Automatically Recommending Code Reviewers Based on Their Expertise: An Empirical Comparison

Christoph Hannebauer* Michael Patalas* Sebastian Stünkel†

Volker Gruhn*

paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Germany
*{first name.last name}@paluno.uni-due.de, †sebastian.stuenkel@stud.uni-due.de

ABSTRACT

Code reviews are an essential part of quality assurance in Free, Libre, and Open Source Software (FLOSS) projects. However, finding a suitable reviewer can be difficult, and delayed or forgotten reviews are the consequence. Automating reviewer selection with suitable algorithms can mitigate this problem. We compare empirically six algorithms based on modification expertise and two algorithms based on review expertise on four major FLOSS projects. Our results indicate that the algorithms based on review expertise yield better recommendations than those based on modification expertise. The algorithm Weighted Review Count (WRC) recommends at least one out of five reviewers correctly in 69 % to 75 % of all cases, which is one of the best results achieved in the comparison.

CCS Concepts

•**Software and its engineering** → **Software defect analysis; Programming teams;** *Open source model;* •**Information systems** → Decision support systems;

Keywords

Code reviewer recommendation, code reviews, expertise metrics, issue tracker, open source, patches, recommendation system

1. INTRODUCTION

A common practice for Quality Assurance (QA) in software development projects are code reviews. In a pre-commit-review policy, every patch is first reviewed by one or more reviewers before it is allowed to be merged into the main codebase stored in a Version Control System (VCS). This is common for Free, Libre, and Open Source Software (FLOSS) projects. FLOSS projects may 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 [51]. Sometimes, submitted patches receive no review at all and therefore are not integrated into the application [38]. Finding reviewers is generally a problem in software development projects [29, 54, p. 182].

There is extensive research on bug triaging, which is the problem of assigning developers to a specified issue and reducing the human effort involved [2,4,8,30,44,45,47]. This is an important problem in practice, and FLOSS projects use tools to support bug triaging [52]. There is also research on the review practices in FLOSS projects in general [6, 11, 28, 40, 53]. More recently, research emerged on tools to support reviewer assignments [5, 25, 50, 51]. These tools rely on algorithms that evaluate the project's review history and calculate the *review expertise* of all potential reviewers for submitted patches as the foundation for the recommendation.

In contrast, Sethanandha et al. propose that the ideal reviewer would be the person with the greatest *modification expertise* for the code that the submitted patch modifies [43]. There is research based on practical observations and empirical data for algorithms to measure the modification expertise of individual developers for a given part of the source code [3, 15, 31, 34, 39, 42]. However, these algorithms have not yet been evaluated for their suitability to recommend reviewers.

Reviewer recommendation tools are only useful if they recommend competent reviewers for submitted patches. This depends on the employed algorithms and on the type of data that these algorithms use for their recommendations. In this paper, we compare the prediction performance of the algorithms File Path Similarity (FPS) [50] and Weighted Review Count (WRC), both of which use review expertise for their decision, against six algorithms based on modification expertise. Our study is the first to evaluate algorithms based on modification expertise for reviewer recommendation and WRC has not been evaluated at all. The evaluation uses historical data from the same three FLOSS projects as in a previous study [50] to ensure internal validity and added Firefox as a fourth FLOSS project for external validity.

2. RELATED WORK

This section describes existing algorithms to measure expertise. A reviewer acquires review expertise when reviewing code. A developer acquires modification expertise when modifying code.

2.1 Review Expertise

Jeong et al. [25] first suggested how reviews of submitted patches could be predicted automatically. They proposed Bayesian Networks to predict the outcome of the review and the actual reviewer of patches.

ASE'16, September 3–7, 2016, Singapore, Singapore
ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970306

Balachandran [5] developed the two reviewer recommendation algorithms Review Bot and RevHistRECO. If a patch for some files is about to be reviewed, Review Bot assigns scores to all previous authors and reviewers of the same lines of code that the patch modifies. RevHistRECO is similar but operates on file history instead of line history. The algorithms use a time prioritization parameter $\delta \in]0;1]$ to decrease the impact of older modifications and patches. The developers with the highest scores are recommended as reviewers. Balachandran compares the two algorithms on two projects with 7035 and 1676 reviews. Review Bot had a better prediction performance than RevHistRECO. Both Review Bot as well as RevHistRECO use both review expertise as well as modification expertise.

Another approach was presented by Thongtanunam et al. [50]. They described the recommendation algorithm **File Path Similarity (FPS)**. The main idea of FPS that files with similar file paths are closely related by their function. It assumes that in most large projects the directory structure is well organized. The algorithm assigns scores to reviewers of files that have similar file paths as the files of a given review request. Reviewers with the highest accumulated score are recommended as reviewers for the given request. Like Review Bot, the algorithm contains a time prioritization factor $\delta \in]0;1]$ to prefer more current over past reviews.

In the following, we define FPS in mathematical terms according to the original definition in Thongtanunam et al.'s paper. Let F, R, and D denote the set of all files, all reviews, and all developers, respectively. Let $\text{Files} : R \to \mathcal{P}(F)$ be the function that maps a review to the set of its reviewed files. FPS uses a function $\text{CommonPath} : F \times F \to \mathbb{N}$ that counts the number of common directory levels from left to right in the paths of two given files. If the two files are identical, the count is one higher than for two files in the same directory because of the identical file name. Similarly, $\text{Length} : F \to \mathbb{N}$ counts the number of directory levels including the file name in a file's path, or in a different formulation $\text{Length}(f) = \text{CommonPath}(f, f)$. The definition of the functions $\text{Similarity} : F \times F \to \mathbb{N}$ and $\text{ReviewSimilarity} : R \times R \to \mathbb{R}$ shall be the following:

$$\text{Similarity} : f_1, f_2 \mapsto \frac{\text{CommonPath}(f_1, f_2)}{\max\{\text{Length}(f_1), \text{Length}(f_2)\}}$$

$$\text{ReviewSimilarity} : r, s \mapsto \frac{\sum_{\substack{f_1 \in \text{Files}(r) \\ f_2 \in \text{Files}(s)}} \text{Similarity}(f_1, f_2)}{|\text{Files}(r)| \cdot |\text{Files}(s)|}$$

The function Reviewers shall map each review to the set of developers that participated in the review. Let $r_1, \ldots, r_n$ denote all reviews in R in chronological order. For a developer d and a review r_x, let ${}_dR_x \coloneqq \{r_i \in R | 0 < i < x \wedge d \in \text{Reviewers}(r_i)\}$, the set of all reviews that happened before r_x in which the developer d participated as a reviewer. Then $\text{FPSScore}_\delta : R \times D \to \mathbb{R}$ gives a prediction of each developer's expertise for each review according to the following calculation:

$$\text{FPSScore}_\delta : r_x, d \mapsto \sum_{r_i \in {}_dR_x} \text{ReviewSimilarity}(r_x, r_i) \cdot \delta^{x-i-1}$$

The time prioritization factor $\delta \in]0;1]$ ensures that older reviews have less influence on the resulting score than newer reviews. For $\delta = 1$, there is no time prioritization and older reviews have the same weight as newer reviews.

Thongtanunam et al. [50] compared the prediction performance of FPS to Balachandran's Review Bot. Both algorithms calculated recommendations for historical reviews of the three FLOSS projects Android Open Source Project (AOSP), OpenStack, and Qt. The algorithms used five different time prioritizations $\delta = 1$, $\delta = 0.8$, $\delta = 0.6$, $\delta = 0.4$, and $\delta = 0.2$. The number of recommended reviewers was limited to $k = 1$, 3, and 5 for each review. Then they compared each set of recommended reviewers with the actual historical reviewers in each review. A recommendation was considered correct if at least one of the k recommended reviewers was one of the actual reviewers.

