Dr hab. inż. Jerzy Nawrocki, prof. PP
Instytut Informatyki
Politechnika Poznańska
jerzy.nawrocki@put.poznan.pl

Poznań, 2020-12-31

Review of Ph.D. dissertation by

Mikołaj Fejzer

entitled:

Mining Software Repositories for Code Quality

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 (**R**ecommendation of **C**ode **R**eviewers). 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[1] 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)[2].

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*.[3] 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 α = 0, β = 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

[1] Review time for the four projects used in the evaluation experiment is presented in Fig. 4.1 on page 38 as a bar chart but this information is NOT used by the proposed solution neither is it by the experiment setup.
[2] According to the problem statement (p. 29) nomination of reviewers is done by '*using the information on the reviewer's past work and on the commit itself*'. This formulation is not precise enough as '*past work*' might comprise code written by the prospective reviewer (this is not the case) and '*information*' could mean any information including review time and quality of reviews.
[3] Longest Common Prefix, Longest Common Suffix, Longest Common Substring, and Longest Common Subsequence + a procedure of combining their results.

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 competitors[4] – 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 repositor**y: 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 (**I**dentification of **B**uggy **F**iles). 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, … by using a scoring function of the form:

$$p_i(b,s) = \sum_k w_k \cdot \phi_k(b,s)$$

where:

Φ_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 Φ_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 Φ_1, Φ_2, … and their weights w_1, w_2, … 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 Φ_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 Φ_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:

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

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:

$$Precision@k = \frac{|top(n) \cap actual(n)|}{|top(n)|} \qquad Recall@k = \frac{|top(n) \cap actual(n)|}{|actual(n)|}$$

where:

- *actual*(*n*) denotes '*actual relevant n documents for specific query*',
- *top(n)* denotes '*top n documents retrieved for the same query*'.

Surprisingly, parameter *k* has no impact on the value of *Precision@k* and *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 *Precision@k* and *Recall@k* are defined for a single pull request (request to review) and then they could be defined e.g. in the following way:

$$Precision@k = ActualTop\text{-}k \;/\; k \qquad\qquad Recall@k = ActualTop\text{-}k \;/\; |Actual|$$

where *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 |*Actual*| is the number of all actual reviewers for that pull request.

There is a similar problem with the definition of *Accuracy@k* presented on p. 11:

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

Top K sounds like a Boolean value and dividing it by a natural number is confusing (it remains confusing even if one assumes *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 *Top-k accuracy* proposed by P. Thongtanunam *et al*. [4]:

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

where R is a set of reviews (pull requests) and *isCorrect* returs 1 if at least one actual reviewer participating in review r was listed on the ranking *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):

