1. Introduction

A software repository is a storage location for source code, its documentation, license, metrics etc. In the era of agile software development changes to source code are inevitable. To help manage those changes software developers use version control systems e.g., Git [1]. Such system records changes to a file (or set of files) over time so that one can recall specific versions later. Recently, web-based repository hosting services equipped with a version control system, e.g., GitHub supported by Git, are getting more and more popular. In 2019 there were over 40 million users of GitHub [2] and they have created over 100 million repositories [6].

Unfortunately, it can happen that some changes are incorrect (buggy). To mitigate this risk, before a given change is incorporated into the software its correctness is checked via review done by one (or more) of the software developers.

The dissertation concerns an interesting area of research, called mining software repositories, that explores the information stored in repositories hosted by GitHub and alike to help software developers cope with quality management problems. More precisely, the Author addresses two important problems:

- How to recommend code reviewers? (Chapter 4)
- How to identify buggy files based on the information contained in the bug report? (Chapter 5)

Moreover, some statistics about GitHub’s software repositories and a simple classifier of commits (into “buggy” / “non- buggy”) are presented in Chapter 3.

The scope of dissertation outlined above looks very interesting and has also some practical potential.

2. Recommendation of code reviewers

One of the features of contemporary version management systems is pull request,. Roughly speaking, pull request is a signal send by a programmer who implemented a given change (e.g., a bug fix). That signal informs the others about readiness of the change to be reviewed. Then a reviewer (or reviewers) should be selected who will perform quality checking. The concept of pull request is especially important in the context of big distributed teams.
The problem discussed in Chapter 4 could be formulated in the following way:

**RCR Problem (Recommendation of Code Reviewers).** How to automate the process of recommending the most suitable reviewers for a given pull request assuming that:

1. Humans are very good at selecting suitable reviewers but slow. Therefore the automatic system should mimic their behaviour and learn from their choices made in the past.

2. Quality of reviews made in the past by a prospective reviewer as well as review time can be neglected when assigning her or him to the task.

3. The main factor influencing decisions about reviewer selection is similarity between the pull request at hand and the reviews done by a prospective reviewer in the past. Other aspects are negligible (e.g. availability of a person for the task).

The pure version of the above problem (i.e. without the mentioned assumptions) is very practical. According to P. Thongtanunam et al. [4], up to 30% of pull requests suffer from delay caused by difficulty of finding a suitable reviewer and that delay can take even 12 days. The hope is that those delays could be avoided by automating the selection of reviewers.

The assumptions mentioned above seem reasonable: they allowed the author to base the evaluation procedure on the data available on GitHub. Without those assumptions perhaps the only way to evaluate a candidate solutions to the RCR Problem would be a controlled experiment – that would be time consuming and expensive. The main weakness of those assumptions is that the first two of them have not been directly stated in the dissertation.

The most valuable contributions presented in Chapter 4 are the following ones:

- **The idea of reviewer profile** that would ‘accumulate’ the information about past reviews done by a given person. Every code review requires an update to the reviewer’s profile, but searching for a suitable reviewer is significantly faster (even 50 times faster for large open-source projects – see p. 30) as there is no need to compare a pull request at hand with all the reviews done by a given person in the past.

- **The idea replacing the four string comparison functions used by P. Thongtanunam et al.** with bag of words constructed over components of file paths concerning the files subject to review. Bag of words are also used by the author to represent reviewer profiles (a very nice example is given in p. 31). To compute similarity between a bag of words representing a pull request and a bag of words representing profile of a prospective reviewer the author considered two options: the Jaccard coefficient and the Tversky index – the latter (with \( \alpha = 0, \beta = 1 \), and no extinguishing of reviewer profiles) proved the best option.

- **Empirical evaluation** of the proposed method based on four open-source projects: Android (~ 5 500 reviews), LibreOffice (~ 6 600 reviews), OpenStack (~ 9 500 reviews), Qt (~ 25 500 reviews). The experiments have shown that the bag of words method supported by the Tversky index is much faster than RevFinder (~ 60 times for Android, ~ 38 times for...
LibreOffice, ~ 180 times for OpenStack, and ~ 90 times for Qt – see Table 4.7, p. 42). Moreover, that method seems to provide more precise recommendations than Review Bot and RevFinder – two most prominent competitors⁴ – and the author of the dissertation claims that this improvement is statistically significant (p. 39). I am surprised by effectiveness of this solution.

