Mining Bug Repositories – A Quality Assessment

Philipp Schugerl¹, Juergen Rilling¹, Philippe Charland²

¹Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{p_schuge, rilling}@cse.concordia.ca

²System of Systems Section
Defence R&D Canada - Valcartier
Québec, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract

The process of evaluating, classifying, and assigning bugs to programmers is a difficult and time consuming task which greatly depends on the quality of the bug report itself. It has been shown that the quality of reports originating from bug trackers or ticketing systems can vary significantly. In this research, we apply Information Retrieval (IR) and Natural Language Processing (NLP) techniques for mining bug repositories. We focus particularly on measuring the quality of the free form descriptions submitted as part of bug reports used by open source bug trackers. Properties of natural language influencing the report quality are automatically identified and applied as part of a classification task. The results from the automated quality assessment are used to populate and enrich our existing software engineering ontology to support a further analysis of the quality and maturity of bug trackers.

1. Introduction

Bug trackers, version control systems, wikis, and message boards are standard tools to help manage the development and evolution of software projects. Such repositories contain explicit and implicit knowledge about software projects that can be mined to provide additional insights to guide the continuous software development and to plan evolutionary aspects of software projects. In what follows, we focus on the mining and analysis of bug reports found in bug repositories [1]. These bug reporting systems allow users to submit, describe, track, comment on, and classify bug reports and feature requests. One popular example of such a bug reporting tool used for open source projects is Bugzilla.¹

Previous studies have shown that many bug reports contain invalid or duplicate information [1]. For the remaining bug reports, a significant portion tends to be of low quality due to the omission of important information or the inclusion of irrelevant information [8, 9]. Our research is motivated by the fact that organizations typically deal with a large number of bug reports that are manually evaluated for their quality and content. We present an automated tool to support the evaluation and classification of bug reports based on their quality will provide organizations with an immediate added benefit. Automatic identification of low-quality bug reports will not only free-up resources otherwise used for manually classifying and evaluating these bug reports. It will also improve the prioritization and timely handling of these bug reports with higher quality descriptions. Evaluating the quality of bug reports represents an important step towards enhancing the overall quality and maturity of bug trackers. The research presented in this article is part of our ongoing research on applying semantic web technologies to support system evolution [23].

The remainder of this paper is organized as follows: In Section 2, we provide an overview of text mining and its support for the extraction of information from unstructured text. Section 3 introduces factors applicable to classify and to evaluate the quality of bug reports. Section 4 reports results from our case study in analyzing the quality of bug reports for an open source project. Section 5 compares our work to other relevant work in the domain, and Section 6 concludes and gives future directions.

2. Text Mining

Text mining is commonly known as a knowledge discovery process that aims to extract non-trivial information or knowledge from unstructured text [3]. Unlike Information Retrieval (IR) approaches [4], text mining does not simply return documents pertaining to a query. It rather attempts to obtain semantic information from the documents themselves using techniques from Natural Language Processing (NLP)
and Artificial Intelligence (AI). Text mining systems are often implemented using component-based frameworks, such as GATE [6] or IBM’s UIMA\(^2\). During the text mining process, a number of standard NLP techniques are commonly used. These techniques include the use of (Unicode) tokenizer to first divide the textual input stream into individual tokens. Then a sentence splitter detects sentence boundaries, and after a statistical Part-of-Speech (POS) tagger is used to assign labels (e.g., noun, verb, and adjective) to each word. Larger grammatical structures, such as Noun Phrases (NPs) and Verb Groups (VGs), can then be created based on these tags using chunker modules. Based on these foundational analysis steps, more semantically-oriented analyses can be performed, which typically require domain and language specific algorithms and resources.

3. Report Quality

Bug reports consist of multiple fields that have to be completed prior to submitting a report. In the context of our research, we are focusing on analyzing the free text used to describe the encountered problem and circumstances under which a bug occurs. Although reporting guidelines typically exist within a project, current bug tracking systems are incapable of enforcing them as they do not analyze this text. Some recent studies [8, 9] have shown that bug quality assessment is largely determined by the free text entered by reporters which greatly varies in quality.

3.1. Analysis

Bug reports provide a number of distinctive characteristics which allow developers to judge their quality. First, the quality of a bug report largely depends on its helpfulness in identifying and understanding the reported problem. A survey performed by Bettenburg et al. in [8] shows that among developers the most important properties maintainers/programmers look for in a bug report are the steps to reproduce the problem (83%), stack traces (57%), test cases (51%), screenshots (26%), code examples (14%), and a comparison of observed versus expected behavior. Second, bug report guidelines, e.g. [7], have been formulated to describe characteristics of a high quality bug report. Guidelines for writing good bug reports include: (1) Be precise; (2) Explain it so that others can reproduce it; (3) One bug per report; (4) Clearly separate fact from speculation; (5) No abuse or whining about decisions.

