Poznan University of Technology
Faculty of Computing and Telecommunications
Institute of Computing Science

Master Thesis

Recommendation of code reviewers for GitHub projects

Sebastian Maciejewski,
Jan Techner

Thesis advisor: prof. dr hab. inż. Jerzy Nawrocki

Poznań, 2021

Abstract

Context Automatic code reviewer recommendation is a developing field of study in which new methods are often proposed. Authors believe that it is important to check whether the proposed approaches to determining quality of those methods are justified and correct.

Aim The aim of this thesis is to review state-of-the-art in the field of automatic code reviewer recommendation. This research focuses on algorithm efficacy metrics and criteria for assessing recommendation quality.

Method Related work in the field is reviewed in the process of systematic literature review. Some of the efficacy metrics have been used in a replication of experiment described in *"Profile based recommendation of code reviewers"* (Fejzer et al. 2018).

Results Recreation of the experiment yielded results that differ from the ones described in original paper. Various possible causes of this were described, including bugs in the original implementation, difference in implemented metrics and algorithm due to lack of precise description of some steps in the algorithm.

Conclusions The currently used metrics seem sufficient to measure the efficacy of solutions for which assessment criteria are based on historical reviewer choices. However, the criteria itself are rather simple and can lead to misleading suggestions in many real-life situations. Therefore, authors suggest coming up with new, more complex assessment criteria taking into account not only the historical assignments, but the quality of those assignments as well.

Table of contents

List of abbreviations

CRR - Code Reviewer Recommendation

PR - Pull Request

SLR - Systematic Literature Review

MRR - Mean Reciprocal Rank

1 Introduction

1.1 Modern code review

In march 2020, as developers worldwide went remote, the traffic in open source projects on GitHub soared. An increase of 40% year on year was noted in the number of open source repositories created (GitHub 2020). This was not an anomaly, the continued growth of open source goes back years and pandemic has just accelerated this trend. In an environment where a huge amount of people from different backgrounds come together remotely to create, the code quality and project maintainability become an important issue.

This is why one of the crucial parts of the distributed software development is the process of code review. Eric S. Raymond in his essay "The Cathedral and the Bazaar" defined the Linus's Law which states that "given enough eyeballs, all bugs are shallow" (Raymond 1999). In other words, the developers involved in the project can look at the code from the different perspective, give a valuable insight and find the bugs only by looking at the code.

Although it might seem like choosing a code reviewer becomes easier, since there are more contributors to choose from, this is usually not a case. Studies found that in popular open source projects, the code review process takes 12 days longer for the reviews with code-reviewer assignment problem (Thongtanunam et al. 2015). As finding a competent contributor, who can perform the review in a timely manner is a difficult task, numerous algorithms were designed to facilitate this process.

While this kind of algorithms are still a relatively novel field of study, there is an abundance of articles on the matter. We believe that thorough analysis of the current state of knowledge is a useful basis for further work in this field and paves the way for new, more optimized solutions in the future.

1.2 The aim and scope of the thesis

The aim of this thesis is to review the state-of-the-art in the field of automatic code reviewer recommendation (CRR) with particular focus on the algorithm efficacy measurement and criteria for assessing recommendation quality.

A foundation of such analysis is a systematic literature review (SLR) we performed using Web of Science, further described in Section 2.

In order to review the assessment criteria, efficacy metrics and potential threats to validity, we decided to repeat the experiment described in *"Profile based recommendation of code reviewers"* (Fejzer et al. 2018). The experiment was run on a set of real pull requests (PR), its design and results can be found in sections 3. and 4. respectively.

The scope of this thesis has been divided between the authors in the following way:

1. Performing a systematic literature review (Jan Techner and Sebastian Maciejewski)
2. Implementation of profile based algorithm and executing the experiment (Jan Techner)
3. Analysis of assessment criteria, efficacy metrics and threats to validity (Jan Techner and Sebastian Maciejewski)

2 Review of the field

In order to gain perspective on the state-of-the-art, we decided to find and review relevant related work in the process of systematic literature review (SLR). This way we could review a wide spectrum of scientific papers quickly, thus minimizing the risk of coming to false conclusions by choosing a tendentious set of papers.

2.1 Systematic literature review (SLR) protocol

2.1.1 Background and aim

The problem of choosing the right code reviewer is not trivial and there are many aspects that may affect the correctness of the choice, e.g. the technical expertise, the seniority level or social dynamics within the team. The problem is even more severe in the open source projects where contributions are made by the developers that may not know each other at all. In such case, making an informed and accurate choice of reviewer is nearly impossible. There are many project-specific aspects that may influence the choice in a positive way resulting in a thorough review, but there is also the risk that in some cases the choice would be wrong and eventually may result in releasing the code with undiscovered bugs. On top of that, inaccurate reviewer choice might foster propagation of bad practices and messy code, especially in projects without a strict policy of abiding by contributing guidelines.

The aim of the SLR is to review the state-of-the-art of CRR methods proposed so far in order to discover what are the assessment criteria for the results of those methods and to find the most relevant metrics used to describe their efficacy. This in fact means investigating how is the correctness defined in those methods - what decides whether a choice of a reviewer is appropriate and how to measure the method efficacy based on those choices.

2.1.2 Research questions and query

Given the above aim we came up with the following research questions (RQ):

> RQ1. What assessment criteria of the results are used in the state of the art methods proposed so far?
>
> RQ2. What metrics to measure the efficacy of the code reviewer choice were proposed so far?

Our bibliographic database of choice is Web of Science and we decided to search it with the following query:

```
TS = (
        ((code review*) OR (code-review*)) AND
        ((recommendation) OR (suggestion))
    )
AND SU = (Computer Science)
AND LANGUAGE: (English)
```

It's worth noting that the query does not limit the results to a specific time frame, thus its results might change in the future. The results described in this thesis were obtained from Web of Science in March 2021.

2.1.3 Validity evaluation

Given the fact that the CRR problem is highly specific and quite novel field of study, there is plenty of keywords unique enough to easily identify a relevant scientific paper. The use of phrases like *code review** combined with *recommendation* ensures only relevant papers will be found, as this kind of nomenclature is uncommon and it is very unlikely for a paper on this topic to avoid using such words. As for the bibliographic database, Web of Science was our choice because it has a vast library of papers and a rich query language that allows gathering the papers more precisely.

Completeness check

In order to verify whether the results obtained by querying Web of Science were sufficient, we decided to perform an additional search for relevant papers and articles using Google Scholar. The query we used was "code review recommendation suggestion", and checked its 100 first results, taking into account only articles and scientific papers, as Google Scholar tends to return single citations as results. The query, just like in the case of Web of Science, was not bound by any specific time frame.

In the 100 first results we found no papers that seemed relevant (based on their titles and abstracts) that were not found by the Web of Science query. Additionally, Google Scholar query results were quite random and did not contain many of the papers Web of Science query yielded, instead the results were often completely irrelevant to the field of code reviewer recommendation. This ensured us in belief, that there are no relevant papers that we missed by using Web of Science, since its querying capabilities clearly are good compared to other services of this nature, like Google Scholar.

2.1.4 Design of the Data Extraction Table

The data extraction table used to gather the answers to proposed research questions has been designed to include the following columns:

- Paper ID
- Motivation
- Assessment criteria (RQ1)
- Metrics (RQ2)
- Remarks

The complete data extraction table and list of papers selected for SLR can be found in a GitHub repository (Maciejewski and Techner 2021) and in the appendix.

2.1.5 Filtering procedure

As the results obtained from Web of Science using our query exceeded 200 papers, performing additional filtering on the results was necessary. In order to minimize the risk of accidentally filtering out relevant papers, we decided to refine the results in three steps.

1. Filtering by title
2. Filtering by abstract
3. Filtering by introduction and conclusions

In the first step, all the papers with titles indicating irrelevance to our research questions were removed from the results. This step, just like the ones following it, was not automated in any way and was performed by two people to ensure diligence and minimize the risk of removing relevant work from the results by mistake. It was followed by a brief analysis of each paper's abstract, which helped to further narrow the list of results. Finally, all the remaining papers and articles were studied based on their

introduction and conclusion sections in order to verify whether the work described in them could help with answering the posed research questions. This kind of multi-layer filtering provided us with a set of papers that were highly likely to contain information relevant for our research.

