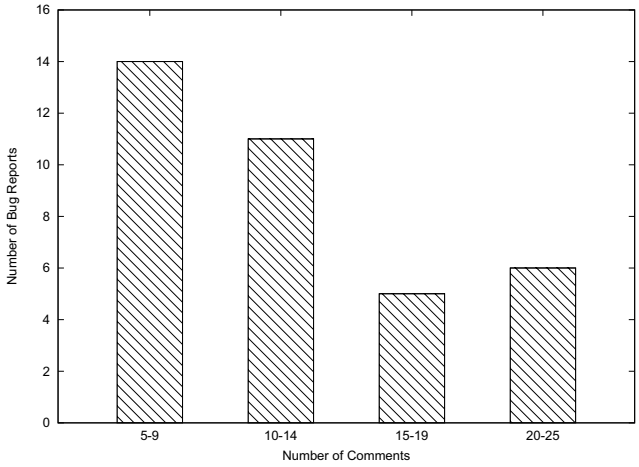

Figure 2: The distribution of bug reports in the corpus with regard to the number of comments.

and large chunks of code as this content may be used but is not typically read by developers. The reports chosen varied in length: 25 reports (69%) had between five and fourteen comments; the remaining eleven bugs (31%) had 15 to 25 comments each. Figure 2 shows the distribution of bug reports based on the number of comments. Nine of the 36 bug reports (25%) were enhancements to the target system; the other 27 (75%) were defects. There are a total of 2361 sentences in these 36 bug reports. This corpus size is similar to that used in training an email classifier; the email corpus contains 39 email threads and 1400 sentences [13].

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.

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 bug report. Figure 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 aver-

[6]`www.cs.ubc.ca/nest/lci/bc3/framework.html`, verified 04/09/09
[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.