The results answer two research questions: First, for all considered combinations of projects and values of k and δ, FPS performed better for greater values of δ. Consequently, $\delta = 1$ outperformed all other tested values of δ. For Review Bot, δ had little influence (less than 0.5 percentage points) with the exception of AOSP, where $\delta = 0.2$ and $k = 1$ is an extreme outlier. Since this outlier is not discussed in the text, and even contradicted with, even this outlier is likely just a misprint in the table. Second, FPS significantly outperforms Review Bot, except for some configurations in Qt especially for low values of δ and number of recommended reviewers. For $k = 5$ and $\delta = 1$, FPS predicts 77.1 %, 78.0 %, and 36.9 % correctly for AOSP, OpenStack, and Qt, respectively. For $k = 5$, Review Bot predicts, with a project-specific optimal value for δ, 29.3 %, 38.9 %, and 27.2 % correctly for AOSP, OpenStack, and Qt, respectively.

In a more recent study, Thongtanunam et al. [51] further improved their prediction algorithm and added the FLOSS project LibreOffice to their evaluation. They used a more sophisticated version of the CommonPath function. This more sophisticated version does not only compare the longest common prefix of two paths, but also three other types of path similarity metrics and combines them with Borda count [7]. The improved algorithm correctly predicts 79 %, 77 %, 41 %, and 59 % with $k = 5$ for AOSP, OpenStack, Qt, and LibreOffice, respectively. This is a change of +2, -1, and +4 percentage points.

The software forge service GitHub [19] offers features to support "social coding" [12]. Jiang et al. [26] analyzed reviewer recommendation algorithms that use social relationships between developers and reviewers among other properties for recommendations. The best algorithm in their evaluation uses support vector machines (SVMs) as machine learning algorithm for recommendations. The results cannot be directly compared to the other approaches, including ours, because of their specialization on GitHub features that are not available in other FLOSS projects.

Zanjani et al. [56] combine three metrics to calculate review expertise. For each combination of reviewer and file, each metric yields a value between 0 and 1. These three values are added to a composed review expertise.

Yu et al. [55] presented an approach based on information retrieval. Their algorithm consists of two parts. The first part tries to find similar reviews using textual semantics. The expertise score of a developer is based on the number of comments in this reviews. The second part uses comment relations to calculate the quantity of common interests between developers. Both values are added together to predict relevant reviewers.

The algorithm **Weighted Review Count (WRC)** is also part of the comparison and uses review expertise. It is defined and discussed in Section 3.3.

2.2 Modification Expertise

This section discusses research on metrics to evaluate the expertise a software developer has about a software artifact. The existing metrics differ on one hand in the source of the measured data and on the other hand in the algorithms used to calculate the resulting expertise. The metrics were not originally designed for reviewer

recommendation, instead common goals of these expertise metrics are

- finding a mentor for new team members, and
- finding an expert for specific questions on a part of the application.

Each unique algorithm we will evaluate for reviewer recommendation in this study is highlighted in boldface.

The most simple expertise calculation algorithm is the **Line 10 Rule**: Only the last editor of a module is assumed to have expertise with it. The name of this algorithm derives from a VCS that stored the author of a commit in line 10 of the commit message. Because of its simplicity, various research uses the Line 10 Rule. [31, 42]

In early research on expertise in the software development domain, McDonald and Ackerman [31] used three types of sources to identify experts, who may be software developers or support representatives: First, they asked all users to create profiles about themselves. Second, they counted the number of changes each developer made on a module using VCS logs. Third, they indexed the descriptions of issues for which support representatives were responsible using data from an issue tracker. This allowed their tool **Expertise Recommender** to recommend a set of experts, given a problem description. The recommendation list is ordered by the date of the last change to the module, and the last editor appears first in the list. This is a straightforward extension to the Line 10 Rule, which considers only the authors of the latest change. Expertise Recommender also takes authors of previous changes into account, although they are considered to have a lower expertise.

However, user profiles are often outdated [34]. The **Number of Changes** to a module is only a very rough metric of users' modification expertise. For example, if the development team switches to the Continuous Integration (CI) development method, the number of logged changes to the modules will increase due to the regular commits to the VCS [14]. This introduces a bias to the metric, as early, seldom committing developers are considered less experienced than later developers with a high commit frequency.

Nevertheless, other research also uses the number of changes to an artifact as a metric to quantify the developers' expertises of the artifact. Balachandran's Review Bot [5] described in Section 2.1 also falls into this category. As other early research of this type, Mockus and Herbsleb [34] define Experience Atoms (EAs) to be "elementary units of experience". They distinguish between several categories of experience, where expertise with a specific module is only one example category. To calculate the EA a developer has acquired, they mine VCS logs. Developers acquire one EA for each commit in each relevant category. Examples of categories are the module that the commit changes and the technology that the commit makes use of. They have also developed a tool called Expertise Browser to visualize the EAs. Expertise Browser allows its user to find experts, for example the developers with the highest number of EAs for a module.

As another example, Robles et al. [41] devised a method to visualize changes to the core group of developers in a FLOSS project at each point in time. Their calculation also counts the number of commits for each developer to the VCS. For each time period, a fraction of developers with the most commits are considered core developers. Qualitative analyses have shown 10 % to 20 % to be reasonable fractions of commits to be considered core developers. Thus, the calculation's output is a project-wide decision whether a specific developer was a core developer at a specific point in time. The visualization techniques itself are not relevant for our study and will not be discussed here.

Minto's and Murphy's Emergent Expertise Locator [33] also calculates the number of times each developer has edited each file and stores this information in a File Authorship Matrix. This matrix is used as part of a calculation to find developers that share expertise and therefore are assumed to be part of an emerging team. The Emergent Expertise Locator can then propose prospective colleagues to a developer to help in the formation of such a team. However, the calculation step used to find a developer's expertise with a file still only counts the number of edits.

Schuler and Zimmermann [42] propose to count not only the Number of Changes directly to a module as modification expertise, but also the number of commits with code that call the module. This is no modification expertise, but usage expertise. However, they provide no algorithm that aggregates the measured modification and usage expertises into a single metric. As modification expertise appears more adequate than usage expertise to recommend reviewers for *modifications* of a module, their approach does not directly contribute an additional algorithm for reviewer recommendation.

Gîrba et al. [20] approximated **Code Ownership** using VCS logs, which is specifically a list of file revisions with an author name and the number of removed and added lines for each file. They developed two algorithms that execute consecutively for the Code Ownership approximation. The first algorithm approximates the size of each code file, as the file size is not directly available in their type of VCS log. For each author, the second algorithm approximates the fraction of lines in the file that this author "owns", i.e. whom the author has edited last. Ownership Maps visualize these ownership data. They also showed how to identify patterns in the Ownership Map.

Alonso et al. [1] assigned each source code file to a category based on its file path. Their tool **Expertise Cloud** measures a developer's expertise in a category as the number of changes to files in the category. Thus, this algorithm is a variant of counting the number of changes on a module instead of the file level. It stands out in that it also specifies how to find out the changed modules from VCS logs. However, their example project, the Apache HTTP Server Project [49], clearly documented its source code layout which implies a classification scheme for the project's files. It is not immediately clear how to classify files in other projects, which have less clear documentation on their source code layout, let alone an automatic classification without manual configuration. Their approach also includes a visualization of expertise in tag clouds, hence the name Expertise Cloud.

