Identifying Shared Software Defects between Open-Source Software Projects

Ivo van Hurne
i.vanhurne@student.utwente.nl

ABSTRACT
Open-source software projects commonly use source code from other projects as building blocks for their own software. Despite sharing source code, information about software defects affecting the code is only sparsely shared. One of the causes for this lack of communication is the difficulty of identifying defects that may also affect other software projects. In this paper three approaches to identifying these shared software defects are compared. Support Vector Machines is found to be the best approach to solve this problem.

Keywords
Software Defects, Bugs, Issue Tracking, Open Source

1. INTRODUCTION
1.1 Motivation
In recent years the number of software projects that are developed using open-source methods has risen spectacularly. Some of them have been highly successful and are even competing with expensive commercially developed software [1].

A lot of open-source projects reduce duplicated efforts by building on top of other open-source projects. The Debian Linux Project[1] for example, develops an open-source operating system but does not write its own web browser. Instead a slightly adapted version of Mozilla Firefox[2] is used.

A disadvantage of this approach is the possibility of introducing software defects (also known as bugs) to a project, that are present in the source code of underlying projects. A software defect is an incorrect piece of source code. Observable incorrect behaviour of the software, caused by such a defect, is called a failure [2].

Additionally, using source code from other projects increases complexity. This means failures can also be caused by unpredictable

1http://www.debian.org/
2http://www.mozilla.com/firefox/

interactions between parts of the project’s own source code and underlying projects.

As can be observed by reading the websites of many open-source projects (and also noted by Ven et al. [3]), independent software vendors play an important role in the distribution of open-source software. They all use different operating systems and different library versions. This makes the situation even more complex and makes it impossible for the software developers to test the software under all important conditions.

Additionally all software projects and software vendors have their own issue tracking systems to keep track of reported software failures. These systems are not linked and thus there is hardly any communication between them. This causes a lot of duplication of effort when failures are caused by lower-level projects. Conversely, knowledgeable users may report a failure to a lower-level project while leaving the upper levels in the dark about it.

As we mentioned earlier, failures are caused by defects. Defects are not observable to users, so they report the resulting failures instead. These failures should then lead the developers to the defects that caused them. Ideally the information about the defects should be passed on to other projects that are using the same source code, to prevent failures in their software.

Recent events, like the Debian/OpenSSL RNG incident [4, 5], show that insufficient communication about (possible) software defects between projects can lead to dangerous failures of important software functions. In this particular case the Debian developers had fixed a supposed defect in the OpenSSL library. Two years later it was discovered that the defect was in fact intended behaviour and that the fix severely reduced the randomness of OpenSSL’s random number generator. Because OpenSSL is one of the most important encryption libraries in the Debian operating system a lot of encrypted data was compromised. Had the issue tracking systems of these projects been linked, the OpenSSL developers could have warned their Debian colleagues about the mistake.

A large number of problems are still to be solved when integrating information about software failures and defects between issue tracking systems. One of these problems is finding software defects that (possibly) also exist in another project, or may even have originated in another project. We will hereafter refer to these kind of software defects as shared software defects. In this paper we apply three well-known machine learning algorithms to finding these shared defects, and compare their performance.
1.2 Research Question

The main research question is:

Which of the proposed solutions performs best at identifying shared software defects in an issue tracking system?

The research question is decomposed into the following subquestions:

- What properties of software failures are commonly stored in issue tracking systems?
- Which of these stored properties could be useful when identifying shared defects?
- Which solutions for identifying relations between software failures have been proposed?
- How successful are the proposed solutions at identifying shared defects?

1.3 Related Work

A number of works that introduce the concepts of software failures and issue-tracking systems have been published. Zeller [2] defined defects and failures, and their relation. Anvik et al. [6] explained the most common failure properties. Canfora et al. [7] described the usual workflow in fixing defects and working with issue-tracking systems. Song et al. [8] classified a number of software defect types that are most often encountered. Tatham [9] explained what information is most useful for developers when fixing software defects.

Some research into identifying relations between software defects has also been done. Čubranić et al. [10] proposed Bayesian classification to identify the developer best qualified to fix a defect. Song et al. [8] proposed association rule mining for defect correction effort prediction. Weiß et al. [11] proposed a combination of text vectors and the nearest neighbour approach for predicting the defect correction effort. Anvik et al. [12] proposed support vector machines to automatically decide which developer should resolve a defect.

We are not aware of any earlier work on the particular problem described in this paper.

1.4 Research Approach

To answer these questions, we reviewed existing literature and performed an experiment. We first defined the properties of software failures that are most often stored in issue tracking systems. These properties were then used to answer the second subquestion.

Because there currently is no easy way to compare software failures between different issue tracking systems, we tried to compare failures in a single issue tracking system using various machine-learning techniques. This may seem like a strange approach, because a shared software defect involves two projects that share some source code and a defect related to that source code. It is however our hypothesis that by comparing a failure to previously reported failures, we can identify failures caused by shared software defects.