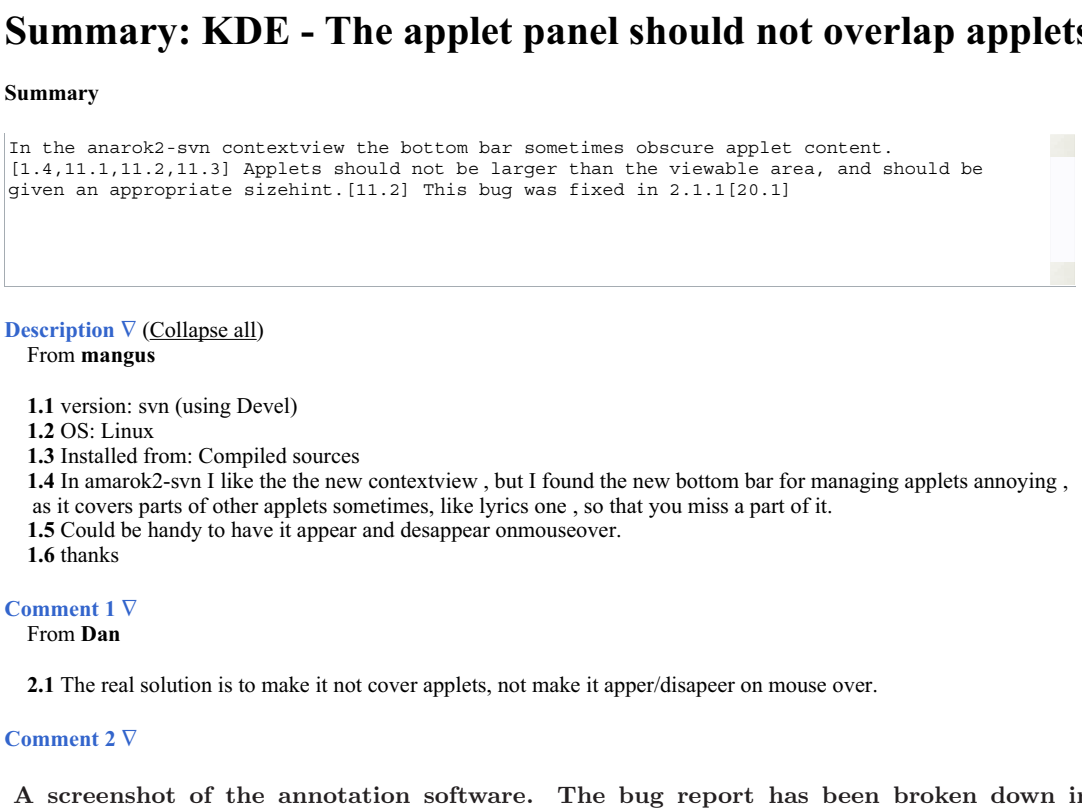

Figure 3: A screenshot of the annotation software. The bug report has been broken down into labeled sentences. The annotator enters the abstractive summary in the text box. The numbers in the brackets are sentence labels and serve as links between the abstractive summary and the bug report. For example,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

Table 1: 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

age) to three sentences in the original bug report. Table 1 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 between the annotators [9]. 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 us a basis on which to experiment with producing bug report summaries automatically. We set out to investigate two questions:

1. Can we produce good summaries with existing conversation-based classifiers?
2. How much better can we do 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 email threads [13]. We chose this classifier as bug report conversations share similarity with email 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 email corpus [11], which consists of 39 annotated email threads (1400 sentences in total).

The second classifier, which we refer to as *EMC*, was trained on a combination of email threads and meetings [13]. 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 sub-

set of the publicly available AMI meeting corpus [6], 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 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*. For the bug report corpus, gold standard summaries include 465 sentences, which is 19.7% of all the sentences in the corpus, and 28.3% of all words in the corpus.

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% of the bug report word count. The selected sentences form the generated extractive summary. We chose to target summaries of 25% of the bug report word count because this value is close to the word count percentage of gold standard summaries (28.3%). All three classifiers were implemented using the Libliner toolkit.[8]

4.1 Conversation Features

The classifier framework used to implement EM, EMC and BRC can learn based on 24 different features. We hypothesize that the features useful for the EM and EMC classifiers are also relevant to the summarization of bug reports since these reports exhibit a conversational structure. The features are based on representing a bug report as a conversation comprised of turns between multiple participants.

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.

The detailed description of features can be found in [13].

5. EVALUATION

To compare the EC, EMC and BRC classifiers, we have used 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.

However, these measures do not tell us whether the intended end users of the summaries—software developers—consider the generated summaries as representative of the original bug report. To check the performance of the classifiers from a human perspective, we also report on an evaluation in which we asked human judges to evaluate the goodness of a set of generated summaries against the original bug reports.

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. We perform this comparison by plotting the receiver operator characteristic (ROC) curve and then computing the area under the curve (AUROC) [8].

For this comparison, instead of using the 25% word count to generate extractive summaries, we investigate different probability thresholds. 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 (GSS) are actually chosen by the classifier.

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

FPR computes the opposite.

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

[8] `www.csie.ntu.edu.tw/~cjlin/liblinear/`, verified 04/09/09

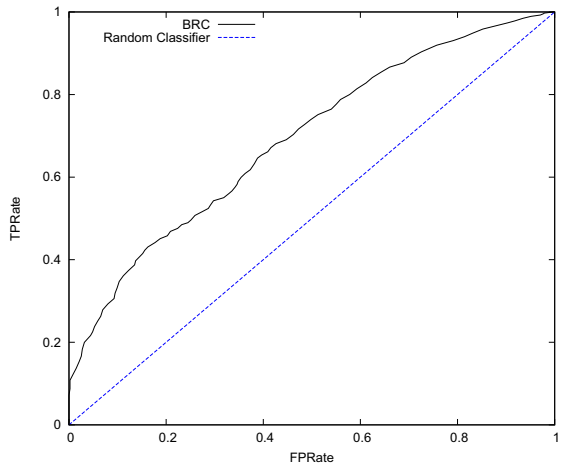

Figure 4: ROC plot for BRC classifier.