The Degree-of-Knowledge (DOK) defined by Fritz et al. [15, 16] aggregates two metrics on different data sources, the Degree-of-Interest (DOI) and the Degree-of-Authorship (DOA). First, the DOI measures how much a developer *uses* a module. The DOI extends Schuler's and Zimmermann's idea of usage expertise [42], as it accounts not only calls to the module, but also other interactions such as opening a module's source code in the Integrated Development Environment (IDE) [27]. Second, the DOA measures how much of the module's source code a developer has *created*. Similarly to Gîrba et al.'s Code Ownership approximation [20], a developer's DOA decreases when other developers change the module. Fritz et al. determined concrete weightings for aggregating the DOA components and the DOI into a single DOK value via linear regression on experiment data. In multiple experiments, they found that the weightings depend on the project, but they also gave general weightings that best fit the aggregated experiment data. Acquiring the data to calculate the DOI requires a plugin on each developer's computer. This impedes its usage in our evaluation in three ways. First, it disallows using data before the plugin was installed. Second, additional effort is necessary to let the developers in the evaluated FLOSS projects

use the plugin, as the plugin needs support and FLOSS developers need to be convinced to cooperate. Third, ensuring the plugin's usage requires a closed experimental setting. Therefore, we used only the **DOA** for the comparison.

Anvik and Murphy [3] compared three different algorithms to determine experts. The algorithms require that the issues have already been closed, and are therefore not suitable for reviewer prediction. However, two of the algorithms evaluate VCS logs to find experts, and one uses data from the issue tracker. In this regard, the comparison resembles the comparison in this paper, as it also compares recommendation systems based on modification expertise with recommendation systems based on expertise acquired when using an issue tracker.

Anvik and Murphy found the two algorithms based on VCS logs to have an average precision of 59 % and 39 %, and an average recall of 71 % and 91 %. The algorithm based on bug reports shows results between the two VCS based algorithms, with an average precision of 56 % and an average recall of 79 %. Thus, no algorithm is better for all use cases, instead there is a trade-off between precision and recall. Furthermore, the number of recommended experts differs between algorithms and their published data show a strong association between the average number of recommended experts and average recall. This suggests that the choice between the two sources of data may have little or no influence at all on the measurement of expertise.

3. EXPERTISE EXPLORER

Each expertise algorithm presented in Section 2.2 works on a dataset from a development tool, most commonly the VCS logs, and calculates the expertise for a given code artifact for a given developer. Some expertise algorithms like EA and DOI/DOA distinguish between different types of expertise, but even these algorithms have an explicit variant to calculate modification expertise.

This modification expertise may or may not be different from the review expertise that the algorithms presented in Section 2.1 approximate. If they are different, these two types of expertise are not distinct: Review expertise implies an understanding of the changes to an artifact, which is helpful or even necessary to create code [48]. Thus, when gathering review expertise, a developer also gathers modification expertise. The reverse is also true: An expert creator of an artifact's code may anticipate side effects of a change and, more generally, knows the context of a code artifact, like its intended purpose, its callers and callees, and its technical constraints. These skills obviously also help when reviewing code.

This study uses the platform Expertise Explorer that we have developed for expertise calculations and the more specific purposes of this study. Expertise Explorer is publicly available [46] under the terms of the GNU General Public License (GPL) and therefore anybody can reproduce our results or use it for similar calculations or comparisons. We also provide a lab package with the source and result data used in this study [23].

This section provides exact definitions for ambiguous terms like review and reviewer. A description of the technical aspects of Expertise Explorer relevant for our evaluation of the historical data follows. The section eventually provides an alternative calculation method for FPS that fits to Expertise Explorer's data structure.

3.1 Data Model for the Review Process

The evaluation prerequires an exact definition of what a review is and when the algorithms must calculate their predictions of reviewers. If the calculation takes place later, then the algorithms may obviously use more data and may come to different predictions. The review model includes the concepts of *issues*, *patch submissions*, *reviews*, and the main *VCS*. Figure 1 depicts the former three and their relationships as an Entity Relationship Diagram (ERD) [9]. An issue is a change request to the FLOSS project's application, either caused by a fault in the software or in case of a feature request by a change to the specification. In either case, only changes to the source code implement this change request. For our definition, documentation files and build scripts also count as source code if they also reside in the VCS, because they can be difficult to distinguish and the review process is the same. Thus, any issue in the FLOSS project's issue tracker counts as issue. Proposed changes to the source code manifest as patch submissions. Each patch submission has an author, occurs at a specific point of time, and modifies one or more files in the VCS. A patch submission belongs to one specific issue and each issue may have any number of patch submissions. Each patch submission may receive any number of reviews. Each review is performed by a reviewer at a specific point of time and belongs to one specific patch submission. The files that a patch submission modifies also belong to its corresponding reviews. Obviously, a review can occur only after its corresponding patch submission. Note that although a review is actually positive or negative, this is not relevant for the model. What exactly counts as a review depends on the issue tracker and will be discussed separately for each FLOSS project in our evaluation. Eventually, some patch submissions are merged into the main VCS at a specific point in time.

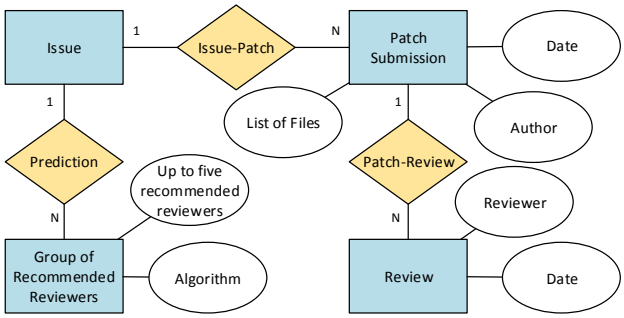

Figure 1: ERD of the core data structures used in the evaluation

Issues without patch submissions or reviews are discarded for the evaluation. For the remaining issues, all algorithms calculate up to five *recommended reviewers* for the issue when the earliest patch is submitted. If the algorithms were used in a recommendation tool in practice, this is when the FLOSS project has to decide who should review the patch. For later patch submissions, the algorithms will not calculate new recommendations: One reviewer often reviews all patch submissions of an issue and therefore a reviewer recommendation tool does not need to recommend new reviewers in these cases. Thus, there is exactly one group of up to five recommended reviewers for each algorithm and for each of the remaining issues. All reviewers who review any patch submission of an issue at any point in time are considered *actual reviewers* for the issue.

An algorithm may use data from all events that have occurred before the time of calculation, i.e. when the earliest patch to an issue is submitted. For WRC and FPS, this includes all reviews on patch submissions in other issues. Reviews in the issue under analysis cannot have occurred before the first patch submission and therefore are not included. There can be multiple reviews for each issue, so the periods between first and last review of two issues may overlap. In these cases, WRC and FPS take early reviews of an issue into consideration while they do not take later reviews of the same issue into consideration, because these later reviews happen only after the patch submission that triggered the calculation.

The six algorithms described in Section 2.2 use data from the main VCS. That means that they can use patch submissions only if and only after they have been merged into the main VCS. In contrast to WRC and FPS, they take the author of a patch submission into account.

3.2 Data Processing

Expertise Explorer needs two sets of data as input. These sets of data may or may not have the same origin, depending on the project under analysis. First, Expertise Explorer needs authorship data as stored in a VCS log. With these data, Expertise Explorer can execute the VCS-based reviewer recommendation algorithms described in Section 2.2. Second, Expertise Explorer needs the review history. With these data, Expertise Explorer can execute the issue-tracker-based reviewer recommendation algorithms. The review history is also needed to compare the computed reviewer recommendations with actual reviewer recommendations, which will be used for the evaluation. The VCS history data may be substituted by the issue tracker history data if the latter contain enough authorship information, especially the number of added and removed lines and whether files are modified or newly created.