It is a pity that the author did not explain why his method gives better results than those obtained by RevFinder. I know it can be difficult but some people managed to provide a persuasive explanation for surprising numeric results. For instance, P. Thongtanunam in her PhD thesis ([5], Sec. 5.7.1, p. 96) gave a reasonable explanation why her RevFinder is more effective than Review Bot:

‘... the performance of ReviewBot would be limited due to the small amount of review history. [...] We observed that 70%-90% of lines of code are changed only once, indicating that MCR tools [MCR stands for Modern Code Review – JN] have a short history of the changed line level. Hence, ReviewBot achieved the low performance in the context of our study.’

I am very curious if something similar could be presented for the solution contained in Chapter 4 of the PhD thesis under review. Maybe the weakness of RevFinder is the way in which the similarity between two file paths is computed and by using bag of words this weakness is eliminated?

3. Identification of buggy files based on bug reports

When a bug (or unexpected behaviour) is noticed by a user he or she is expected to submit a bug report. Then one of developers is assigned to fix the bug. To do it one needs (amongst others) to identify the files that must be corrected. In Chapter 5 it is assumed that a software project is equipped with two repositories:

- **Code repository**: It is a set of files containing the source code along with their previous versions and some metadata (e.g., the time a given version of a file was created, who was the author etc.);
- **Bug reports repository**: A set of bug reports concerning the project along with their metadata (e.g., the time a given bug report was submitted).

The above assumption is reasonable as version management systems (e.g. Git) and bug tracking tools (e.g. Bugzilla) are very popular.

To help the developers with identifying buggy files the author decided to work on the following problem:

**IBF Problem** (Identification of Buggy Files). Given a bug report concerning a software project, the code repository corresponding to it, and the bug reports repository, identify the files that must be changed to fix the bug.

It appears the most mature solution to the above problem was presented by Xin Ye et al. [8].

The most valuable contributions contained in Chapter 5 are the following:

- Modification of the method proposed by Ye et al. (the details are presented below).
- Experimental evaluation of the proposed method (comprising reproduction of some previous results which is a pretty time-consuming task). The results presented in Chapter 5 show that the proposed modifications are a real improvement to the solution proposed by Ye (but not only – in Table 5.7 one can find a comparison of effectiveness of the proposed modifications not only to Ye’s method but also to 5 other solutions). The scope of experimental work is impressive.

---

⁴ According to Google Scholar the papers announcing them got over 100 citations each.
Unfortunately, it is not clear if the results of those experiments are statistically significant with respect to the metrics presented in Table 5.7.

As regards modification of the method proposed by Ye et al., the two most important items are the following:

**Adding yet another feature.** Roughly speaking, the main idea of Ye et al. is to create for a given bug report \( b \) a ranking of source files \( s_1, s_2, \ldots \) by using a scoring function of the form:

\[
p_i(b, s) = \sum_k w_k \cdot \Phi_k(b, s)
\]

where:

- \( \Phi_k \) is a real number, called feature, which describes a particular aspect of similarity between the bug report \( b \) and a given source file \( s \) (for instance, \( \Phi_5(b, s) \) equals the inverse of the distance in months between bug report \( b \) and the last bug report concerning file \( s \));
- \( w_k \) is a real number, called weight, associated with feature \( \Phi_k \).

Then the IBF problem can be transformed into the learning-to-rank problem: Given the data contained in the code repository of the project and in the bug reports repository find features \( \Phi_1, \Phi_2, \ldots \) and their weights \( w_1, w_2, \ldots \) that would maximize a given quality metric (e.g., Accuracy@\( k \)) applicable to a ranking created by the scoring function created by those features and their weights.

Ye et al. proposed to use 19 features (they are presented also in the PhD thesis in Table 5.2 on p. 50). Feature \( \Phi_2(b, s) \) represents similarity between bug report \( b \) and textual descriptions fetched from the project API specification and concerning all the classes and interfaces contained in file \( s \) (more precisely, it is cosine similarity between bags of words representing those two texts). In Chapter 5 of the PhD thesis it is proposed to add one more feature, denoted as \( \Phi_2^* \), which ‘is based purely on the API derived from AST of the source code available before the bug report was reported’. The aim is to have more precise narrative text describing all the classes and interfaces contained in file \( s \).