2.2 Systematic literature review results

The output of the query consisted of 206 articles and papers. Filtering procedure narrowed down the resulting set in the following way: 44 papers were marked as relevant after filtering by title and 26 were chosen by looking at abstract. The third step did not filterer out any paper, therefore we decided to mark the remaining 26 papers as the final set.

2.2.1 Assessment criteria of the algorithms' results

During the analysis of papers and articles from the result set, it became apparent that a vast majority of methods use historical reviewer assignments as a key criteria to assess the correctness of the results. What's more, the proposed algorithms are often based on and constructed upon the historical choices of the reviewers. This might in fact lead to numerous validity threats. Such approach, however logical it appears, might be flawed in its very essence, because there is no guarantee that the actual reviewer choices were good.

Let's consider a project in which a team of contributors works together in a single company and their workflow (and reviewer choices) are dependant on outside factors like management, workload and schedule in other projects etc. This means that the reviewer choices in such project might often be sub-optimal, as the best reviewer might be e.g. on vacation or occupied in another project. Similar situations often happen in open source projects as contributors come and go, taking their expertise with them. Therefore a method basing its results on past reviewer choices is subject to the risk of error because of such validity threats.

2.2.2 Efficacy metrics

As the most common assessment criteria are heavily based on the conformance with the real-life reviewer choices, it comes as no surprise that the metrics used in recommendation efficacy measurement are ones used for set comparison. Overall, most papers studied in the scope of this SLR use some subset of the following common metrics:

- Precision
- Recall
- F-Measure
- Mean Reciprocal Rank

Metrics listed above are defined in section 3.2. In some cases, authors define an *accuracy* metric, which is equal to recall in its definition.

Keeping this in mind we made an attempt to recreate an experiment originally conducted for the purpose of measuring efficacy of a CRR algorithm.

3 Experiment design

3.1 Experiment goal

The basic idea behind the experiment is to investigate metrics of efficacy of code reviewer recommendation and threats to validity of experiments measuring it. In order to do that, we decided to repeat the experiment described in *"Profile based recommendation of code reviewers"* (Fejzer et al. 2018).

The original authors' goal was the empirical evaluation of a method proposed by them and comparing its effectiveness against the state-of-the-art of methods - *ReviewBot* (Balachandran 2013) and *RevFinder* (Thongtanunam et al. 2015).

Since the code reviewer recommendation algorithm's output is a ranked list of reviewers, we decided that an important part of the experiment evaluation is comparing the results yielded by the tested algorithm with the actual reviewer choices made by contributors. This is a standard approach in CRR algorithms measurement, used as well by the authors of original experiment, thus the main experiment results for comparison are the indicators taking into account the real-life reviewer choices. Such analysis might help better understand whether the metrics describing method efficacy are sufficient to model the reality of code reviewer problem.

3.2 Applied metrics

Similar to the original experiment, we measured the following performance indicators:

- Precision, defined as:

$$precision(R, n) = \frac{\sum_{i=1}^{|R|} |actual(PR_i) \cap top(PR_i, n)|}{\sum_{i=1}^{|R|} |top(PR_i, n)|} \tag{1}$$

- Recall, defined as:

$$recall(R, n) = \frac{\sum_{i=1}^{|R|} |actual(PR_i) \cap top(PR_i, n)|}{\sum_{i=1}^{|R|} |actual(PR_i)|} \tag{2}$$

- F-Measure, defined as:

$$F\text{-}measure(R, n) = \frac{2 * precision(R, n) * recall(R, n)}{precision(R, n) + recall(R, n)} \tag{3}$$

where

- R represents the repository defined as an ordered list of pull requests PR_i
- PR_i is a i-th pull request to be reviewed
- n is the number of recommended reviewers (ranging from 1 to 10)
- $actual(PR_i)$ is the set of all actual reviewers for the i-th pull request PR_i
- $top(PR_i, n)$ is the set of n top reviewers recommended by the algorithm for the i-th pull request PR_i

Those indicators are in line with what was used in the original experiment, however the original authors additionally calculated the mean reciprocal rank (MRR), defined as

$$MRR = \frac{1}{|R|} \sum_{i=1}^{|R|} \frac{1}{rank(top(PR_i, 10)} \tag{4}$$

where $rank$ is the index of the first actual reviewer in the recommendation $top(PR_i, 10)$. The reason for calculating the MRR value was performing a comparison between the profile based method proposed by authors and the *ReviewBot* and *RevFinder* methods.

We are not performing such comparison in the experiment, but the MRR is calculated in the results for the sake of diligence.

3.3 Experiment setup

Due to the fact, that we did not intend to measure execution speed, memory usage etc. in the experiment, the machine specification is not relevant to the experiment.

What is relevant however, is the technology which was used to implement the algorithm - in this case the profile based algorithm was implemented in Python 3.8.

This fact is significant, because the algorithm was implemented based on the description found in the paper and some programming constructs used in it could require additional implementation in certain languages. As Python is widely recognized as the go-to language for scientific experiments, we decided to choose it for the task. Authors of the original experiment used Python as well, but the code prepared for this experiment was not influenced by the original authors' code in any way.

In order to test the algorithm on additional data, we decided to implement a custom data provider. It is a Node.js application that fetches specific repository metadata via GitHub GraphQL API (GitHub n.d.), parses it to a specific JSON schema format and saves as a JSON file for the algorithms to use. The quality and correctness of data returned by the provider was first tested on a small repository containing just a few pull requests and commits.

3.4 Algorithm implementation

3.4.1 Implementation based on the paper

Our implementation is based on the algorithm described in the original paper (Fejzer et al. 2018), as it is called for the purpose of this thesis. In the paper the authors compared multiple variants of the algorithm by applying various modifications and possible enhancement. However, the variant without any modification (described in the original paper as *"Tversky No Ext"*) turned out to be the most effective thus this one was our variant of choice.

The algorithm is implemented in Python 3.8, however we decided to use an object oriented programming (OOP) approach as it allows to present the code in a clear, readable manner and also to avoid bugs resulting from spaghetti code which is often a case when using Python as a scripting language. The structure of the program is presented in the form of UML diagrams in 3.1 and 3.2. The code for implementation used in the experiment as well as the data provider can be found in a GitHub repository (Maciejewski and Techner 2021).