The machine-learning techniques that were used are not specifically geared to finding shared defects. They have all kinds of applications, for example spam filtering or predicting defect correction effort. Because of their widespread use in solving other related problems, we assumed they would also be applicable in this context.

We applied the proposed solutions to a sample database and compared the resulting assessments to human assessment. The Debian Bug Tracking System (DBTS) [13] database was used as sample database. This database contains information about approximately 499,000 reported software failures in Debian Linux. Additionally, Debian developers flag failures in this database if they are forwarded to other open-source software projects. This provides a way to verify the results against human assessment.

Because the database is very large (more than 20 GiB) and there was a limited amount of disk space available, we were only able to use the so called ‘active’ part of the database. This part contains the unresolved failures and the failures that were resolved in the last 28 days. We took a snapshot of this database part on October 9, 2008; it contains 70,385 reported software failures.

The DBTS database is only available in plain text format, and therefore difficult and time-consuming to query. To query the database more efficiently, we converted a database snapshot to a PostgreSQL database using software based on the work of Francisco et al. [14]

When running the experiments on this snapshot it became apparent that, even using the faster database, 70,000 reports still take a very long time to analyze. About ten reports per minute could be analyzed using Bayesian classification (which is the fastest of the algorithms used in this paper [15]). One analysis of the whole database would consequently take almost five full days. We therefore decided to use just 1,000 reports that were randomly selected from the 70,000 reports in our database snapshot.

2. DEBIAN BUG TRACKING SYSTEM

2.1 Issue Tracking Systems

DBTS is an issue tracking system. This kind of system is an information system used to manage information about problems or tasks (issues) in software projects. It aids in communication about these issues and provides information about the history of the project and the way issues have been dealt with in the past [11]. It is part of the philosophy behind open-source projects to have an ‘open’ issue tracking system, that allows anyone (not just the developers of the project) to view and report issues [16].

A lot of different information about the issues is stored. There are free-form fields (e.g. issue summary) as well as pre-defined fields (e.g. severity), fields that define relations between issues (e.g. duplicate of) and attachments [11, 12].

Every issue also has a state, which indicates whether or not the issue has been resolved and if someone is currently working on resolving the issue. Examples of states are: UNCONFIRMED, ASSIGNED, RESOLVED and REOPENED [11, 12].

2.2 Issues in the DBTS

In the DBTS, issues are called bugs. Users do not report issues about source code files or libraries, but about packages. A package

3http://www.postgresql.org/
usually contains an application or a library, plus an installer and metadata.

One could argue that not all issues in the database are software failures, because some of them are classified as wishlist issues. The DBTS documentation says this classification is used for “for any feature request, and also for any bugs that are very difficult to fix due to major design considerations” [17]. Because it is likely that feature requests that are reported to Debian will also contain information useful to the project originally developing the application, we will consider feature requests to be software failures in this paper.

2.3 Fields to Analyze
There is no previous work on this particular problem that the author is aware of. Therefore it is difficult to argue which of the fields recorded in DBTS should be considered when identifying shared defects. We assume subject, summary, package and replies to be the most important fields. Subject, summary and replies are free-form fields that uniquely describe the failure. The package field indicates which (according to the reporter of the failure) component or application caused the failure.

We think other fields like severity, date reported and reporter are far less likely to identify a shared defect. Shared defects do not cause more severe or less severe failures than non-shared defects per se. Users will report any kind of failure, not just shared or non-shared defects (they are most likely not even aware of the difference), and regardless of the time or date.

3. POSSIBLE SOLUTIONS
Earlier work on relations between software failures provides a number of approaches that can possibly also be put to use identifying shared software defects between projects.

3.1 Naïve Bayes Classification
Bayesian classification was first introduced by Kalt [18] for the classification of text documents. Nigam et al. later introduced an improved version [19].

An issue report is split into features. In our case, these are the words in the free-form fields and the name of the package the issue was reported for. For each feature, the probability that an issue containing this feature belongs to a certain class is calculated. These probabilities are then combined to a single probability for each possible classification and the classification with the highest probability is chosen.

The technique is called ‘naïve’ because it assumes all features are independent of each other. This is of course a false assumption, but it has worked remarkably well in a lot of applications [15].

3.2 Nearest Neighbour Approach
The nearest neighbour approach (kNN) [20] is frequently used to estimate effort and cost of software projects in early phases [11].

A target issue is again split into features. It is then compared to other issues using a distance function. Such a function combines the distances of all the features between two issues into one distance. For example, an issue contains the word ‘crash’ one time and another issue does not contain the word crash. In this case the distance between the issues is one.