**Proposing an ameliorated score function** for the training phase:

\[
p^*(b, s) = \sum_{i=1}^n w_i \cdot \Phi_i(b, s) + 1_{[s \in fixes(r)]} \cdot \max_{i=1 \ldots n} \Phi_i(b, s)
\]

The role of the right term is to increase the value of \( p^* \) for all fixed files – that’s an interesting proposal.

**4. Editorial remarks**

**Confusing definition of performance metrics.** On p. 10 of the PhD thesis two performance metrics are defined:

\[
\text{Precision}\!@\!k = \frac{|\text{top}(n) \cap \text{actual}(n)|}{|\text{top}(n)|} \quad \text{Recall}\!@\!k = \frac{|\text{top}(n) \cap \text{actual}(n)|}{|\text{actual}(n)|}
\]

where:

- \( \text{actual}(n) \) denotes ‘actual relevant \( n \) documents for specific query’,
- \( \text{top}(n) \) denotes ‘\( n \) documents retrieved for the same query’.

Surprisingly, parameter \( k \) has no impact on the value of \( \text{Precision}@k \) and \( \text{Recall}@k \). Moreover, the above definition suggests that there is no other relevant document outside top \( n \) documents.
recommended by a given recommendation algorithm – such an assumption would be too strong and unacceptable. To my understanding, it should be precisely stated that \( \text{Precision}@k \) and \( \text{Recall}@k \) are defined for a single pull request (request to review) and then they could be defined e.g. in the following way:

\[
\text{Precision}@k = \frac{\text{ActualTop}-k}{k} \quad \text{Recall}@k = \frac{\text{ActualTop}-k}{|\text{Actual}|}
\]

where \( \text{ActualTop}-k \) is the number of actual reviewers amongst top \( k \) reviewers listed by the reviewer recommendations algorithm for the pull request at hand and \( |\text{Actual}| \) is the number of all actual reviewers for that pull request.

There is a similar problem with the definition of \( \text{Accuracy}@k \) presented on p. 11:

\[
\text{Top } K = \# \text{ at least one correct in top } k, \quad \text{Accuracy}@k = \frac{\text{Top } K}{|Q|}.
\]

\( \text{Top } K \) sounds like a Boolean value and dividing it by a natural number is confusing (it remains confusing even if one assumes \( \text{Top } K \) returns 0 or 1 as it divides result of one recommendation by the number of recommendations \( |Q| \)). Why not to re-use the definition of \( \text{Top-k accuracy} \) proposed by P. Thongtanunam et al. [4]:

\[
\text{Top-k accuracy}(R) = \frac{\sum_{r \in R} \text{isCorrect}(r, \text{Top}-k)}{|R|} \times 100\%
\]

where \( R \) is a set of reviews (pull requests) and \( \text{isCorrect} \) returns 1 if at least one actual reviewer participating in review \( r \) was listed on the ranking \( \text{Top}-k \) for this review and 0 otherwise?

**Confusing multiset union operator.** Reviewer profile is defined in terms of multiset operations (Sec. 4.4.1). Those operations are not presented in the PhD thesis; instead the author refers the reader to D. Knuth’s monograph (Sec. 4.6.3, p. 483, Exercise 19) and a paper written by D. Singh et al. (p. 31):

*We follow the semantics of multisets from [70, 134].*

Unfortunately, in both sources there are two unions defined over multisets \( A, B \): \( A \uplus B \), \( A \cup B \) and it is not clear which of them the author has in mind. The example of page 31 implies the former (see e.g. the value of \( P_{a}(2) \) and the number of occurrences of \( \text{java} \)) while symbols used to define \( m(C') \) suggest the latter:

\[
m(C') = \bigcup_{r' \in C'} m(r')
\]

As the symbol \( \uplus \) is not easy to obtain, it is pretty understandable to use a simpler one but then a ‘re-definition’ would be necessary along with a short explanation.