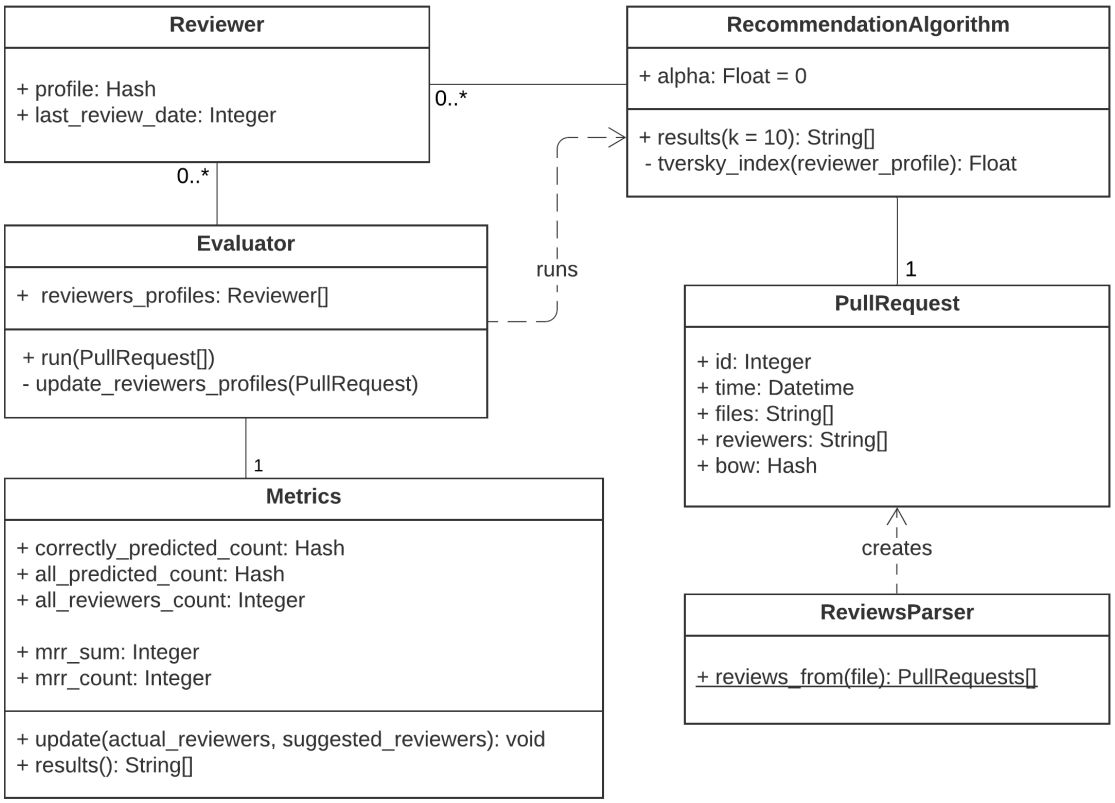

Figure 3.1: UML class diagram for the algorithm implementation

As can be seen in the sequence diagram (3.2) the algorithm consists of two main loops. First one is used to parse the repository's data from a JSON file and create a list of `PullRequest` objects. Every `PullRequest` contains the creation time, list of modified files, list of actual reviewers and the `BOW` attribute representing a bag of words created from the paths of modified files. The list of `PullRequests` is then sorted by creation time and passed to the `Evaluator` in order to perform the measurements. The `Evaluator` is responsible for managing reviewers' profiles, running the second loop and returning the final results. In the loop, consecutive `PullRequests` along with current reviewers' profiles are passed to the `RecommendationAlgorithm` which calculates the score for every reviewer. This is the crucial part of the algorithm - the reviewer's score in the context of the given `PullRequests` is calculated as the Tversky index between `PullRequests`'s `BOW` and reviewer's profile.

Afterwards the reviewers are sorted by their score and returned as the list of recommended reviewers for the given `PullRequests`. Then the list of recommended reviewers together with the list of actual reviewers are sent to the `Metrics` object which calculates and stores the results (*precision*, *recall* and *MRR*). Additionally, at the end of the iteration, the `Evaluator` updates the reviewers' profiles by the `PullRequests`'s `BOW`. After the loop, the `Evaluator` fetches the final metrics from the `Metrics` object and returns them to the main function.

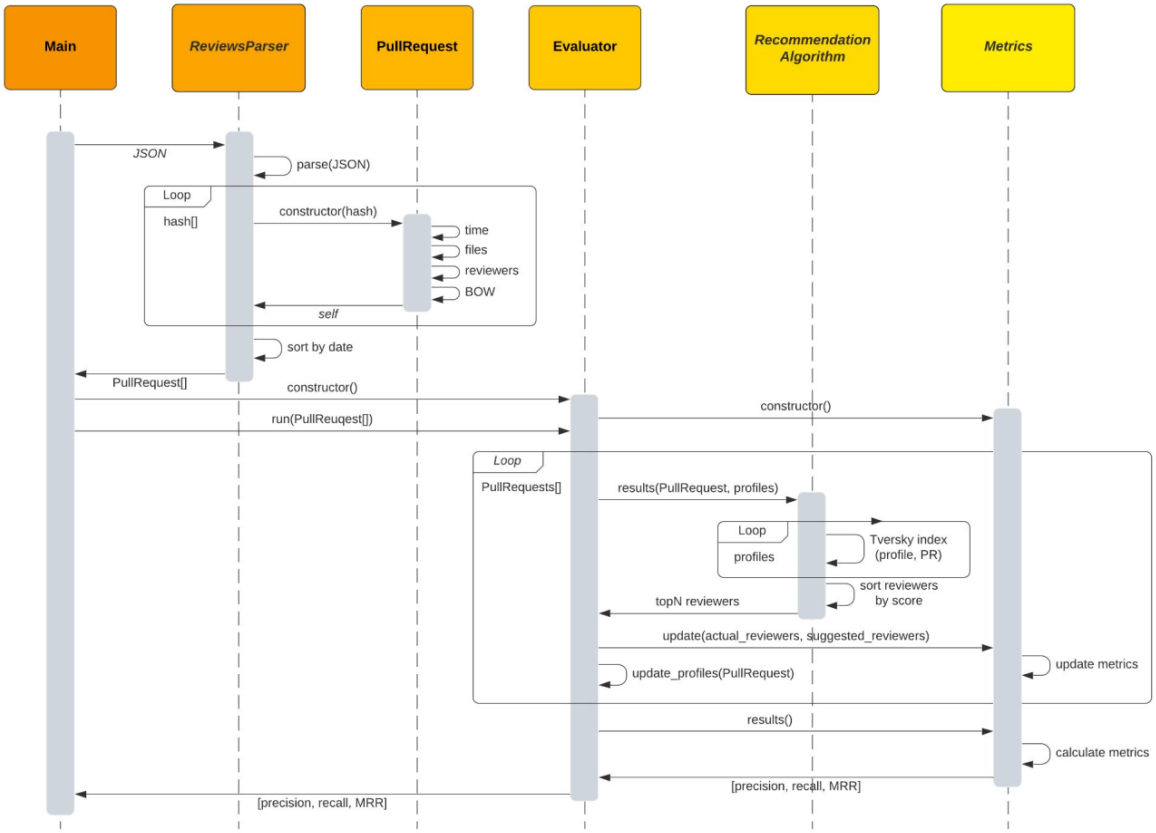

Figure 3.2: UML sequence diagram

3.4.2 Original authors' implementation

In the experiment we decided to run not only our implementation of the algorithm, but also the original implementation (as will be called for the purpose of this thesis) found in the repository on author's GitHub (Fejzer 2017). The repository contained three different variants of the algorithm but we chose the unmodified one as it was also the basis of our implementation of the algorithm. The original implementation was also created using Python language, but in the version 2.7.

There is no explicit reference to the repository in original paper, however we decided to run the found code anyways, as this way we can perform an additional comparison of results to get a perspective of how the original results were obtained.

3.5 Experiment dataset

In the original experiment, the algorithms were tested using a rather small set of large repositories - we decided that testing the algorithm effectiveness on such dataset might make it difficult to spot mistakes in yielded result. In order to avoid such mistakes, the algorithm was tested on several smaller repositories fetched with our custom data provider in the format corresponding with the one used in original experiment. Such approach made it possible to verify the algorithm output *"by hand"* and facilitated debugging.

The original dataset consisted of four big repositories:

- Android
- LibreOffice
- OpenStack
- Qt

Data for each repository is represented as a list of pull requests, each of whom consist, among other things, of list of changes, pull request status, authors id, review history with reviewer ids etc. List of pull requests for given repository is stored as JSON file.

In order to ensure a fair and meaningful comparison the algorithms were run with the exact same data as in the original experiments. The complete dataset can be found in the original *revfinder* repository (Thongtanunam 2014).

4 Experiment results

4.1 Results presentation

Due to the fact that we decided to run both original implementation (original) and our own implementation (new), we gathered two distinct data series. For each of the four repositories, the programs were run for top-k reviewers with k ranging from 1 to 10. The results for calculated precision, recall and MRR can be found in the tables 4.1, 4.2 and 4.3.