3.2. Attributes

In what follows we introduce a new set of quality guidelines used for the evaluation of free form bug description typically associated with bug reports. The attributes themselves are derived from results observed in [8, 9] and from general guidelines for good report qualities, such as the ones discussed in [7]. Each attribute definition is illustrated through bug excerpts extracted from the ArgoUML\(^1\) bug repository. Keywords and key expression are highlighted in bold.

3.2.1. Reproducibility. The bug report description includes steps to reproduce a bug or the context under which a problem occurred.

```
Cannot delete a diagram.
After adding a diagram (class/state), I couldn't delete it from the project.
```
(Bug# 269)

```
Checking if names are unique
First, create two packages and one class diagram by package. Then, add one class to a package....
```
(Bug# 79)

3.2.2. Observability. The bug report contains a clearly observed behavior, whether positive or negative. Evidence of the problem such as screenshots, stack traces, or code samples is provided.

```
GUI hangs when attempting to bold text
The GUI hangs (CPU load for the java process jumps to 90% + and does not stop) when I try to change the style of a text object.
```
(Bug# 364)

```
Question mark does not work in text fields
In text fields, the question mark does not work. I have a German keyboard layout and version 0.9.3
```
(Bug# 374)

3.2.3. Certainty. The level of speculation is embedded in a bug description. A high certainty indicates a clear understanding of the problem and often also implies that the reporter can provide suggestions on how to solve the problem.

```
Individual parts won't link after downloading
I'm new to Java, hence this is probably a very simple error and not a 'true' bug. When I type ...
```
(Bug# 333)

```
Import class from another package?
To Me it seems not to be possible to create a class within a diagram from a different package?
```
(Bug# 378)

---

\(^1\) http://argouml.tigris.org/

\(^2\) www.research.ibm.com/UIMA/
3.2.4. Focus. The bug description does not contain any off-topic discussions, complaints or personal statements. Only one bug is described per report.

Hi,
I'm a very new user to ArgoUML. I found it exciting and hope to be an enthusiastic contributor. Direct to the point...

(Bug# 236)

V0.10 on OS X has no menu bar
When launching v0.10 on OSX, no menu bar is visible. Additionally, none of the hot keys work (like Ctrl-S for save).

(Bug# 860)

In addition to these categories, general text quality measurements, such as evaluating the grammatical correctness of the bug description, the number of spelling errors per sentence, and readability indices can be applied.

4. Evaluation

As previously mentioned, the quality of bug reports plays an important role in projects in which bug trackers are publicly accessible with a large number of reported bugs, as it is the case for many open source products. For the evaluation of our approach, we selected ArgoUML, a leading UML editor with a publicly accessible bug tracking system. Since its interception in 1998, ArgoUML has undergone several release cycles and is still in active development. Its bug database counts over 5,100 open/closed defects and enhancements. In what follows, we describe the data set extracted from the ArgoUML bug repository and the NLP techniques used to mine the bug descriptions. At the end of the section, we provide a discussion on the observed results from our automated analysis of the bug description quality in ArgoUML.

4.1. Data

From the ArgoUML bug repository, we extracted a dataset consisting of 4,839 bug reports, which included 3,731 reported defects and 1,108 feature and enhancements requests. From the reported bugs, we only considered the closed cases. Closed cases include all bugs that have already undergone the triage problem [22] and, therefore, have been classified or assigned to a maintainer. An analysis of the reported defects showed that 80% of all bugs in the repository can be considered as closed. Among these closed bugs a relatively large percentage had been either marked as invalid (21%) or duplicate (12%), resulting in only 67% of all closed bugs to be fixed bug instances.

4.2. Methodology

For the assessment and identification of our quality attributes introduced in section 3.2, we used natural language processing in conjunction with simple field extraction, as supported by the GATE framework. In what follows, we explain in more detail the extraction approach applied for each of the quality attributes.

Prior to be able to determine the Reproducibility of a report, the context in which an error occurred needs to be identified. By manually evaluating over 500 bug reports, time clauses used in bug descriptions were found to be a reliable indicator for paragraphs describing the context in which a problem occurred. For example, “When I clicked the button” or “While starting the application”. These can be relatively easy annotated using a POS tagger and JAPE grammar. To identify the listing of reproduction steps, the standard GATE sentence splitter has been modified to recognize itemizations (characters ‘+’, ‘o’, ‘*’) as well as enumerations (in the form of ‘1.’, ‘(1)’, ‘[1]’).