3.3 Support Vector Machines
The Support Vector Machines (SVM) algorithm [21] uses a function to classify issues. To illustrate how this works, let us assume for a moment that all issues have just two features. We draw a graph of all issues using the two features as x and y axes (Figure 1). If the issue is caused by a shared defect, it is represented as a circle. The other issues are represented as crosses. SVM then draws a line for us that splits the graph into one area with the most circles and one area with the most crosses. We can now easily predict the classification for an issue by checking in which area it is located.

The classifications of the k issues with the smallest distance to the target issue are then averaged to classify the issue.

3.4 Implementation
The solutions were implemented using the Python4 language, which is increasingly considered as an alternative to traditional scientific computing tools like Matlab [15, 22]. We adapted the example source code written by Segaran [15] for the previously mentioned approaches. The Psyco5 just-in-time compiler was used to speed up the execution. No formal verification of the implementation has been done.

4. RESULTS

4.1 Performance Criteria
We assess the performance of the approaches using the precision (Formula 1) and recall (Formula 2), combined into the F-measure (Formula 3). This is an often used measure of performance in the field of document classification [23].

\[
\text{Precision} = \frac{\# \text{ of true positives}}{\# \text{ of true positives} + \# \text{ of false positives}} \quad (1)
\]

\[
\text{Recall} = \frac{\# \text{ of true positives}}{\# \text{ of true positives} + \# \text{ of false negatives}} \quad (2)
\]

\[
\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)
\]

\[\text{Figure 1: Illustration of linear classification} \quad [15]\]

Note that SVM can use different kinds of classification functions, not just linear ones.

4.2 Contributions

- A dataset consisting of Debian bug reports was created.
- The naive Bayesian and kNN classification methods were implemented.
- The SVM approach was applied.
- The approaches were compared using performance criteria.

5. CONCLUSION

The goal of this paper was to investigate possible solutions for identifying shared software defects between projects. We believe our work is a valuable contribution to the field of software engineering and opens up new avenues for research.

Note:

4Python v2.5.2, http://www.python.org
5http://psyco.sf.net/
### Table 1: Shared defect identification results

<table>
<thead>
<tr>
<th>Approach</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>12</td>
<td>75</td>
<td>20</td>
</tr>
<tr>
<td>kNN, (k = 3)</td>
<td>50</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>SVM Linear</td>
<td>75</td>
<td>33</td>
<td>46</td>
</tr>
</tbody>
</table>

\[
\text{Recall} = \frac{\text{# of true positives}}{\text{# of true positives} + \text{# of false negatives}}
\]

(2)

\[
\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

(3)

In this case a 'positive' outcome means that the approach that is being tested has classified the issue as a shared defect. 'Negative' means it was classified as non-shared by the approach. ‘True’ or ‘false’ indicate whether or not the human assessment is in agreement with the outcome.

We randomly selected 1,000 issues (as mentioned earlier) and divided them into a training set (900 issues) and a test set (100 issues). This was done two times and for each approach the best performance was chosen. Training on 90% of the database is fairly common in these kinds of experiments [24]. The classifications were verified against the human assessment of the Debian developers (as recorded in the DBTS). The results are shown in Table 1.

We also tried using SVM with a radial classification function and kNN with \(k\) ranging from 4 to 10. They are however not shown here, as the results were not statistically significantly different (=<1%) from linear SVM and kNN \(k = 3\).

### 5. CONCLUSIONS

#### 5.1 Evaluation

The results clearly show that SVM is the best-performing approach, as it has the highest F-measure. It is hard to say whether the classifications made by SVM are good enough for fully automatic shared defect identification. We think that implementing it as an advisory function in the issue-tracking system first, may be a good way to further investigate this.

#### 5.2 Threats to validity

There are a number of circumstances that could affect the significance of the results. It is unknown to what extent the the ‘active’ part of the DBTS database that we used (and the issues randomly selected from it) are representative for the database as a whole.

Additionally it is possible that certain issues do not yet have the correct human assessment, because the developers have not yet had time to identify them as shared defects. There is no ‘unknown’ classification in DBTS, issues are classified as not being shared defects by default.

### 6. FURTHER WORK

A lot of work is still to be done regarding this problem. The research presented in this paper should be seen as a starting point for further research, rather than a definitive solution to the problem. We will give some pointers for such research.

First, the extraction of features from issues could be improved. We used just the individual words, but better results may be achieved by using word pairs. Furthermore stemming and assigning different weights to different issue fields should be considered.

Second, because of the limited time available for this research we were not able to fully optimize the parameters of the SVM algorithm for the dataset. Results might thus be improved by looking into this.

Third, it would be interesting to see how the proposed solutions perform for issue-tracking systems of different (open-source) projects.

### ACKNOWLEDGEMENTS

I would like to thank Maarten Fokkinga and my peers for their invaluable support and feedback.

### REFERENCES