Precision	Android		LibreOffice		OpenStack		Qt	
	new	original	new	original	new	original	new	original
Top1	0.5579	0.5637	0.3453	0.3370	0.4217	0.4198	0.3661	0.3659
Top2	0.3826	0.3718	0.2367	0.2324	0.3379	0.3011	0.2723	0.2669
Top3	0.2905	0.2788	0.1851	0.1818	0.2816	0.2375	0.2192	0.2127
Top4	0.2316	0.2210	0.1566	0.1529	0.2432	0.1962	0.1827	0.1764
Top5	0.1929	0.1837	0.1359	0.1336	0.2135	0.1668	0.1569	0.1509
Top6	0.1662	0.1579	0.1204	0.1187	0.1905	0.1451	0.1374	0.1318
Top7	0.1462	0.1386	0.1087	0.1069	0.1717	0.1288	0.1221	0.1168
Top8	0.1307	0.1237	0.0995	0.0977	0.1567	0.1159	0.1100	0.1050
Top9	0.1183	0.1118	0.0912	0.0898	0.1435	0.1056	0.0996	0.0950
Top10	0.1078	0.1019	0.0845	0.0834	0.1328	0.0971	0.0913	0.0870

Table 4.1: Precision measurements for new and original implementation

Recall	Android		LibreOffice		OpenStack		Qt	
	new	original	new	original	new	original	new	original
Top1	0.5136	0.5189	0.3403	0.3316	0.2899	0.2902	0.3418	0.3412
Top2	0.6971	0.6773	0.4651	0.4559	0.4625	0.4146	0.5080	0.4972
Top3	0.7853	0.7536	0.5435	0.5334	0.5695	0.4834	0.6120	0.5932
Top4	0.8235	0.7858	0.6112	0.5962	0.6469	0.5256	0.6789	0.6545
Top5	0.8460	0.8056	0.6615	0.6501	0.7017	0.5522	0.7273	0.6988
Top6	0.8641	0.8209	0.7014	0.6912	0.7445	0.5713	0.7628	0.7312
Top7	0.8773	0.8321	0.7367	0.7246	0.7763	0.5868	0.7893	0.7543
Top8	0.8873	0.8401	0.7690	0.7550	0.8028	0.5985	0.8106	0.7731
Top9	0.8954	0.8467	0.7907	0.7789	0.8197	0.6083	0.8246	0.7857
Top10	0.8994	0.8500	0.8124	0.8019	0.8367	0.6164	0.8383	0.7978

Table 4.2: Recall measurements for new and original implementation

	Android		LibreOffice		OpenStack		Qt	
	new	original	new	original	new	original	new	original
MRR	0.6816	0.7326	0.4792	0.5797	0.5803	0.5497	0.5235	0.5971

Table 4.3: MRR measurements for new and original implementation

In addition to the results of original and new implementations, we also decided to include the results gathered and described in the original paper (paper). However, as the authors focused mainly on comparing various variants of the same CRR algorithm, the results were limited to the MRR and the recall for top 1, 3, 5 and 10 reviewers of the unmodified variant of the algorithm (*"Tversky No Ext"*).

The comparison between the results found in original paper and the ones received in the experiment is presented in the table 4.4.

		Recall				MRR
		Top1	Top3	Top5	Top10	
Android	new	0.5136	0.7853	0.8460	0.8994	0.6816
	paper	0.5492	0.8034	0.8591	0.9066	0.7301
	original	0.5189	0.7536	0.8056	0.8500	0.7326
LibreOffice	new	0.3403	0.5435	0.6615	0.8124	0.4792
	paper	0.3353	0.5390	0.6571	0.8106	0.5799
	original	0.3316	0.5334	0.6501	0.8019	0.5797
OpenStack	new	0.2899	0.5695	0.7017	0.8367	0.5803
	paper	0.4177	0.6985	0.7967	0.8892	0.5500
	original	0.2902	0.4834	0.5522	0.6164	0.5497
Qt	new	0.3418	0.6120	0.7273	0.8383	0.5235
	paper	0.3655	0.6351	0.7480	0.8540	0.5973
	original	0.3412	0.5932	0.6988	0.7978	0.5971

Table 4.4: Comparison with results found in paper

As the results from the table 4.4 are the most significant, they are presented on charts 4.1, 4.2, 4.3, 4.4 and 4.5 for the purpose of further analysis. The first chart, 4.1, does not present a distinct series for *TopN* reviewers, which is due to the nature of MRR metric. For this reason, MRR is plotted for all four repositories on a single chart, whereas in case of recall *Top1*, *Top3*, *Top5* and *Top10* series are plotted for every single repository.