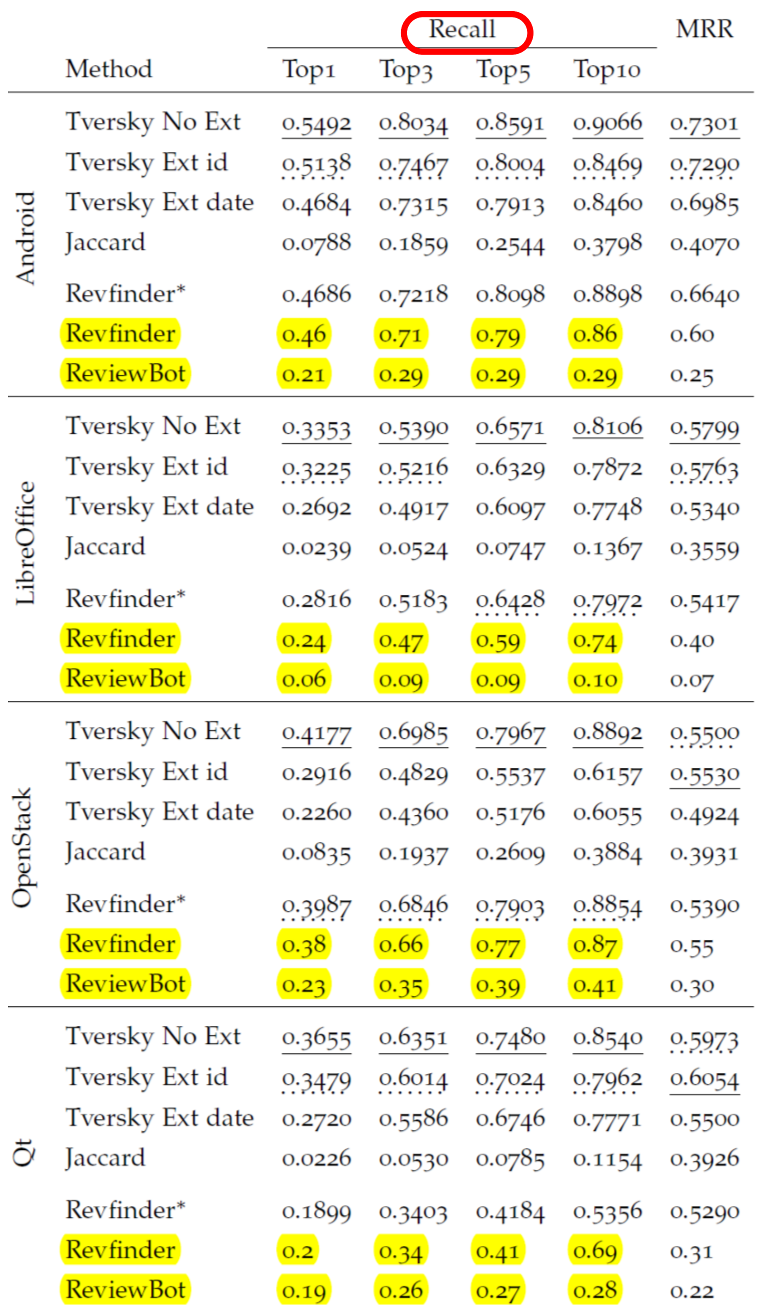
We follow the semantics of multisets from [70, 134].

Unfortunately, in both sources there are two unions defined over multisets A, B: A ⊎ B, A ∪ 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 *java*) while symbols used to define $m(C^t)$ suggest the latter:

$$m(C^t) = \bigcup_{f_i^t \in f(C^t)} m(f_i^t)$$

As the symbol ⊎ 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



		Recall				MRR
	Method	Top1	Top3	Top5	Top10	
Android	Tversky No Ext	0.5492	0.8034	0.8591	0.9066	0.7301
	Tversky Ext id	0.5138	0.7467	0.8004	0.8469	0.7290
	Tversky Ext date	0.4684	0.7315	0.7913	0.8460	0.6985
	Jaccard	0.0788	0.1859	0.2544	0.3798	0.4070
	Revfinder*	0.4686	0.7218	0.8098	0.8898	0.6640
	Revfinder	0.46	0.71	0.79	0.86	0.60
	ReviewBot	0.21	0.29	0.29	0.29	0.25
LibreOffice	Tversky No Ext	0.3353	0.5390	0.6571	0.8106	0.5799
	Tversky Ext id	0.3225	0.5216	0.6329	0.7872	0.5763
	Tversky Ext date	0.2692	0.4917	0.6097	0.7748	0.5340
	Jaccard	0.0239	0.0524	0.0747	0.1367	0.3559
	Revfinder*	0.2816	0.5183	0.6428	0.7972	0.5417
	Revfinder	0.24	0.47	0.59	0.74	0.40
	ReviewBot	0.06	0.09	0.09	0.10	0.07
OpenStack	Tversky No Ext	0.4177	0.6985	0.7967	0.8892	0.5500
	Tversky Ext id	0.2916	0.4829	0.5537	0.6157	0.5530
	Tversky Ext date	0.2260	0.4360	0.5176	0.6055	0.4924
	Jaccard	0.0835	0.1937	0.2609	0.3884	0.3931
	Revfinder*	0.3987	0.6846	0.7903	0.8854	0.5390
	Revfinder	0.38	0.66	0.77	0.87	0.55
	ReviewBot	0.23	0.35	0.39	0.41	0.30
Qt	Tversky No Ext	0.3655	0.6351	0.7480	0.8540	0.5973
	Tversky Ext id	0.3479	0.6014	0.7024	0.7962	0.6054
	Tversky Ext date	0.2720	0.5586	0.6746	0.7771	0.5500
	Jaccard	0.0226	0.0530	0.0785	0.1154	0.3926
	Revfinder*	0.1899	0.3403	0.4184	0.5356	0.5290
	Revfinder	0.2	0.34	0.41	0.69	0.31
	ReviewBot	0.19	0.26	0.27	0.28	0.22