For the identification of the Observability in bug descriptions, we compared word frequencies with the expected numbers from non-bug related sources. For words appearing distinctively more frequently in bug descriptions than expected, a categorization in positive and negative sentiment has been performed. Table 1 shows a sample of identified words and their sentiments.

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg. Noun</td>
<td>attempt, crash, defect, failure, ...</td>
<td>22</td>
</tr>
<tr>
<td>Neg. Verb</td>
<td>disappear, fail, hang, ignore, ...</td>
<td>32</td>
</tr>
<tr>
<td>Neg. Adj.</td>
<td>broken, faulty, illegal, invalid, ...</td>
<td>34</td>
</tr>
<tr>
<td>Pos. Verb</td>
<td>allow, appear, display, found, ...</td>
<td>24</td>
</tr>
<tr>
<td>Pos. Adj.</td>
<td>correct, easy, good, helpful, ...</td>
<td>16</td>
</tr>
</tbody>
</table>

We applied a gazetteer to annotate both positive and negative observations. A GATE plug-in was used to identify negated sentence fragments and transform positive observations into negative observations (e.g., “not fully working” is a reliable hint for an error observation, while the word “working” alone is not).

Stack traces supplied as part of bug reports can be identified relatively easily using regular expressions. Similarly, source code fragments are identified by searching for typical expressions such as variable declarations or method calls and class names. We also categorize attachments by their file type. This information is already available in most bug-trackers.

Kilicoglu et. al. [10] have recently shown that hedges, an important indicator for the Certainty expressed in a report, can be found with high accuracy.
using syntactic patterns and a simple weighting scheme. In our approach, we reused the gazetteer lists presented in [10] to identify speculative language. Due to the availability of a negation-identifier, it was further possible to add additional hedging cues based on negated verbs and adjectives (e.g., “not sure”). As suggestions to solve a problem also make use of hedging, a distinction between problem description and suggested solution has to be made. As the problem descriptions tend to appear at the start of a bug report, while suggestions tend to appear at the end, only hedges found in the first half of an error report have been counted. Additionally, the default GATE sentence splitter has been modified to correctly tag question-sentences.

We assessed the Focus of bug reports by identifying emotional statement (such as “love” or “exciting”) as well as topic splitting breaks (such as “by the way” or “on top of that”) through a gazetteer.

4.3. Results

For the evaluation of our approach, we selected a random data sample consisting of 178 bug reports in the ArgoUML bug repository. Nine experienced Java developers (master and Ph.D. students who have previously worked with ArgoUML at the source code level) have been asked to complete a questionnaire assessing the quality of these bug reports. For each of the selected bugs, the users performed a subjective evaluation of the bug report quality using a scale ranging from 1 to 5 (with 1 corresponding to very high quality and 5 to very low quality).

From these 178 bugs, the developers rated 78% of the reviewed bug reports as being either of high or very high quality; 22% of the bug reports were classified as average to very low quality. The distribution of the collected answers indicates that the average quality of existing bugs of ArgoUML is overall relatively high. This observation might be directly related to the strong technical background of the ArgoUML community. ArgoUML itself is a design tool, which is typically used by software designer and developers with a solid background in programming and software development.

The collected data was then normalized and analyzed for high deviation (low confidence in the assigned quality value). As a result of this data normalization step, 15 bugs with high deviation were filtered out. The remaining 163 bugs provided the input for the supervised training of our learning models. We used a Naïve Bayes classifier with leave-one-out cross validation (predicting the quality of each bug by learning from all other bugs). We have converted language characteristics (annotated by GATE as described in section 4.2) into features by using a custom XML import plug-in for RapidMiner as shown in table 2.

Table 2. Extraction per bug (*=10-bin-discretization)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Rep. Context Mention</th>
<th>Enumeration</th>
<th>Source-code Fragment</th>
<th>Attachment Type</th>
<th>Observed Behavior</th>
<th>Hedge</th>
<th>Personal Statement</th>
<th>Question</th>
<th>Spelling error</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Text</td>
<td>Count</td>
<td>Count</td>
<td>Text</td>
<td>Text</td>
<td>Text</td>
<td>Text</td>
<td>Text</td>
<td>Count*</td>
<td>Count*</td>
</tr>
</tbody>
</table>

Table 3 shows the precision and recall of our learned quality model. The columns denote the average quality rating observed by developers; rows show the quality predicted by our approach. Dark grey cells show a direct overlapping between the predicted quality and the one rated by developers. In addition, the light grey areas include the predictions which have been off-by-one from developer ratings.