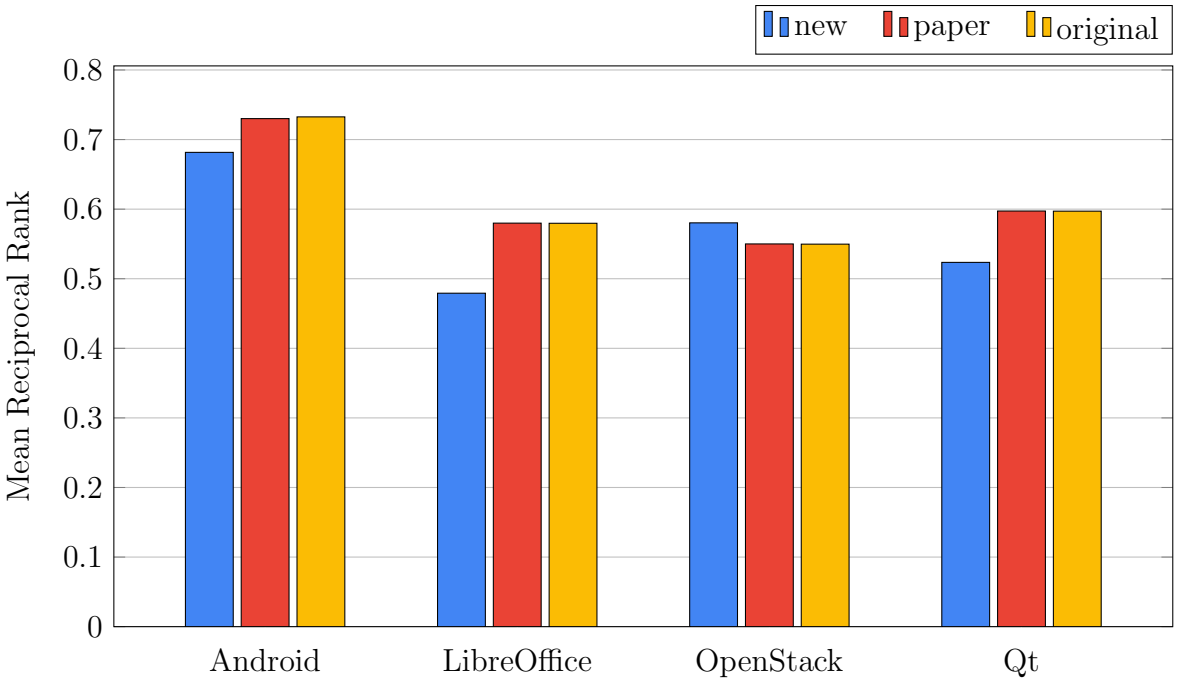

Figure 4.1: MRR comparison

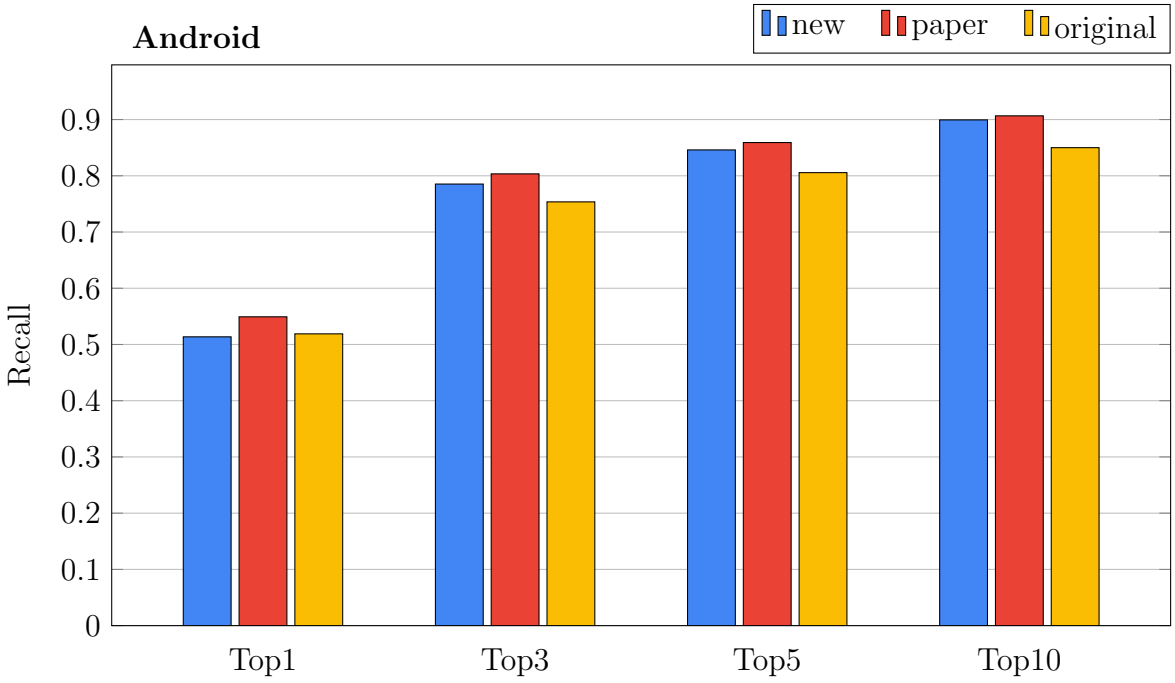

Figure 4.2: Recall comparison for Android repository

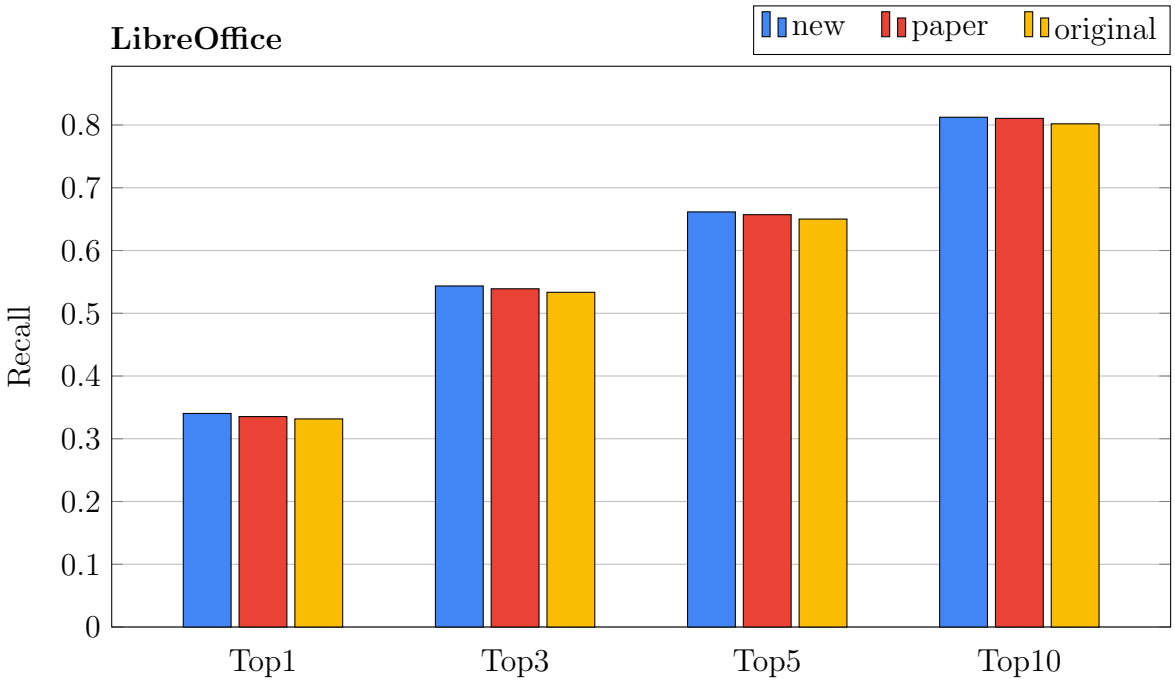

Figure 4.3: Recall comparison for LibreOffice repository

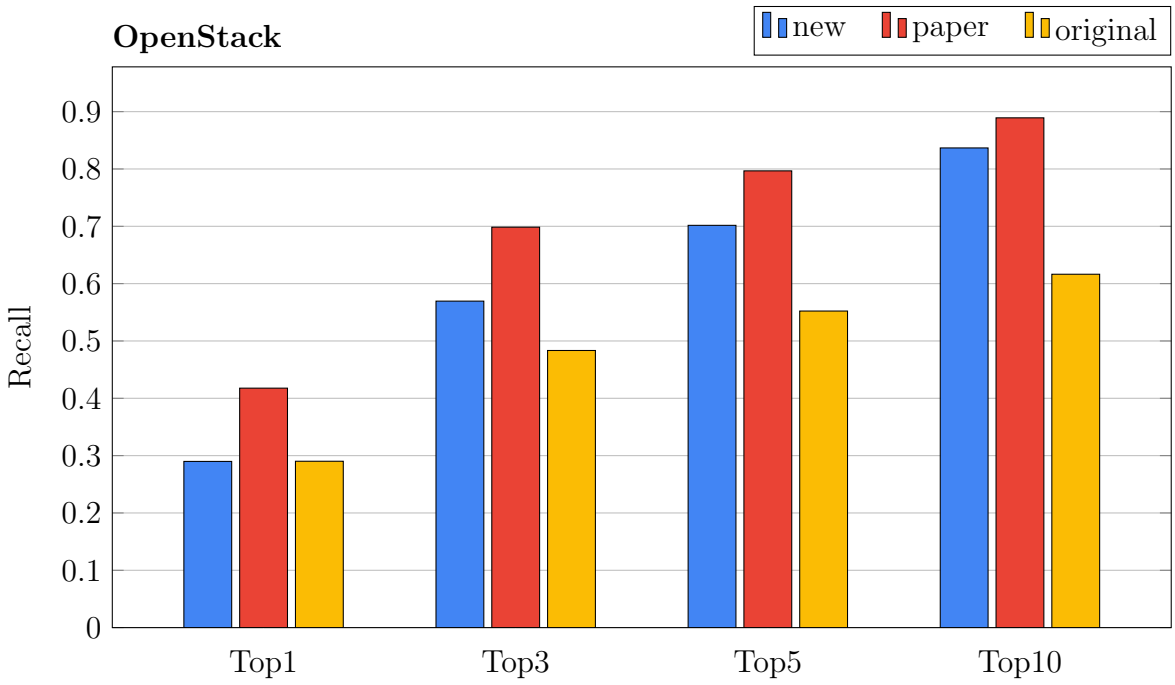

Figure 4.4: Recall comparison for OpenStack repository

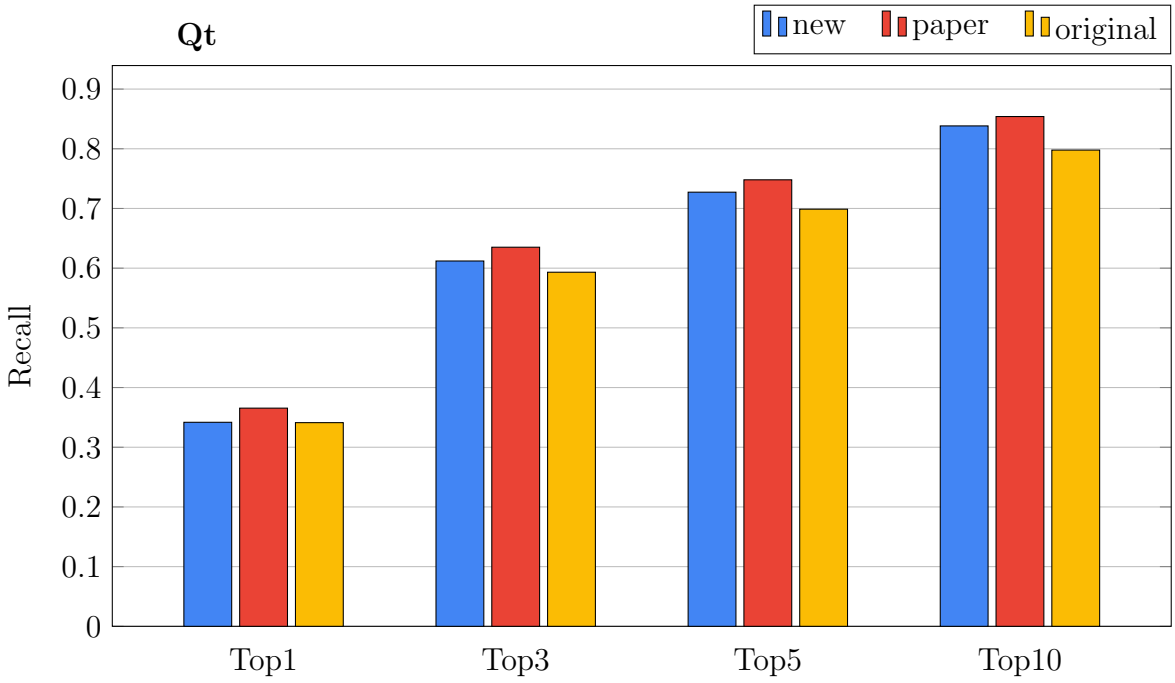

Figure 4.5: Recall comparison for Qt repository

4.2 Results interpretation

As can be seen in the above tables the results of our implementation of the algorithm are significantly different than the results of the original implementation and the results presented in authors' paper (Fejzer et al. 2018).