A database stores for each project authorship data, expertise values for each combination of file and contributor, and the algorithm comparison results described in Section 3.1. In a first import step, all authorship data are imported into the database. Next, Expertise Explorer iterates chronologically through a list of activities in the project's issue tracker, specifically reviews and patch submissions.

For each first patch submission, Expertise Explorer calculates all algorithms' expertise values for all combinations of contributors and submitted files using review and authorship data that occurred before the data of this first patch submission. Thus, the expertise values correspond to the values that a reviewer recommendation tool would have had access to at the date and time of the first patch submission to the issue. Afterwards, Expertise Explorer calculates for each algorithm the five contributors with the highest expertise for the patch. If the patch contains multiple files, this involves an algorithm-specific aggregation of expertise values. For most algorithms, expertise values are simply added. Note that adding and the arithmetic mean yield the same reviewer recommendations, as the order of recommended reviewers is the same for both types of aggregations. For Line 10 Rule and Expertise Recommender, the maximum expertise value is used instead of the sum, as this yields the latest editor of any of the reviewed files. For each review, the algorithms based on review expertise update their data to include the review. Furthermore, the reviewer is stored as an actual reviewer for the issue.

Afterwards, another module of Expertise Explorer iterates over the list of issues in the database. For each issue, it compares the up to five recommended reviewers for each algorithm with the set of actual reviewers for the issue to find out which algorithms had correctly recommended reviewers and which had not.

3.3 Alternative Algorithm for FPS

Thongtanunam et al. [50] provided an algorithm in pseudo-code for FPS which directly uses the formulas presented in Section 2.1. Because of Expertise Explorer's database structure, Expertise Explorer cannot directly use this algorithm. However, we will show in this section that there is an equivalent definition for $FPSScore_\delta$ that Expertise Explorer can use.

Using the definitions of Section 2.1, we define a function $\mathrm{WRC}_\delta : F \times D \times R \to \mathbb{R}$ that calculates the value **Weighted Review Count (WRC)**, an intermediate value that specifies the review experience a developer has with a specific file at the time of a specific review. Remembering that $r_1, \ldots, r_n$ are all reviews in chronological order, WRC_δ has the following definition:

$$\mathrm{WRC}_\delta : f, d, r_x \mapsto \sum_{r_i \in_d R_x, f \in \mathrm{Files}(r_i)} \delta^{x-i-1} \left|\mathrm{Files}\left(r_i\right)\right|^{-1}$$

Expertise Explorer treats WRC like any other algorithm and calculates its value for all developers and all files. Using the pre-calculated values for WRC allows a different calculation of FPS using the following equivalence:

$$\begin{aligned}
&\mathrm{FPSScore}_\delta\left(r_x, d\right) \\
&= \sum_{r_i \in_d R_x} \mathrm{ReviewSimilarity}\left(r_x, r_i\right) \cdot \delta^{x-i-1} \\
&= \sum_{r_i \in_d R_x} \frac{\sum_{f_1 \in \mathrm{Files}(r_x)} \sum_{f_2 \in \mathrm{Files}(r_i)} \mathrm{Similarity}\left(f_1, f_2\right)}{\left|\mathrm{Files}\left(r_x\right)\right| \cdot \left|\mathrm{Files}\left(r_i\right)\right|} \cdot \delta^{x-i-1} \\
&= \sum_{f_1 \in \mathrm{Files}(r_x)} \left|\mathrm{Files}\left(r_x\right)\right|^{-1} \sum_{\substack{r_i \in_d R_x \\ f_2 \in \mathrm{Files}(r_i)}} \frac{\mathrm{Similarity}\left(f_1, f_2\right)}{\left|\mathrm{Files}\left(r_i\right)\right|} \cdot \delta^{x-i-1} \\
&= \sum_{f_1 \in \mathrm{Files}(r_x)} \left|\mathrm{Files}\left(r_x\right)\right|^{-1} \sum_{\substack{f_2 \in F \\ r_i \in_d R_x, f_2 \in \mathrm{Files}(r_i)}} \frac{\mathrm{Similarity}\left(f_1, f_2\right)}{\left|\mathrm{Files}\left(r_i\right)\right|} \cdot \delta^{x-i-1} \\
&= \sum_{\substack{f_1 \in \mathrm{Files}(r_x) \\ f_2 \in F}} \frac{\mathrm{Similarity}\left(f_1, f_2\right)}{\left|\mathrm{Files}\left(r_x\right)\right|} \sum_{r_i \in_d R_x, f_2 \in \mathrm{Files}(r_i)} \left|\mathrm{Files}\left(r_i\right)\right|^{-1} \delta^{x-i-1} \\
&= \sum_{\substack{f_1 \in \mathrm{Files}(r_x) \\ f_2 \in F}} \frac{\mathrm{Similarity}\left(f_1, f_2\right) \mathrm{WRC}_\delta\left(f_2, d, r_x\right)}{\left|\mathrm{Files}\left(r_x\right)\right|}
\end{aligned}$$

This alternative algorithm to calculate FPS has the advantage over the original variant that the values for WRC can be calculated before knowing which files belong to r_x. Thus, Expertise Explorer pre-calculates all current values for WRC. When the FLOSS project needs a reviewer recommendation for a set of files $\mathrm{Files}\left(r_x\right)$, the alternative algorithm can provide the recommendation more quickly.

4. EMPIRICAL EVALUATION

We evaluated the prediction performance of the six algorithms based on modification expertise described in Section 2.2, in particular Line 10 Rule, Number of Changes, Expertise Recommender, Code Ownership, Expertise Cloud, and DOA. We compared these algorithms to FPS developed by Thongtanunam et al. [50] described in Section 2.1. We used the original definition of CommonPath and not the more sophisticated version developed in their more recent publication [51] because calculating FPS values for all our data takes multiple weeks even if there are no problems and it came to our attention only recently. Furthermore, the more sophisticated version's results differ from the original version by only about +2, -1, and +4 percentage points for the three FLOSS projects AOSP, OpenStack, and Qt, which is less than the differences we expected between the other algorithms. Furthermore, we used only $\delta = 1$ as parameter, as this was the optimum tested value for FPS as determined by Thongtanunam et al. [50]. In their more recent study, Thongtanunam et al. [51] also used only $\delta = 1$. We also included WRC as defined in Section 3.3 in the comparison, because Expertise Explorer had to calculate its values for technical reasons anyway. We did not include Review Bot [5] in the comparison, as it had already been shown to be inferior to FPS [50, 51] and, differently to all other considered algorithms, Review Bot requires much more input data, specifically a line-based modification history.

We evaluated the algorithms on data from four different projects: Firefox, AOSP, OpenStack, and Qt. Thongtanunam et al. [50] also used AOSP, OpenStack, and Qt, which allows us to compare results. We added Firefox to increase external validity, as it uses a different issue tracker than the other three FLOSS projects.

Firefox uses Bugzilla to keep track of reported software bugs, AOSP, OpenStack, and Qt use Gerrit Code Review. Table 1 summarizes the characteristics of our study data. The study period includes a training phase of one year. Only prediction results after the training phase were part of the further analysis. This ensured that the algorithms could access historical data of at least one year for each recommendation. The numbers in parentheses refer to the properties excluding the data from the training phase.