**A mix-up of data coming from different metrics.** Table 4.5 (a copy is presented in Fig. AAA) contains important data about effectiveness of the proposed method and its competitors, RevFinder and Review Bot. The data about RevFinder and Review Bot are marked with yellow background and they have been ‘imported’ to the PhD thesis from the paper written by P. Thongtanunam et al. [4]—compare Fig. BBB. The problem is inconsistency. In Table 4.5 all the data in
columns Top1 through Top10 are labelled *Recall*, including the rows corresponding to RevFinder and Review Bot although those numbers in fact represent *accuracy* (in Fig. AAA and BBB those labels are taken in red ovals). I was terrified when I noticed this mistake as the definitions of recall and accuracy are different – see Fig. CCC. Fortunately, the problem is not as serious as it appears to be. As shown by Thongtanunam et al. ([4], Table I), average number of reviewers per review is close to 1 (1.01 for LibreOffice, 1.06 for Android, 1.07 for Qt, and 1.44 for OpenStack). It means that in most cases the number of actual reviewers (denoted in the PhD thesis by \( |\text{actual}(n)| \)) equals 1 and then recall equals accuracy. It is a pity that this explanation has been omitted in the PhD thesis.

Fig. AAA. Table 4.5 from the PhD thesis (p. 40; the *yellow highlights* have been added by the reviewer).
TABLE IV: The results of top-accuracy of our approach RevFinder and a baseline ReviewBot for each studied system. The results show that RevFinder outperforms ReviewBot.

<table>
<thead>
<tr>
<th>System</th>
<th>REVINDER</th>
<th>Top-1</th>
<th>Top-3</th>
<th>Top-5</th>
<th>Top-10</th>
<th>REVIEWBOT</th>
<th>Top-1</th>
<th>Top-3</th>
<th>Top-5</th>
<th>Top-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>46 %</td>
<td>71 %</td>
<td>79 %</td>
<td>86 %</td>
<td>21 %</td>
<td>29 %</td>
<td>29 %</td>
<td>29 %</td>
<td>29 %</td>
<td></td>
</tr>
<tr>
<td>OpenStack</td>
<td>38 %</td>
<td>66 %</td>
<td>77 %</td>
<td>87 %</td>
<td>23 %</td>
<td>35 %</td>
<td>39 %</td>
<td>41 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qt</td>
<td>20 %</td>
<td>54 %</td>
<td>41 %</td>
<td>69 %</td>
<td>19 %</td>
<td>26 %</td>
<td>27 %</td>
<td>28 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LibreOffice</td>
<td>24 %</td>
<td>47 %</td>
<td>59 %</td>
<td>74 %</td>
<td>6 %</td>
<td>9 %</td>
<td>9 %</td>
<td>10 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. BBB. The data ‘imported’ by the author from [4] (p. 148) represent accuracy, not recall.

\[
\text{Recall@k} = \frac{|\text{top}(\eta) \cap \text{actual}(\eta)|}{|\text{actual}(\eta)|} \\
\text{Top-k accuracy}(R) = \frac{\sum_{r \in R} \text{isCorrect}(r, \text{Top-k})}{|R|} \times 100%
\]

Fig. CCC. The definition of recall provided on p. 10 of the PhD thesis (a) and the definition of accuracy used by Thongtanunam et al. in [4] to compute the numbers presented in Fig. BBB (b).

I am surprised with lack of analysis of threats to validity in Chapter 4. Nowadays it is a must in empirical research. Such an analysis is provided in Chapter 5, so the author knows what it is about.

Chapter 5 is understandable only after reading first the paper by Xin Ye et al. [8]. The main reason is that the features listed in Table 5.2 are left unexplained. Moreover, the notion of fold appears in Sec. 5.4 (p. 50) but is (very briefly) explained in Sec. 5.5 (p. 51). Also the motivation behind introduction the feature $\Phi_2$ could be presented in a more readable way (an example would be very welcomed).

5. Knowledge of the candidate

The candidate exhibits extensive knowledge of the area of Software Engineering and Data Science. He knows very well modern approaches to software development and especially in the context of open source software. His special area of interest is mining software repositories and he has acquired many skills useful in this area (natural language processing, machine learning methods and tools). The bibliography and the related work sections of the thesis also confirm that he has pretty deep knowledge in the subject area. Nevertheless, a few references are missing, e.g. [5], [3], and [7]

6. Summary

I find this body of work to be well conducted. It presents an interesting contribution to the field of Mining Software Repositories, and therefore to Software Engineering. In my opinion it is of sufficiently high quality for a PhD dissertation. The candidate has proved be able to conduce scientific work.

References