The data presented on the charts show some interesting patterns. The results of our implementation and the original implementation are almost the same for Top1 reviewers in every repository but rise as Top-k increases. The cause for this behaviour has been found and is described in the section 4.2.1.

Looking at the differences between the *new* and *paper* we can observe, that our implementation is significantly worse than the one described in the paper.

What is the most interesting, the biggest difference lies between the results presented in the paper and the results of the original implementation. This can be clearly observed in the OpenStack repository but is present also in other examined repositories. It is a strong indicator that the results presented in the original paper do not come from the implementation found in the authors' GitHub repository.

Another observation is that discrepancies between implementations differ between repositories. The results for the LibreOffice repository are almost the same in opposite to the OpenStack repository where the differences are noticeable.

4.2.1 Potential causes

Discrepancies between performance for different repositories may be caused by the specific characteristics of chosen repositories and indicate a presence of some kind of differentiating factors in the repositories. Such factors could include the amount of active contributors, pull requests and overall project stage as it relates to types of open issues, which might require different types of changes (Kallis et al. 2021).

The technologies used in the project might also be a factor, as some frameworks value convention over configuration which might lead to a lot of changes being made to files in similar location, no matter the type of pull request. Changes of such kind are a problem for CRR methods based on bag of words constructed from paths of files modified in the scope of pull request.

Implementation

As datasets used in the experiments were exactly the same, we came to the conclusion that the differences are caused by the implementation itself.

The analysis of the original implementation shows that the CRR algorithm has been implemented correctly, however there are several bugs in the metrics' calculation. The bugs were identified by a thorough analysis and debugging of the original code using a smaller test repository in which the metrics could be easily calculated by hand.

The first bug was found in the calculations of the precision and recall. In both equations that the implementation was based on (precision and recall), the nominator is the intersection of two sets - the actual reviewers set and the recommended reviewers set. The original implementation was iterating over all the suggested reviewers and checking if they are present in the actual reviewers set. However, after the first recommended reviewer found in the actual reviewers set, the loop execution was interrupted by the *break* operation, which ensured that no matter how many correct reviewers were suggested, the size of the intersection was maximally 1. This is the cause why the recall results are getting worse as the k in Top-k rises.

The second bug relates to the MRR metric defined in equation 4. The mean reciprocal rank in the problem of CRR is the average of the reciprocal ranks of results for a list of pull requests in a repository. Reciprocal rank is the reciprocal of the rank position of the first relevant reviewer in the list of reviewers recommended by the algorithm. When the recommendation list contains none of actual reviewers the reciprocal rank is equal to 0. Thus, every incorrect recommendation lowers the overall sum of reciprocal ranks and therefore lowers the MRR, which is in fact quite intuitive. In the original code, the recommendations with no correct reviewers were completely excluded from the mean (they were not included in the $|R|$). The correct implementation of MRR in our algorithm included the incorrect recommendations in the metric causing the MRR to be lower than the MRR calculated in the original experiment. This is clearly visible in the chart 4.1. In contrast to the found trend, the MRR calculated by our algorithm is higher for the OpenStack repository. We found no valid explanation for this unusual sample.

5 Conclusions

5.1 Assessment criteria review

As described in section 2.2.1, the assessment criteria based on historical choices of the reviewers might lead to suboptimal and sometimes even misleading results.

The majority of methods operate based on ex-ante assessment of historical reviewer choices correctness. This does not have to be the case, as the environment in which the code is being made varies greatly between repositories and is subject to constant change. In the case of nearly 60% open source projects studied in 2020, the turnover rate of contributors in the team exceeded 30% (Ferreira et al. 2020). Such dynamic changes suggest that discovering the factors of code reviewer choice today does not implicate that the reviewers chosen based on those factors will be optimal choices in the future, as the environment in which the factors were discovered has evolved.

There is also an important distinction between the problem of code reviewer recommendation in free, open source projects and projects maintained by or within a company. Because in the context of CRR algorithms, the problem is usually defined for an open source project, throughout the whole paper we considered all the aspects of CRR problem in this setting. There is more to this, as sometimes small open source project can foster a very homogeneous community with low turnover rate, because experienced developers tend to stick together between projects (Hahn et al. 2008). Such community of contributors might resemble a company more than an open project with lots of contributors who don't know each other very well, thus having more interactions outside of the repository, what might lead to assigning pull requests arbitrarily.

One might argue that despite those facts, treating the historical reviewer choices as optimal is the way to go, as it is unrealistic to assume that such intricate things as the team dynamics and overall environment can be captured and accounted for by an algorithm. This might be correct to an extent, since the problem of fading recommendation relevance can be countered by only taking into account recent code reviewer choices. Such approach has a single major flaw - it reduces the size of past pull

request pool, on which the recommendations will be made. In case of some methods, like e.g. the the *"Profile based recommendation"*, used in experiment we recreated, taking into account only limited amount of recent pull requests all but denies their purpose, as they rely on incrementally updated profile of contributors or other factor aggregated over the time.

Even though this kind of assessment criteria might be problematic, it could serve as a good basis for a composite approach - treating the recent reviewer choices as optimal, additionally considering a range of other, less obvious factors. Perhaps methods basing their recommendation on such composites could even be tailored specifically to the project or an organization's needs.

5.2 Possible assessment criteria improvements

Collaboration between people writing code is never a single dimensional thing, there is a whole lot more to it. When thinking about the CRR method assessment criteria as a composite, it might be easier to come up innovative ways of looking at CRR problem. Aspects such as contributor activity in recent days or most frequent issue respondents are just a fraction of what can be discovered in a project with software engineering methods and can influence the reviewer choice. Plenty of approaches to the problem account for those kinds of aspects, e.g. by adjusting the set of reviewers to choose from.

Among such approaches, extending beyond a conventional assessment criteria is the one to validate the quality of reviewer assignment by locating fix-inducing changes (Sliwerski et al. 2005) - the idea behind this is that if a pull request introduced a bug, then perhaps the reviewer recommended for it was not the best one available. This way, one can still base their method on actual reviewer choices, but purge the potentially bad, fix-inducing pull requests from the set of historical reviews. A set of actual reviewers prepared this way might then serve as a training or evaluation set for a CRR method.

Another way of approaching the problem of imperfect historical reviewer assignments could be a survey among developers after closing the pull request, asking them post-fact who they think would be the best reviewer for this pull request. Results of this survey could then shed a light on the quality of actual reviewer choices as an as-

sessment criteria and could be taken into account by CRR methods. This could mean e.g. training and evaluating a method only for pull requests where the contributors' choice in the survey was the actual reviewer.

An empirical approach to improving the assessment criteria could mean e.g. constructing a weighted composite value for each actual reviewer, taking into account various factors such as contributor workload at a time, number of issues etc. This would mean that there is no absolute value to compare the method result with, as results are based on weights and the weights itself could be changed and tweaked according to the project. Such problem would complicate the comparison of CRR methods but could also give teams a flexibility in terms of tailoring the methods to their own needs by teaching or evaluating it against empirically discovered set of weights.

5.3 Efficacy metrics review

In the situation where the comparison between the algorithm's output and the historical choices is the main assessment criteria of the CRR results, using precision, recall and MRR seems to be the most appropriate as those metrics are a set-based measures. This approach was used in the distinct majority of the CRR algorithms evaluations and those metrics can be regarded as the domain standard.

One of the drawbacks of this approach is that precision and recall are calculated many times for the one recommendation list, e.g. given the ranked list of 10 recommended reviewers the metrics are calculated for every top-k set. The algorithm assessment would be easier if the recommendation list were evaluated using a single number.

It is also worth noting that the assessment criteria that relies on something different than sets comparison, such as the approaches proposed in the previous section, may require completely new metrics.