4.1 Firefox

Mozilla is the parent project of Firefox and hosts the infrastructure for Firefox and its other subprojects. Part of this infrastructure is Mozilla's issue tracker Bugzilla, which happens to be a Mozilla project itself. Mozilla tracks the issues of all Mozilla projects in one Bugzilla instance [36], which will be referred to as just Bugzilla in the following. Mozilla employs a policy that requires an issue in Bugzilla for every change to the code before it is committed to the Mozilla codebase. Furthermore, every change needs positive review before it is committed, even if the author is the module's maintainer or someone else with review rights. In these latter cases, author and reviewer need to be different. The author of the change attaches a patch in diff [24] format to the issue and requests the review. Reviewers indicate the result of their review with a flag on the reviewed attachments. Usually, one review from an eligible reviewer suffices before committing, although there are exceptions. [35]

The VCS log of Mozilla's Mercurial (hg) repository for Firefox [37] is one of two data sources used for the comparison. They contain all information the VCS-based recommendation algorithms need to compute expertise values. In particular, hg logs not only the committer of a patch, but also its author. Expertise Explorer reads hg logs to find this information. The VCS log used for the analysis contains all changes committed between 2007-03-22 and 2013-03-07. On 2007-03-22, Firefox migrated from the VCS Concurrent Versions System (CVS) to hg, therefore no older VCS logs were used.

The second data source of the comparison is the history data of Mozilla's issue tracker Bugzilla [36]. One component for the download and analysis of Bugzilla's history data of the Firefox project was Zhou's and Mockus's download and data transformation scripts [57] for Mozilla. The applicable output of the scripts is a Comma Separated Values (CSV) file with one "activity" in Bugzilla per line. Reviews are one of those activities. The CSV file contains the following information for each review: Who was the reviewer? When did the review take place? The CSV file does not say which files were reviewed, but only an identification number for an issue's attachment containing the patch as diff file. To retrieve this missing information, Expertise Explorer has a component AttachmentCrawler that downloads the diff files of each reviewed patch to find out which files the patch changes. Expertise Explorer parses the CSV file and filters for relevant reviews such that each included activity fulfills the following constraints:

- Each relevant activity contains one of the flags "review+" or "review-", as these indicate completed reviews [6]. The analysis discards architecture reviews indicated by "superreview", as these are not in the focus of this work.
- The review took place between 2007-03-22 and 2013-03-07, as the VCS logs span this time frame.
- Reviews must be for files of the Firefox project. All Mozilla projects use Bugzilla and are therefore present in the CSV file. The reviewed file names must match a file in the VCS log to fulfill this condition.

In some cases, it is still ambiguous to which Mozilla subproject an issue belongs and it is up to subjective decisions. A clear distinction between Firefox issues and issues of other Mozilla projects is therefore not always possible.

4.2 AOSP, OpenStack, and Qt

For the Gerrit-based projects AOSP, OpenStack, and Qt, we used the data obtained from Hamasaki et al. [21], analogously to Thongtanunam et al. [50]. However, we used the newest available data, which are newer than Thongtanunam et al.'s. The newer data cover the period up to the end of 2014 instead of only to the beginning and middle of 2012, so we considered 14 times as many reviews in total.

The Gerrit data contained some inconsistencies that we had to take care of. First, the format of the data is different to the description [21]. In the latest dataset, there are unfortunately no flags which indicate who performed the review and when. However, these flags are essential for our studies, because these data are used both as the basis for WRC and FPS as well as for the evaluation for all algorithms. However, it is possible to extract the reviews from auto-generated message texts. Messages containing one of the following seven text patterns designate reviews: "Code-Review+", "Code-Review-", "Looks good to me", "Approved", "I would prefer that you didn't [...]", "Do not merge", and "Do not submit".

A further problem concerns the Qt data. From 2014-06 onwards, only the last patch submission of an issue is stored in the data. Since patch submissions contain the list of modified files, we could not identify on which files a review was made, except for reviews of the last patch. As this problem affects only the two algorithms WRC and FPS that are based on review expertise, the data are biased in favor of the other six algorithms based on modification expertise. Thus, we have limited the observation period for Qt to issues occurring before 2014-06.

Expertise Explorer needs two sets of data, one for authorship and one for review history data. Both datasets can be generated from Hamasaki et al.'s JavaScript Object Notation (JSON) formatted files [21]. The first set of data could have been obtained from the project's VCS, but we chose to generate it from Hamasaki et al.'s JSON formatted files. One advantage is that these files use consistent names for authors and reviewers.

We wrote a Python script that expects Hamasaki et al.'s JSON file as input and creates an authorship data file. The script iterates over all JSON elements, wherein an element represents an issue, and for every issue over all messages. Only issues with at least one merge message and one file are considered in the further processing. The script recognizes merge messages based on the following text patterns: "successfully merged into [...]", "successfully cherry [...]", and "successfully pushed". We interpret the merge message date as the date that the referenced patch has been merged into the VCS.

To generate the review history data, we wrote another Python script. Like the first script, it iterates over all messages of every issue. It uses the above-mentioned text patterns to identify a review message. Similar to the first script, we use the date of a review message as the review date, the author of the message as the reviewer and the files of the referenced patch as the reviewed files. As a result, the script generates one line in a CSV for every review. Additionally, the script creates one result line for every patch submission.

Table 1: Characteristics of study data, the values in parentheses exclude training data

	Firefox	AOSP	OpenStack	Qt
Study Period including Training Year	2007-03–2013-03	2008-10–2014-12	2011-07–2014-12	2011-05–2014-06
Number of Contributors	1762	2489	3717	1143
Number of Reviewers	781 (712)	1001 (997)	2813 (2803)	268 (244)
Number of Issues	60 329 (52 892)	39 090 (37 875)	119 287 (117 145)	70 358 (48 709)
Number of Reviews	73 355 (64 416)	55 292 (53 886)	376 494 (371 570)	75 907 (52 570)
Number of Files in VCS	123 939	704 112	103 486	215 890
Arithmetic Mean of Reviews per Issue	1.22	1.41	3.16	1.08

5. RESULTS

Analogously to the previous research on reviewer recommendation [5,25,50,51], accuracy is the percentage of correct recommendations on the total number of recommendations. A recommendation is considered correct if the intersection between the set of recommended reviewers and the set of actual reviewers on the same issue is not empty. Again analogously to previous research, we did not use the concepts of recall and precision, because we let all algorithms recommend five reviewers if possible and so there is no trade-off between recall and precision.

Figure 2 shows the aggregated top-5 accuracy of the eight recommendation algorithms in the four evaluated FLOSS projects over the course of the study period excluding the training phase. Table 2 shows all top-1, top-3, and top-5 prediction performances at the end of the study period.

5.1 Discussion

The main research goal of this study was an empirical comparison of reviewer recommendation algorithms. Two of the compared algorithms used review expertise for their recommendations and six algorithms used modification expertise. For every evaluated FLOSS project, the algorithm with the highest accuracy was an algorithm based on review expertise. Surprisingly, in two out of four FLOSS projects, the algorithm WRC outperformed FPS. Furthermore, WRC performs close to the optimum even for AOSP and OpenStack, where it is not the best algorithm. Hence, WRC is a reliable algorithm. In contrast, FPS seems to be more sensitive to the characteristics of the FLOSS project, as its top-5 accuracy is only 42 % for Qt.