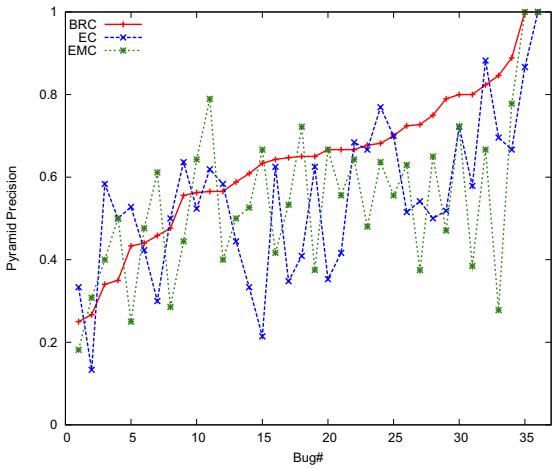

Figure 5: Pyramid precision for all classifiers.

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 somewhere in between, preferably close to 1.

Figure 4 shows the ROC curve for the BRC classifier. The diagonal line is representative of a random classifier. The area under the curve (AUROC) for BRC is equal to 0.722, indicating that this classifier performs better than a random classifier. We also computed the AUROC values for EC and EMC. The values are 0.719 and 0.689 respectively suggesting that each of the classifiers has the same level of efficiency compared to a random classifier.

5.2 Comparing Classifiers

AUROC is a measure of the general effectiveness of the classifiers. When the summaries are generated with a predefined length—25% in this paper—we need other measures to compare the classifiers.

To investigate whether any of EC, EMC or BRC work better than the other two based on our desired 25% 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 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 trade-off between precision and recall, the F-score is used as an overall measure.

$$Fscore = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Figure 6 shows the values of F-score for the three classifiers across all the bug reports. In this plot, the bug reports have been sorted based on the values of the F-score for the summaries generated by BRC. This figure shows that the best F-score typically occurs with the BRC classifier.

5.2.2 Pyramid Precision

The pyramid evaluation scheme by Nenkova and Passonneau [15] 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 [5] 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 4 sentences with 3 links and 5 sentences with 2 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 8 links is therefore computed as

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

Figure 5 shows the values of pyramid precision for the three classifiers across all the bug reports in the corpus. The bug reports have been sorted based on the values of the pyramid precision for the summaries generated by BRC. The figure shows that BRC has better precision values for most of the bug reports.

5.2.3 Summary

Figure 5 and Figure 6 show that BRC out-performs the other two classifiers for most of the bug reports. Table 2 shows the values of precision, recall, F-score, and pyramid precision for each classifier averaged over all the bug reports. To investigate whether the bug report classifier (BRC) is

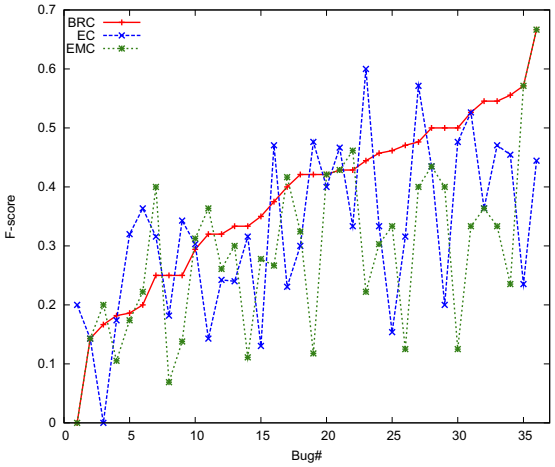

Figure 6: F-score plot for all classifiers.

Table 2: Evaluation measures.

Classifier	Pyramid Precision	Precision	Recall	F-Score
BRC	.63	.57	.35	.4
EC	.54	.43	.3	.32
EMC	.53	.47	.23	.29

significantly better than the other two classifiers (*EC* and *EMC*), we performed four paired t-tests: one to see if the pyramid precision of *BRC* is significantly better that that of *EC*, one to see if the pyramid precision of *BRC* is significantly better that that of *EMC*, and so on. The results confirm that the bug report classifier (*BRC*) out-performs the other two classifiers with statistical significance (where significance occurs with $p < .05$).