5.4 Future work

For achieving better reviewer recommendation accuracy and overall performance, the currently proposed CRR algorithms could be extended with a more inclusive assessment criteria. This would mean new possible optimizations but also new, sometimes previously unforeseen, challenges. One of those challenges could be an effective measurement and evaluation of methods that go beyond a standard way treating historical reviewer assignments as a benchmark for efficacy.

In recent years we witnessed an adoption of code reviewer recommendation tools into mainstream platforms like GitHub (GitHub 2019). This brings more attention to the field, what, combined with rising popularity of machine learning and artificial intelligence, could yield even more innovative approaches to CRR problem.

One thing is certain - the methods of code reviewer recommendation are already a part of a workflow in open source projects, and with the growing number of contributors, their significance is set to be even greater in the future.

References

Balachandran, Vipin (May 2013). "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation". In: pp. 931–940. ISBN: 978-1-4673-3073-2. DOI: 10.1109/ICSE.2013.6606642 (Cited on page 10).

Fejzer, Mikołaj (2017). *reviewers$_r$ecommendationGitHubrespository*. https://github.com/mfejzer/reviewers_recommendation (Cited on page 16).

Fejzer, Mikołaj, Piotr Przymus, and Krzysztof Stencel (2018). "Profile based recommendation of code reviewers". In: *Journal of Intelligent Information Systems* 50, pp. 597–619. DOI: 10.1007/s10844-017-0484-1 (Cited on pages i, 3, 10, 13, 24).

Ferreira, Fabio, Luciana Silva, and Marco Valente (Oct. 2020). "Turnover in Open-Source Projects: The Case of Core Developers". In: DOI: 10.1145/3422392.3422433 (Cited on page 27).

GitHub (2019). *The GitHub Blog - Code review assignment (beta)*. https://github.blog/changelog/2019-11-12-code-review-assignment-beta (Cited on page 30).

— (2020). *The 2020 State of the Octoverse Community Report*. https://octoverse.github.com/static/github-octoverse-2020-community-report.pdf (Cited on page 2).

— (n.d.). *GitHub GraphQL API Reference*. https://docs.github.com/en/graphql/reference (Cited on page 12).

Hahn, Jungpil, Jae Yun Moon, and Chen Zhang (Sept. 2008). "Emergence of New Project Teams from Open Source Software Developer Networks: Impact of Prior Collaboration Ties". In: *Information Systems Research* 19, pp. 369–391. DOI: 10.1287/isre.1080.0192 (Cited on page 27).

Kallis, Rafael, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella (July 2021). "Predicting Issue Types on GitHub". In: DOI: 10.1016/j.scico.2020.102598 (Cited on page 25).

Maciejewski, Sebastian and Jan Techner (2021). *GitHub respository*. https://github.com/S-Maciejewski/code-reviewer-recommendation (Cited on pages 7, 13).

Raymond, Eric S. (Oct. 1999). *The Cathedral and the Bazaar*. ISBN: 978-1-56592-724-7 (Cited on page 2).

Sliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller (July 2005). "When do changes induce fixes?" In: vol. 30. DOI: 10.1145/1082983.1083147 (Cited on page 28).

Thongtanunam, Patanamon (2014). *revfinder GitHub respository*. https://github.com/patanamon/revfinder (Cited on page 17).

Thongtanunam, Patanamon, Chakkrit Tantithamthavorn, Raula Kula, Norihiro Yoshida, Hajimu Iida, and Ken-Ichi Matsumoto (Mar. 2015). "Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. DOI: 10.1109/SANER.2015.7081824 (Cited on pages 2, 10).

Appendix A. Data Extraction Table

Below the reader will find the full version of the Data Extraction Table which summarizes all the papers that during our Systematic Literature Review proved relevant to our study.

ID	Paper ID	Motivation	Assessment criteria (RQ1)	Metrics (RQ2)	Remarks
4	Rebai 2020	Validation of multi-objective approach	Past reviews and collaboration	Precision@k, Recall@k, MRR@k	Recommendations on average better than state of the art
5	Kovalenko 2019	Is CRR relevant ***in vivo?*** Determining relevance with survey	A mix of criteria based on past activity	top-k accuracy, MRR	An overview of the topic; study whether the CRR is helpful for developers
6	Garousi 2020	Extensive research on the relevance of SE research	-	-	Does not answer RQs
8	Jiang 2019	Time decaying relationships	Past reviews; focus on time-decaying of relevance	accuracy, MRR	Describes the problem of relevance diminishing over time
10	Liao 2019	A new CRR algorithm proposition (TIRR), focus on recommendation accuracy - an opportunity to study how a process of developing new algorithm looks like	Past reviewers	Precision, Recall, F-Measure	Strength of connection between reviewers is based on an absolute value (number of commented PR's), thus it can favour 'busy' reviewers over less active ones
12	Sulun 2019	Automatically suggesting appropriate reviewers have two main benefits: (1) reducing overall review time by automatically assigning reviewers (2) increasing the review quality and thus reducing the potential errors related to the reviewed artifact.	comparison with actual reviewers set	top-k accuracy, MRR	implementation uses Software Artifact Traceability Graphs
14	Asthana 2019	we found that reviewer recommendation systems will often assign a large proportion of reviews to a small set of experienced developers (e.g., 20% of the developers in a project are assigned 80% of the reviews)	matching reviewers who the developers chose manually	precision@m, recall@m, F-score@m, PR Hits (P@n), as the fraction of all pull-requests where we were able to make at least one positive suggestion	load balancing recommended reviewers; there is an (often implicit) assumption that those who participated in the review were in fact the best people to review the change and those who were not invited were not appropriate reviewers.
17	Ye 2017	as of June 2018, Github has over 28 million users and 57 million repositories; Gousios conducted a survey with 749 core members and found that they feel overwhelmed in managing pull requests. Integrators are struggling to maintain the quality of the code and are spending a lot of time to review pull requests	compare the ranking result with the ideal ranking in which the candidates who are reviewers should be listed at the top	Accuracy@k, Mean Average Precision, MRR	Many features that the algorithm may be constructed upon.
19	Fejzer 2017	Commits without assigned reviewers take the longest time to be merged due to the lack of both interest and proper domain knowledge; the number of potential reviewers in GitHub is significantly higher than in a traditional code review system like Gerrit.	Past reviewers; time decaying params	top-k prediction accuracy (precision, recall, F-Measure), MRR	low computational complexity due to the incremental model
20	Peng 2018	Marlow found that GitHub reviewers tend to examine contributors' profiles before deciding on whether to accept or reject the PR review request. Despite the in- sightful findings, they did not study the needs from the contributors' perspective and the process did not involve ARR.	-	-	evaluation of mention bot in GitHub (study how software developers perceive and work with the Facebook mention bot, one of the most popular ARR bots in GitHub) - results show that developers appreciate mention bot saving their effort, but are bothered by its unstable setting and unbalanced workload allocation
21	Mohamed 2018	It is important to predict reopened pull requests immediately after pull requests' first close, and help integrators reopen pull requests in time. If pull requests are reopened a long time after their close, they may cause conflicts with new submitted pull requests, add software maintenance cost, and increase burden for already busy developers.	-	-	Study on how to predict if pul request would be reopened - can be used to enhence reviewer recommendation
23	Lipcak 2018	The automated recommendation of code reviewers can reduce the efforts necessary to find the best code reviewers, cutting time spent by code reviewers on understanding large code changes	Comparison with the actual reviewers	top-k accuracy (the percentage of correct recommendations on the total number of recommendations) and MRR	categorization of recommendation algorithms into four groups: i) Heuristic-based approaches, ii) Machine Learning-based, iii) Social Networks-based, iv) Hybrid ap- proaches; In our experiments, we evaluate whether one of the top-k recommended reviewers actually evaluated the pull request. However, we do not know whether this reviewer was the best candidate for the specific task. Historical data might include wrong reviewers choices.
25	Li 2017	each pull request gets 2.6 comments in average and only about 16% of the pull-requests have extended discussions	-	-	Thorough analysis of a code review process in Github; focused mainly on review comments classification
26	Jiang 2017	popular projects receive many pull requests, and commenters may not notice new pull requests in time, or even ignore appropriate pull requests	Comparison with actual commenters; time decaying parameter	top-k prediction accuracy (precision, recall)	Focused mainly on commenter recommendation; The activeness is the most important attribute in the commenter prediction; the mean turnover ratio in GitHub projects is 66% for each quarter
27	Xia 2017	integrators often have difficulty to process the incoming pull requests in time - they can become tired of prioritizing the pull requests due to the heavy workload; 15% of contributors complain the delay of pull request response.	Similarity with the actual reviewer assignment	precision and recall for top-[1,2,3,4,5] recommendations	Vast review of a related work; method focused on social relations
29	Saxena 2017	a method to create a more detailed technology skill profile of a candidate based on her code repository contributions	-	-	SkillMap, a visual representation of candidate skill profile, for quick review and comparison with other candidate profiles
30	Zanjani 2016	it is not always easy to determine who has the most expertise given a particular change for review, especially for newcomers to a codebase or those changing parts of the code with shared ownership by many people	reviewers who contributed in reviews to a given code change and not those reviewers who are assigned to review the code change	precision, recall, F-score for top-[1,2,3,5] recommendation, and MRR	
31	Yu 2016	having access to more potential reviewers does not necessarily mean that it's easier to find the right ones (the "needle in a haystack" problem); three hundred new pull-requests each month	Past reviews; training set consisting of previously prepared set of pull requests (only representative ones are chosen)	top-k prediction accuracy (precision, recall, F-Measure)	comment network making use of social relationships between developers
32	Ouni 2016	Linus's Law "many eyes make all bugs shallow"; Inappropriate reviewers assignment may lead to an inaccurate, time consuming and non effective review process	actual reviewers, that are known, as the ground truth	precision@k, recall@k for top-[1,3,5,10] and MRR	Recent studies showed that when reviewers have a priori knowledge of the context and the code, they complete reviews more quickly and provide more valuable feedback to the author; code review is basically a human process involving personal and social aspects, thus the socio-technical factor plays an extremely improtant role in finding peer reviewers
35	Hannebauer 2016	projects have a large community with a many reviewers, so it can be difficult to decide who should review which patch, especially for newcomers; problems with reviewer assignment in FLOSS projects can defer the acceptance of patches by 6 to 18 days on average; sometimes, submitted patches receive no review at all and therefore are not integrated into the application	Similarity with the actual reviewer assignment	top-k accuracy (the percentage of correct recommendations on the total number of recommendations)	comparison of the prediction performance of eight reviewer recommendation algorithms
36	Rahman 2018	Reliable information on reviewers' expertise (e.g., tech- nology skill) is often not readily available, and it needs to be carefully mined from the codebase; the task of identifying appropriate reviewers is even more challenging and time-consuming for novice developers since they are neither familiar well with the codebase nor are aware of the skills of the hundreds of fellow potential reviewers	evaluatoin using the real code review data from Vendasta codebase.	Top-K Accuracy, Mean Reciprocal Rank, Mean Precision and Mean Recall	solution packaged as a web service and a plug-in for Google Chrome browser