In addition to its good prediction performance, WRC is a simple algorithm, especially compared to other algorithms with good prediction performance like FPS and Expertise Cloud: It requires less input data per recommendation, it is easier to implement, and its value can be computed more quickly. The definition in Section 3.3 immediately shows that WRC is simpler than FPS as the value of WRC is used in the calculation of FPS. Like FPS, Expertise Cloud evaluates expertise not only directly on the reviewed files, but also on files with similar paths. As a consequence, calculating a value for Expertise Cloud or FPS relies on data for a large proportion of files in the FLOSS project. In large FLOSS projects like the ones evaluated in this study, this can involve querying hundreds of thousands of values. FPS and Expertise Cloud therefore have a worse computational performance than the other algorithms including WRC. During the evaluation in this study, FPS and Expertise Cloud needed at least one order of magnitude more time for the computation than the other algorithms.

5.1.1 Reviewer Base Set

As described above, algorithms based on review expertise usually perform better in terms of accuracy than those based on modification expertise. However, it has to be taken into account that the former algorithms have an implicit advantage over the latter: they automatically exclude developers without review rights. As shown in Table 1, the number of contributors in a FLOSS project exceeds the number of reviewers by a factor of 1.3 to 4.3. By design, the algorithms based on review expertise can only recommend reviewers who had already performed a review in the past and therefore have a higher chance to currently have review rights. Algorithms based on modification expertise may therefore recommend developers who never perform reviews, as they lack review rights. If the algorithms would filter out reviewers without review rights before recommendation, this might increase accuracy especially for algorithms based on modification expertise and thereby turn the tide.

5.1.2 The Most Competent Reviewer

Although it may seem paradoxical first, it may actually be undesirable in a FLOSS project that each patch receives its review from the most competent reviewer. This may be the case if some members of the FLOSS project are the most competent reviewers for so many patches that they cannot review all of them. It might be better if less competent but also less busy reviewers perform some of the reviews, as the disadvantages of long delays may at some point be greater than the disadvantages of slightly worse feedback in the reviews. As another reason, some very competent reviewers might be even more competent developers. If this is the case in a FLOSS project, good developers might have higher value than good reviewers. These two reasons appear more likely when considering that a small group of about 3.1 % to 8.9 % of all contributors are the core developers of a FLOSS project who contribute 80 % of the code [18]. Whether those trade-offs between reviewer competency and reviewer cost are necessary depends on the specific FLOSS project. Algorithms based on review expertise may have an advantage in this case, as their recommendations do not directly depend on competency but on who reviewed patches in the past – which already reflects current practice in the FLOSS project and the trade-offs that their members decided about.

5.1.3 Vicious Circles

If a FLOSS project extensively uses a reviewer recommendation tool, algorithms based on review expertise base their decisions on data that they themselves influenced – they recommend reviewers, these reviewers review the patch, and then they assume that these reviewers are well suited to review those types of patches because they did it in the past. This induces the danger of a vicious circle, where some unsuitable reviewers are recommended more and more often for some kind of patches because of prior unsuitable recommendations. This problem has also been discussed for other types of recommender systems [10]. Recommendation algorithms based on modification expertise do not suffer from this problem by design, as they do not take review expertise into account.

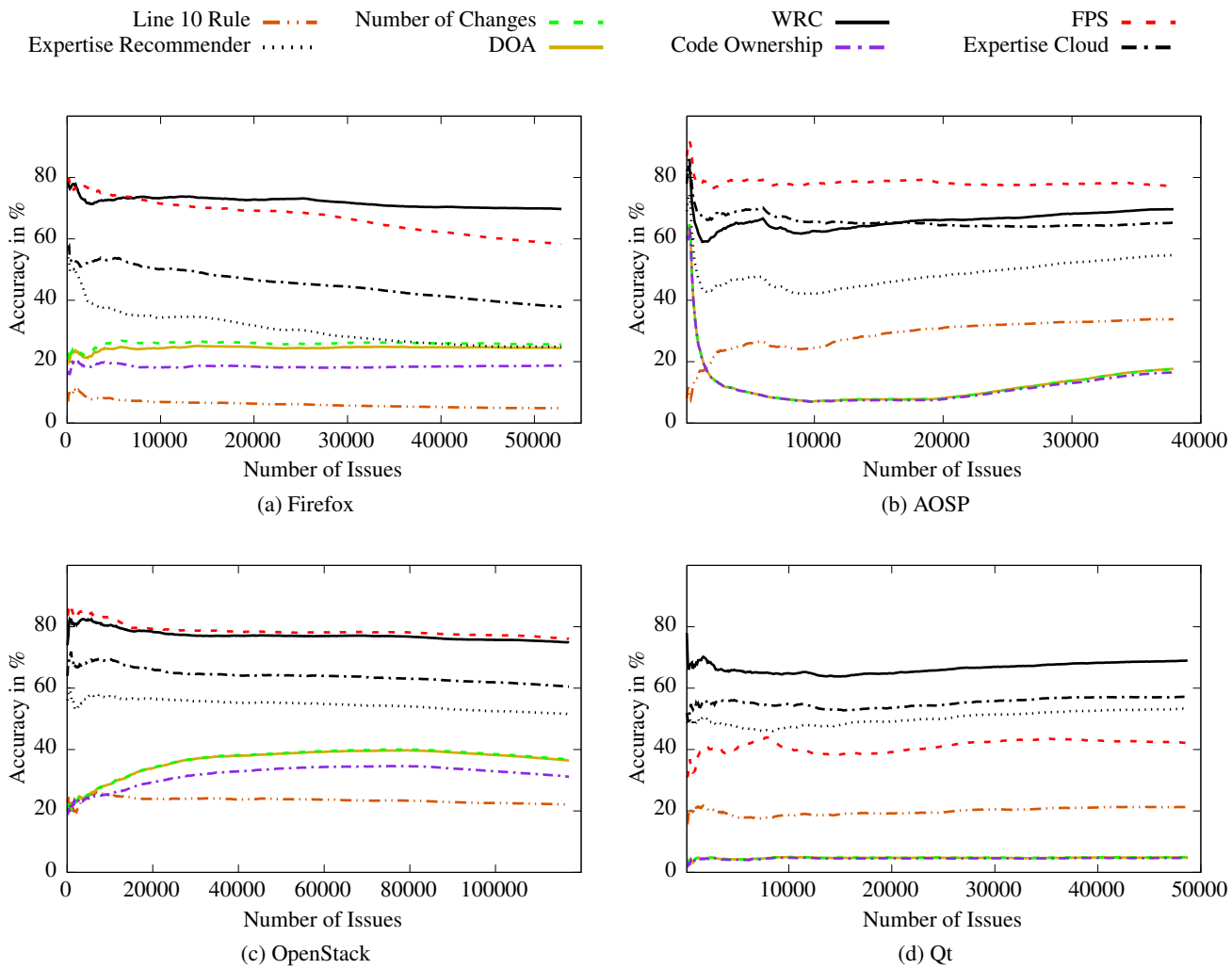

Figure 2: Project results excluding data from the training phase

5.1.4 *Variation between Projects*

The prediction performance varies stronger between projects for some algorithms than for others. Especially the Line 10 Rule and its close relative Expertise Recommender have a top-5 accuracy of only 4.8 % and 24.9 % in Firefox, respectively, but their accuracy for AOSP, OpenStack, and Qt is 22.0 % to 33.8 % and 51.5 % to 54.8 %, respectively. AOSP, OpenStack, and Qt seem to follow a policy similar to the Line 10 Rule without a reviewer recommendation system.