The classifier trained on emails (*EC*) and the classifier trained on emails and meetings (*EMC*) have similar performance. The paired t-test confirms that there is no significant difference.

5.3 Feature Selection Analysis

All of the three classifiers used in our study, *EC*, *EMC*, and *BRC*, use a set of 24 features to generate summaries of bug reports. The values of these features for each sentence are used to compute the probability of the sentence being part of the summary. To see which features are informative for generating summaries, we perform a feature selection analysis.

For this analysis, we compute the F-score [9] value for each of the 24 features using the approach of [7]. This score is commonly used to compute the discriminability of features in supervised machine learning. The score depends only on the set of features and the training data and is independent of the classifier. Features with higher F-score 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.

[9]This score is computed for the feautures and is different from the F-score computed for the summaries in Section 5.2.1.

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 conv.
SLEN	word count, globally normalized
SLEN2	word count, locally normalized
TPOS1	time from beg. of conv. to turn
TPOS2	time from turn to end of conv.
DOM	participant dominance in words
COS1	cos. of conv. splits, w/ *Sprob*
COS2	cos. of conv. splits, w/ *Tprob*
PENT	entro. of conv. up to sentence
SENT	entro. of conv. after the sentence
THISENT	entropy of current sentence
PPAU	time btwn. current and prior turn
SPAU	time btwn. current and next turn
BEGAUTH	is first participant (0/1)
CWS	rough ClueWordScore
CENT1	cos. of sentence & conv., w/ *Sprob*
CENT2	cos. of sentence & conv., w/ *Tprob*

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 [13].

Figure 7 shows the values of F-score computed for the features used by the classifiers to summarize bug reports. The results show that the length features (SLEN & SLEN1) are among the most helpful. Several lexical features are also helpful: CWS, CENT1, CENT2, SMS, & SMT. [10] These results suggest that we may be able to train more efficient classifiers by combining lexical and length features of the conversation.

5.4 Human Evaluation

The generated summaries are intended for use by software developers. Does a classifier with pyramid precision of 0.63 produce summaries that are useful for developers? To investigate whether the summaries are of sufficient quality for human use, we set up an evaluation of the generated summaries of the *BRC* classifier with a group of eight human judges. We chose to focus on the *BRC* classifier since it had performed the best based on the earlier measures.

Eight of our ten annotators agreed to evaluate a number of machine generated summaries. We asked the eight judges to evaluate a set of eight summaries generated by the *BRC* classifier. Each summary was evaluated by three different judges. The human judges were instructed to read the original bug report and the summary before starting the evaluation process. Figure 8 provides an example of a generated extractive summary the judges were asked to evaluate;

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

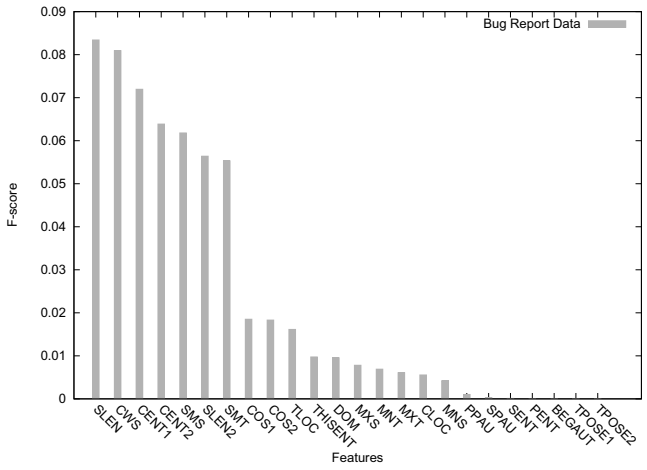

Figure 7: Feature F-scores for the bug report corpus.

a portion of the original bug appears earlier in Figure 1. We asked each judge to rank, using a five-point scale with five the high value, each bug report summary based on four statements (mean and standard deviations are provided in parentheses following the statement):

1. The important points of the bug report are represented in the summary. (3.54 ± 1.10)

2. The summary avoids redundancy. (4.00 ± 1.25)

3. The summary does not contain unnecessary information. (3.91 ± 1.10)

4. The summary is coherent. (3.29 ± 1.16)

An evaluation of meeting data summarized by multiple approaches uses similar statements to evaluate the goodness of the generated summaries [14].