Data Extraction Table pt. 1

ID	Paper ID	Motivation	Assessment criteria (RQ1)	Metrics (RQ2)	Remarks
40	Thongtanunam 2015	The results show that 4%-30% of reviews have code-reviewer assignment problem. These reviews significantly take 12 days longer to approve a code change.	Similiarity with past reviews	Percentage of correct recommendations (at least one correct reviewer (real choice) returned in the top 10 recommendations) and MRR	RevFinder - one of the earliest algorithms. The original test set - Android, LibreOffice, OpenStack and Qt.
41	Xia 2015	74.4 review requests submitted daily to QT's Gerrit from 2011 – 2012	Past reviews	top-k prediction accuracy, MRR	Improved version of the RevFinder proposed by Thongtanunam (combination of file based approach and text mining)
42	Yu 2014	the time of the recommended reviewer submitting his first comment on PR is on average 40.8 hours shorter than those without recommendation; the project managers may not completely find out all potential reviewers from crowds	Comparison with actual reviewers	top-k prediction accuracy (precision and recall)	Comment network approach
44	Balachandran 2013	The reviews would be time consuming or inaccurate if appropriate reviewers are not set. Most of the time, novice developers have to reach out to experienced developers or search the file revision history to assign appropriate reviewers.	Revision history	accuracy defined as the number of correct top-k recommendation (at least one correct reviewer) divided by the number of pull requests	
45	Jeong 2009	On average a review takes 1.5 days and between 39% and 45% of patches are accepted; review requests without initial reviewer assignment take longer and have lower chances to be accepted	Comparison with actual reviewers	Accuracy (top N) (The percentage of predictions with at least one correct reviewer in the top N)	Interesting threats to validity presented in the paper - open source projects are specific, reviewer retirement information is hidden.

Data Extraction Table pt. 2

ID	Authors	Title	Publication Year
4	Rebai, S; Amich, A; Molaei, S; Kessentini, M; Kazman, R	Multi-objective code reviewer recommendations: balancing expertise, availability and collaboratic	2020
5	Kovalenko, V; Tintarev, N; Pasynkov, E; Bird, C; Bacchelli, A	Does Reviewer Recommendation Help Developers?	2020
6	Garousi, V; Borg, M; Oivo, M	Practical relevance of software engineering research: synthesizing the community's voice	2020
8	Jiang, J; Lo, D; Zheng, JT; Xia, X; Yang, Y; Zhang, L	Who should make decision on this pull request? Analyzing time-decaying relationships and file si	2019
10	Liao, ZF; Wu, ZX; Wu, JS; Zhang, Y; Liu, JY; Long, J	TIRR: A Code Reviewer Recommendation Algorithm with Topic Model and Reviewer Influence	2019
12	Sulun, E; Tuzun, E; Dogrusoz, U	Reviewer Recommendation using Software Artifact Traceability Graphs	2019
14	Asthana, S; Kumar, R; Bhagwan, R; Bird, C; Bansal, C; Maddila, C	WhoDo: Automating Reviewer Suggestions at Scale	2019
17	Ye, X	Learning to Rank Reviewers for Pull Requests	2019
19	Fejzer, M; Przymus, P; Stencel, K	Profile based recommendation of code reviewers	2018
20	Peng, ZH; Yoo, J; Xia, M; Kim, S; Ma, XJ	Exploring How Software Developers Work with Mention Bot in GitHub	2018
21	Mohamed, A; Zhang, L; Jiang, J; Ktob, A	Predicting which pull requests will get reopened in GitHub	2018
23	Lipcak, J; Rossi, B	A Large-Scale Study on Source Code Reviewer Recommendation	2018
25	Li, ZX; Yu, Y; Yin, G; Wang, T; Wang, HM	What Are They Talking About? Analyzing Code Reviews in Pull-Based Development Model	2017
26	Jiang, J; Yang, Y; He, JH; Blanc, X; Zhang, L	Who should comment on this pull request? Analyzing attributes for more accurate commenter rec	2017
27	Xia, ZL; Sun, HL; Jiang, J; Wang, X; Liu, XD	A Hybrid Approach to Code Reviewer Recommendation with Collaborative Filtering	2017
29	Saxena, R; Pedanekar, N	I Know What You Coded Last Summer: Mining Candidate Expertise from GitHub Repositories	2017
30	Zanjani, MB; Kagdi, H; Bird, C	Automatically Recommending Peer Reviewers in Modern Code Review	2016
31	Yu, Y; Wang, HM; Yin, G; Wang, T	Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and	2016
32	Ouni, A; Kula, RG; Inoue, K	Search-Based Peer Reviewers Recommendation in Modern Code Review	2016
35	Hannebauer, C; Patalas, M; Stunkel, S; Gruhn, V	Automatically Recommending Code Reviewers Based on Their Expertise: An Empirical Compari	2016
36	Rahman, MM; Roy, CK; Redl, J; Collins, JA	CORRECT: Code Reviewer Recommendation at GitHub for Vendasta Technologies	2016
40	Thongtanunam, P; Tantithamthavorn, C; Kula, RG; Yoshida, N; Iida	Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approa	2015
41	Xia, X; Lo, D; Wang, XY; Yang, XH	Who Should Review This Change? Putting Text and File Location Analyses Together for More Ac	2015
42	Yu, Y; Wang, HM; Yin, G; Ling, CX	Reviewer Recommender of Pull-Requests in GitHub	2014
44	Balachandran, V	Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Ana	2013
45	Jeong, G; Kim, S; Zimmermann, T; Yi, K;	Improving Code Review by Predicting Reviewers and Acceptance of Patches.	2009