Simple metrics like the Line 10 Rule and Expertise Recommender do not need a specialized reviewer recommendation system, as the usual development environment already contains a VCS client that can display the VCS log, which immediately shows the experts according to Line 10 Rule and Expertise Recommender. Since the Line 10 Rule and Expertise Recommender are very simple metrics of modification expertise that have been shown to be inferior to metrics like DOA [15], a reviewer recommendation tool may supply useful additional information to select reviewers for a patch.

5.1.5 *Variation between Algorithms*

An algorithm either makes a correct or an incorrect prediction for each issue, so the results have a Binomial Distribution. As the algorithms worked on the same data, we used a paired test to check whether the differences are statistically significant: A McNemar test with continuity correction [13, 32] confirms highly significant ($p < 0.001$) differences for most pairs of algorithms, with exception of Expertise Recommender and DOA in Firefox and DOA and Number of Changes in AOSP, which are not significant at $\alpha = 0.05$, and between Expertise Recommender and Number of Changes in Firefox ($p = 0.0016$). These results can be reproduced with the lab package [23].

As visible in the graphs in Figure 2, the algorithms Number of Changes and DOA generally have very similar accuracies. In Qt and AOSP, Code Ownership is also very similar to Number of Changes and DOA. Having large datasets, the differences are still highly significant except for the cases mentioned in the previous paragraph. Even where the differences are statistically highly significant, they are often very small, for example there are only 87 out of 48 709 recommendations for Qt, for which DOA and Number of Changes have differences in correctness, i.e. one of the algorithms recommends a correct reviewer while the other does not. This implies that decreasing the expertise value when other authors change a file, which is the main difference between DOA and Number of Changes in regard to reviewer recommendation, has little influence on the overall results.

Table 2: Top-k accuracy of the algorithms in Firefox, AOSP, OpenStack, and Qt in %, excluding training data

	Firefox			AOSP			OpenStack			Qt		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Line 10 Rule	4.84	4.84	4.84	33.82	33.82	33.82	22.04	22.04	22.04	21.33	21.33	21.33
Expertise Recommender	4.85	15.23	24.85	33.83	50.96	54.75	22.04	42.67	51.52	21.36	44.76	53.33
Number of Changes	12.41	21.50	25.57	12.03	16.62	17.58	20.61	32.47	36.69	3.48	4.67	4.97
Code Ownership	4.95	13.68	18.70	10.37	15.05	16.50	16.23	26.88	31.14	2.88	4.22	4.70
Expertise Cloud	12.94	27.56	37.91	32.99	56.42	65.23	27.91	50.74	60.48	21.98	46.96	57.27
Degree-of-Authorship	11.38	20.28	24.46	11.84	16.63	17.63	19.91	32.00	36.38	3.26	4.55	4.90
Weighted Review Count	35.22	60.62	69.73	46.93	65.69	69.66	43.44	67.35	74.86	37.70	61.97	68.97
File Path Similarity	23.90	45.51	58.24	44.11	69.10	77.32	43.13	67.69	76.00	17.23	34.59	42.12

5.1.6 Time Sensitivity

Figure 2 shows that some algorithms in some FLOSS projects gain prediction performance over the months, while others lose prediction performance. Yet others have a more or less stable prediction performance. Possible causes are changes to the FLOSS projects' review policies, for example if each patch requires more reviewers, it will be easier for the algorithms to recommend one of them correctly. If a FLOSS project grows, it will be harder to recommend a reviewer correctly, because the number of selectable reviewers grows. However, these changes can only affect all algorithms at once and do not explain differences between algorithms.

It is possible to distinguish three different categories of algorithms: First, Line 10 Rule and Expertise Recommender are *time-local*, they use only data for their recommendations that were generated shortly before their execution. Second, Code Ownership and DOA are *forgetful* algorithms. These algorithms consider all historical data for their recommendations, but older data have less influence and expertise decreases during phases of inactivity. For DOA and Code Ownership, an author's expertise decreases when other authors modify files that the author has expertise with. For $\delta < 1$, WRC and FPS are also forgetful, but this is not the case in this study. Third are *time-global* algorithms like Expertise Cloud, Number of Changes, as well as WRC and FPS with $\delta = 1$. They use all historic data for their recommendation and all data have the same value, independently to when they were generated.

Forgetful and time-global algorithms may increase their prediction performance over time, as they can base their recommendations on a larger dataset. However, time-global algorithms may have trouble with changes in the project, because they always take old data into consideration that are based on obsolete and now invalid processes. For example, two reviewers that were active for two years will both be recommended with equal probability, even if one of them was active only five years ago and then left the project and the other is still active. This effect decreases their prediction performance over time and may also affect forgetful algorithms that do not properly forget obsolete data. Conversely, if we observe a changing prediction performance in time-local algorithms, this must be an effect of the project as a whole.

For Firefox, both time-local algorithms stabilize on a slow loss of predictive performance. A possible cause for the loss is Firefox's growth. Consequently, other algorithms like Expertise Cloud and FPS also slowly loose predictive performance. WRC is more or less stable, though, as the benefits of its growing database cancels out the losses through Firefox's growth. Expertise Cloud and FPS both use path similarities, which lets them access much more data for each prediction. These additional data possibly include obsolete data and therefore make them more vulnerable to changes in the project structure.

In this study, FPS and therefore also WRC uses the time prioritization factor $\delta = 1$, so effectively data do not age. This time prioritization turned out to be the best value for FPS in previous studies [50, 51]. However, the next-lower tested value was $\delta = 0.8$, so the optimum may be between 0.8 and 1. In fact, $\delta = 0.8$ seems like a very low value for large FLOSS projects. Firefox, as an example, has 33.5 reviews per day. This means that expertise gained through a review has only $0.8^{33.5} \approx 0.000567$ times its original value after one day. Reasonable values for δ should therefore be much closer to 1 or the time prioritization factor should not have exponential impact. The exact optimum should be the subject of future research, as a forgetful FPS could have a higher long-term prediction performance than a time-global.

5.1.7 Training Phase

The algorithms' prediction performance becomes steady only after some time because of two effects. First, accuracy is an average and therefore varies while the total number of recommendations is low. Second, the algorithms use past reviews and modifications for their recommendations and so they cannot correctly recommend reviewers if the number of past reviews or modifications is still low. In this study, the algorithms are granted a training phase of one year, which is enough to mitigate the second effect.

However, the algorithms have different required training phases. Figure 3 shows aggregated accuracies for Firefox including the training phase. Although WRC has the best prediction performance after the whole study period of six years, it takes about 25 000 issues of training phase to overtake FPS. Apparently, algorithms like FPS and Expertise Cloud have a shorter required training phase than algorithms like WRC. WRC considers only expertise with the file itself, while FPS and Expertise Cloud can induce from other related files if the information about a requested file is insufficient.

6. THREATS TO VALIDITY

This section discusses threats that endanger the validity of our findings. This includes threats to internal validity, including one threat to construct validity, and a threat to external validity.

6.1 Internal Validity

Each algorithm's accuracy is the fraction of issues in which the algorithm recommends reviewers that turned out to be actual reviewers in reality. However, actual reviewers may in some cases not be competent reviewers. This is a threat to construct validity. In an extreme case, an algorithm may recommend reviewers more competent than the actual reviewers are and have a low accuracy because of the differences between recommended and actual reviewers. Nevertheless, the self-selection of reviewers ensures that actual reviewers are at least a good approximation to competent reviewers.