We ensured in this judging process that the bug reports were assigned to judges who had not annotated the same reports during the annotation process. We also took care to choose summaries with different values of pyramid precision and F-score so as to not choose only the best examples of generated summaries for judging.

5.5 Threats

The primary threats to the evaluations we have conducted are the size of the bug report corpus and the annotation by non-experts in the projects. While the size of the corpus is sufficient for initial experimentation with these approaches, we are limited in the size of the training set that can be used to train BRC and we are unable to use a separate set of bug reports for training and testing.

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. This risk is mitigated by the experimental setup that requires a mapping from abstractive to extractive sentences. 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.

For the human evaluation, the primary threat is the use of non-experts who may not be the best judges of summaries of the reports. Another threat is not asking the human judges about the properties of interest through multiple questions, which may have allowed a better determination of the judges opinions on those properties. Finally, the judges may have wanted to please the experimenters. In future studies, we will consider interspersing classifier-generated and human-generated summaries to reduce this risk.

6. DISCUSSION

With this work, we have shown that it is possible to generate summaries for a diverse set of bug reports with reasonable accuracy as judged both against gold summaries and by human judges. The ability to produce summaries opens up the possibility of using a summary generator to support the work of software developers. We discuss these possibilities as well as consider ways to further improve the summaries produced.

6.1 Using a Bug Report Summary

Automatically generated bug report summaries could aid a software developer in several ways. During bug report triage activities, a developer (triager) must often consult other reports. For instance, the triager must typically determine if a new incoming report is a duplicate of existing reports. As this determination can be difficult for large, fast-changing repositories, recommenders have been built to suggest to the triager a set of existing bug reports to consider as duplicates (e.g., [17, 20]). Assessing these recommendations to determine which are true duplicates can be time-consuming as the determination typically involves reading much, if not all, of the report. An automatically produced summary of each existing report could ease this task by greatly reducing the amount of text the triager must read, understand and consider.

As another 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 [18]. 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.

Overall, summaries can make it possible for a developer to use the body of knowledge about the project that is captured in the repository. The ability to produce summaries automatically opens up new possibilities in making such recommenders effective and practical to use. In future work, we plan to investigate whether the provision of such summaries can provide these benefits.

6.2 Summarizing Other Project Artifacts

Bug reports are not the only artifacts with a substantial amount of text in a software development project. Requirements and design documents can contain text describing the domain, discussions of alternative approaches and rationale for why decisions have been made. Even source code, which is typically not considered to contain much text, can include many sentences about decisions or how a piece of code works. Given our success in producing summaries of bug reports,

```
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.

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

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

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)

The bug that is being shown here is the fact that you cannot yet resize your ap-
plets, and as such we also don't set default sizes sanely.

Of course :)  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 :)

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)
```

Figure 8: An extractive summary generated by *BRC* for Bug 188311 from the KDE bug repository.

we plan to investigate the generation of summaries for other text that appears in software development projects. In particular, we are interested in investigating whether textual summarization techniques can be used to help explain features implemented in source code.

6.3 Improving a Bug Report Summarizer

The bug report classifier we trained produces reasonable summaries. Although we have trained and evaluated the classifier across reports from four systems, we do not know whether the classifier will produce good summaries for reports from systems on which it was not trained. We plan to investigate the generality of the summarizer to other bug reporting systems.

Another possibility for improving the accuracy of a bug report summarizer is to augment the set of features used. 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 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'. These labels might also be used to differentiate the importance of a sentence for a summary. We leave the investigation of these intentions to future research as we do not yet have the ability to automatically label the sentences for the test data.