As expected, the classification of bugs with good quality tends to be easier than identifying poor quality bug descriptions. A further analysis of features weights in the classification showed that “personal statements”, “observed behavior”, and a low number of words in general have the largest influence on detecting poor quality descriptions. “Enumerations” and “reproducible context mentions” do provide a relative stable hint in
detecting good quality bug reports. Spelling errors were found in both high and low quality bug reports and, therefore, should not be considered as a reliable indicator for further assessment. This might be related to the open source community and its common, more relaxed form of reporting.

4.4. Validation

We have applied our learned quality model to the ArgoUML bug tracker to evaluate the overall quality of bugs stored in the bug repository. Approximately 60% of all bugs have been classified by our approach as ‘very good’. This result corresponds closely with the results obtained from our manually evaluation of the ArgoUML bug repository in which also a large percentage of reported bugs has been rated ‘good’ or ‘very good’. Additionally, the percentage of bugs with low quality (9% poor or very poor and 16% average) correlates closely with the number of invalid bugs.

Our case study showed that our model can provide a good initial assessment of the quality level of ArgoUML’s bug tracker repository. Furthermore, the classification results can be further analyzed to investigate and assess the overall maturity level of bug trackers.

By applying our model, we can now provide an initial measurement for the quality level of ArgoUML’s bug tracker repository.

\[
\text{Level} = \frac{Bad\text{QualTimeToFix}}{Avg\text{TimeToFix}} \times Bad\text{QualOpen}
\]

The ability to reason about the quality of bug reports can also provide additional insights with respect to the performance of bug reporters and their report writing skills.

Figure 1 shows the distinctive groups of bug reporters submitting reports in the ArgoUML bug trackers. The groups are based on their ability to write bug reports. There exists a large group (46%) of report writers who always produce high quality reports (reports with a quality ranking of either good or very good). However there is a surprisingly large group (37%) of bug report writers, who do not adhere to good reporting guidelines, resulting in reports that are only of poor or very poor quality ratings.

Having identified a group of bug reporters that always submit low quality bug reports, this group of reporters can now be specifically targeted in attempt to improve the quality level of their bug reports (e.g., stricter or more refined guidelines, additional training and samples of good reports).

5. Related Work

Very little work exists on text mining software documents containing natural language. Most of this research has focused on analyzing texts at the specification level, e.g., in order to automatically convert use case descriptions into a formal representation [12, 13], or detect inconsistent requirements [14]. In comparison, our work focuses on the analysis of quality attributes of free form bug descriptions. Analyzing the natural language in bug reports is an inherently difficult problem since these descriptions cover various levels of abstractions, ranging from feature requests to compilation errors.

There is a significant body of work that has studied bug reports to automatically assign developers to bug reports [15], to assign locations to bug reports [16], to track features over time [17], to recognize bug duplicates [18, 19], and to predict effort for bug reports [20]. Antoniol et. al. [21] pointed out that there often exists a lack of integration between version archives and bug databases. The Mylyn tool by Kersten and Murphy [22] can attach a task context to bug reports for traceability purposes.

However, only limited work exists on modeling and automatically evaluating the quality of bug reports. The work most closely related to ours is from Bettenburg et. al. and their QUIZILLA tool [8]. They also evaluate quality of bug reports using different quality attributes. Our work can be seen as a continuation of their work. Our reproducibility attribute is a refinement of Bettenburg’s [9] attribute, as our work also considers the context in the bug description. We also extend their observability property with negative observations to be further analyzed. Furthermore, we introduce two new properties: certainty and focus. Certainty evaluates the confidence level of the bug writer in analyzing and describing the bug. Our focus property considers words and phrases indicative of emotions and other prose text that might bloat the bug description and make it less comprehensible. Ko et. al. [11] performed a linguistic analysis of bug reports. Their work, however, lacks both a concrete application and an evaluation of their
approach since its interception in 1998 was limited to bug titles, while our work analyzes the full bug description.

6. Discussion

As we have shown throughout the article, writing bug reports of high quality is not a simple task. Different factors can affect the overall quality of bug reports. The free form descriptions attached to bug reports often contain important information describing the context of a bug, the type of unexpected behavior that occurs, and even potential solutions to resolve the problem. However, manually assessing the quality of these natural language based descriptions is a time consuming task. Being able to provide an automated quality assessment is a first step towards improving the quality and maturity of bug reports.

10. References