Fig. AAA. Table 4.5 from the PhD thesis (p. 40; the yellow highlights have been added by the reviewer).

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 $|actual(n)|$) equals 1 and then recall equals accuracy. It is a pity that this explanation has been omitted in the PhD thesis.

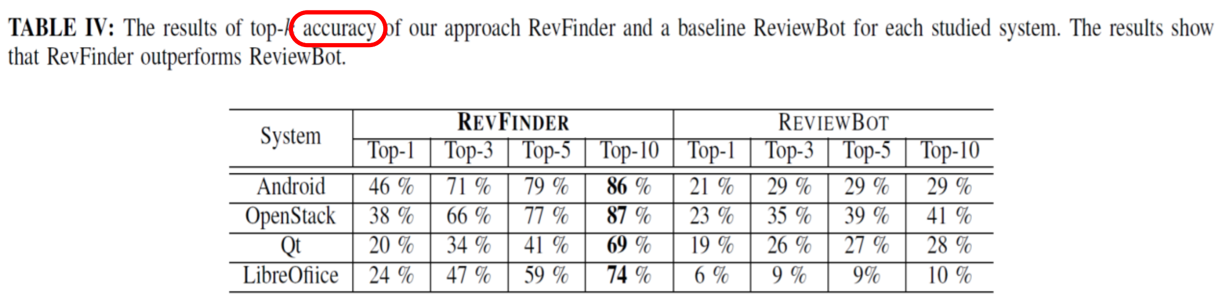


TABLE IV: The results of top-k accuracy of our approach RevFinder and a baseline ReviewBot for each studied system. The results show that RevFinder outperforms ReviewBot.

System	REVFINDER				REVIEWBOT			
	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
Android	46 %	71 %	79 %	**86** %	21 %	29 %	29 %	29 %
OpenStack	38 %	66 %	77 %	**87** %	23 %	35 %	39 %	41 %
Qt	20 %	34 %	41 %	**69** %	19 %	26 %	27 %	28 %
LibreOfiice	24 %	47 %	59 %	**74** %	6 %	9 %	9%	10 %

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

$$Recall@k = \frac{|top(n) \cap actual(n)|}{|actual(n)|}$$

(a)

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

(b)

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 Φ_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

[1] https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

[2] https://octoverse.github.com/ (visited on 2020-09-14).

[3] Shade Ruangwan, Patanamon Thongtanunam, Akinori Ihara, Kenichi Matsumoto, *The impact of human factors on the participation decision of reviewers in modern code review*, Empirical Software Engineering (2019) 24: 973–1016.

[4] P. Thongtanunam, Ch. Tantithamthavorn, R. G. Kula: *Who Should Review My Code?*, SANER 2015, Montréal, Canada, 141-150.

[5] P. Thongtanunam, *Studying Reviewer Selection and Involvement in Modern Code Review Processes*, Doctoral Dissertation, Nara Institute of Science and Technology September 5, 2016.

[6] https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/

[7] Yaojing Wang, Feng Xu, Yuan Yao, Yong Wu, *Detecting Buggy Files based on Bug Reports: A Random Walk Based Approach*, Proceedings of the 7th Asia-Pacific Symposium on Internetware, 62-69, 2015.

[8] Xin Ye, Razvan C. Bunescu and Chang Liu, *Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation*, IEEE Transactions on Software Engineering 42.4 (2016), pp. 379–402