It is also still an open question whether extractive summarization is the most appropriate choice for bug reports or whether better results could be obtained through an abstractive summarizer. We leave this determination to future work.

7. 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 and determining why particular design decisions have been made as recorded in design documents and elsewhere. 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. The human judges we asked to evaluate reports produced by the bug report summary generator agree that the generated extractive summaries contain important points from the original report and are coherent. This work opens up new possibilities to improve the effectiveness of existing systems for recommending duplicate bug reports and for recommending similar changes completed in the past for a current software evolution task. It also opens up the possibility of summarizing other software project artifacts to enable developers to make better use of the rich knowledge base tracked for most software developments.

Acknowledgments

Thanks to Giuseppe Carenini and Raymond Ng for useful discussions on summarization and to Jan Ulrich for helping with the annotation software. This research is supported by NSERC. Thanks also to the annotators for helping to create the bug report corpus.

8. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[3] J. K. Anvik. *Assisting bug report triage through recommendation*. PhD thesis, University of British Columbia, 2007.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[5] G. Carenini, R. T. Ng, and X. Zhou. Summarizing emails with conversational cohesion and subjectivity. In *ACL-08: HLT: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 353–361, New York, NY, USA, 2008. ACM.

[6] J. Carletta, S. Ashby, S. Bourban, M. Flynn, T. Hain, J. Kadlec, V. Karaiskos, W. Kraaij, M. Kronenthal, G. Lathoud, M. Lincoln, A. Lisowska, and M. W. P. D. Reidsma. The ami meeting corpus: A pre-announcement. In *MLMI '05: Proceedings of Machine Learning for Multimodal Interaction: Second International Workshop*, pages 28–39, 2005.

[7] Y.-W. Chen and C.-J. Lin. Combining svms with various feature selection strategies. In *Feature extraction, foundations and applications*, pages 315–324. Springer, 2006.

[8] T. Fawcett. Roc graphs: Notes and practical considerations for researchers. Technical report, HP Laboratories, 2004.

[9] J. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[10] U. Hahn and I. Mani. The challenges of automatic summarization. *Computer*, 33(11):29–36, 2000.

[11] B. Klimt and Y. Yang. Introducing the enron corpus. In *CEAS '04: Proceedings of the First Conference on Email and Anti-Spam*, 2004.

[12] A. J. Ko, B. A. Myers, and D. H. Chau. A linguistic analysis of how people describe software problems. *VL-HCC '06: Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, 0:127–134, 2006.

[13] G. Murray and G. Carenini. Summarizing spoken and written conversations. In *EMNLP '08: Proceedings of the 2008 Conference on Empirical Methods on Natural Language Processing*, 2008.

[14] G. Murray, S. Renals, J. Carletta, and J. Moore. Evaluating automatic summaries of meeting recordings. In *MTSE '05: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics, Workshop on Machine Translation and Summarization Evaluation*, pages 39–52. Rodopi, 2005.

[15] A. Nenkova and R. Passonneau. Evaluating content selection in summarization: the pyramid method. In *HLT-NAACL '04: Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2004.

[16] O. Rambow, L. Shrestha, J. Chen, and C. Lauridsen. Summarizing email threads. In *HLT-NAACL '04: Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2004.

[17] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[18] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[19] S. Wan and K. McKeown. Generating overview summaries of ongoing email thread discussions. In *COLING '04: Proceedings of the 20th International Conference on Computational Linguistics*, pages 549–556, Morristown, NJ, USA, 2004. Association for Computational Linguistics.

[20] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 461–470, New York, NY, USA, 2008. ACM.

[21] K. Zechner. Automatic summarization of open-domain multiparty dialogues in diverse genres. *Computational Linguistics*, 28(4):447–485, 2002.

[22] X. Zhu and G. Penn. Summarization of spontaneous conversations. In *Interspeech '06-ICSLP: Proceedings of the 9th International Conference on Spoken Language Processing*, pages 1531–1534, 2006.