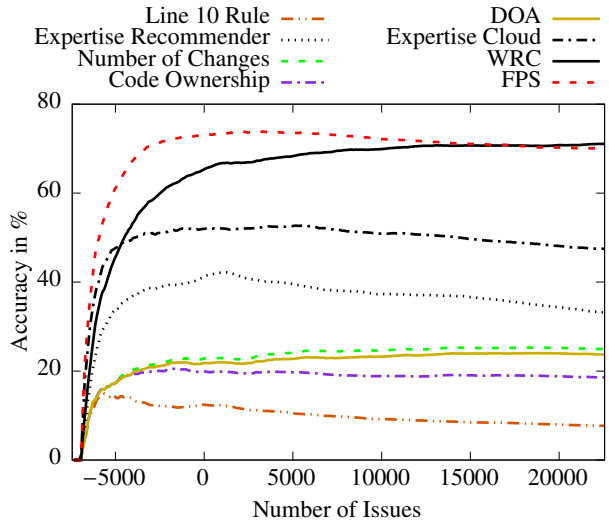

Figure 3: Firefox results with focus on the training phase, issues with negative numbers designate issues of the training phase

Furthermore, even if a recommendation tool fails to recommend more competent reviewers than a manual selection does, it is still faster, which is a benefit by itself [43]. Besides, Section 5.1.2 argues that recommending the most competent reviewer can be a pitfall. Comparison to actual reviewers has also been used in all previous empirical research of this type that we know of [5, 50, 51].

As a considerable special case of the previous threat, the data indicate that AOSP uses the Line 10 Rule to select reviewers (see Section 5.1.4). This and other selection strategies that resemble evaluated algorithms introduce a systematic bias to the comparison. In these cases, a good prediction performance only indicates that the algorithm is currently already used, but not that it is necessarily better than other algorithms. However, some algorithms are consistently better than others among all evaluated projects with their diverse possible reviewer selection strategies, which mitigates this problem.

As discussed in Section 5.1.6, if a FLOSS project grows over time, prediction performance decreases. The observed results therefore depend on the length of observation. As shown in Table 1, each of the four evaluated FLOSS projects had a different study period. This threat impairs comparisons between the four FLOSS projects. Since all algorithms are evaluated within the same time period for each individual FLOSS project, this is no threat to comparisons of different algorithms within each FLOSS project.

The data used for the evaluation contain measurement artifacts. For example, someone might have accidentally flagged a patch in Firefox twice, in which case this reviewer would have twice been awarded review expertise. We included consistency checks to filter out corrupt data items. For example, we filtered out reviews that a single reviewer performed within two seconds, which not only prevents errors like the initial example but also duplicated review entries due to failures of the FLOSS project's databases.

The exact dates of commits to Firefox's VCS are especially vulnerable to measurement errors. These dates stem from the developers' machines. The VCS respects time zones, but these may be incorrectly configured on the developers' machines. Indeed, the time and date on the developers' machines may be completely wrong in some cases. We filtered out issues with reviews that happened before their corresponding patch was submitted. We also filtered issues with obviously wrong time stamps, like when some activities happened outside of our study period. Despite our efforts to eliminate measurement artifacts, a complete check of the data was not possible and therefore measurement artifacts may still remain.

When Firefox migrated their VCS from CVS to hg, they imported all existing source code in a single commit with the author "hg@mozilla.com". The algorithms assumed an enormous modification expertise for hg@mozilla.com for some time after the migration. The training phase however allowed to recover from these types of problems. To test whether hg@mozilla.com biased the results, we also evaluated results after filtering out all issues for which one of the algorithms had recommended hg@mozilla.com as a reviewer. This had only minor impact on the results and especially did not change the order of algorithms in terms of prediction performance.

6.2 External Validity

Section 5.1.4 discusses how the prediction performance varies between the four evaluated FLOSS projects. While there is variation between projects, most results apply universally to all four evaluated FLOSS projects. This indicates that the results also apply to many other FLOSS projects and are, in fact, to a large extent independent from the specific FLOSS project. There may be exceptions though, especially for those algorithms that vary stronger already between the four evaluated projects, like the Line 10 Rule.

7. CONCLUSION AND FUTURE WORK

Submitted patches to FLOSS projects must usually undergo a code review before they are accepted. Problems with reviewer assignment cause delayed and forgotten patches. Reviewer recommendation systems may help with reviewer assignment problems, but only if they accurately recommend competent reviewers, i.e. have a good prediction performance.

This study compared the prediction performance of eight reviewer recommendation algorithms using historical data from four large FLOSS projects. Six algorithms are metrics of modification expertise that have been described in literature, but had not been used for reviewer recommendation before. The algorithm FPS was included as a comparison reviewer recommendation algorithm. The eighth algorithm WRC is a pre-stage to FPS, but had not yet been considered as a recommendation algorithm. On the bottom line, WRC achieves the best results with the additional advantage of its simplicity.

The data set used in our evaluation is more than an order of magnitude larger than those used in previous studies, e.g. [51, 56]. Our results show that calculating reviewer recommendations is possible in large FLOSS projects. We have also implemented a Proof-of-Concept realization of a reviewer recommender using WRC as part of a larger system [22].

Our findings raise questions open for future research: Can a hybrid of existing algorithms combine the advantages of those algorithms? Can additional information like the current group of core reviewers improve the algorithms' performance? Can a better time prioritization improve the prediction performance of FPS? Do the tested algorithms really recommend competent reviewers or only common reviewers? When should an algorithm make a trade-off between competency and workload to relieve competent but busy reviewers?

Furthermore, when is a review necessary at all and which patches should undergo a more rigorous review [17]?

8. REFERENCES

[1] O. Alonso, P. T. Devanbu, and M. Gertz. Expertise identification and visualization from cvs. In *Proceedings of*

the 2008 International Working Conference on Mining Software Repositories, MSR '08, pages 125–128, New York, NY, USA, 2008. ACM.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[3] J. Anvik and G. Murphy. Determining implementation expertise from bug reports. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pages 2–2, May 2007.

[4] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.*, 20(3):10:1–10:35, Aug. 2011.

[5] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 931–940, Piscataway, NJ, USA, 2013. IEEE Press.

[6] O. Baysal, O. Kononenko, R. Holmes, and M. Godfrey. The secret life of patches: A firefox case study. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 447–455, 2012.

[7] D. Black. Partial justification of the borda count. *Public Choice*, 28(1):1–15, 1976.

[8] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1767–1772, New York, NY, USA, 2006. ACM.

[9] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.

[10] D. Cosley, S. K. Lam, I. Albert, J. A. Konstan, and J. Riedl. Is seeing believing?: How recommender system interfaces affect users' opinions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 585–592, New York, NY, USA, 2003. ACM.

[11] K. Crowston and B. Scozzi. Bug fixing practices within free/libre open source software development teams. *Journal of Database Management*, 19(2):1–30, 2008.

[12] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[13] A. L. Edwards. Note on the "correction for continuity" in testing the significance of the difference between correlated proportions. *Psychometrika*, 13(3):185–187, 1948.

[14] M. Fowler. Continuous integration, May 2006. http://martinfowler.com/articles/continuousIntegration.html [accessed 2015-08-28].

[15] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Trans. Softw. Eng. Methodol.*, 23(2):14:1–14:42, Apr. 2014.

[16] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 385–394, New York, NY, USA, 2010. ACM.

[17] N. Froyd. On code review and commit policies. Blog, June 2014. https://blog.mozilla.org/nfroyd/2014/06/23/on-code-review-and-commit-policies/ [accessed 2015-08-27].

[18] J. Geldenhuys. Finding the core developers. In *39th Euromicro Conference on Software Engineering and Advanced Applications*, volume 0, pages 447–450, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[19] GitHub, Inc. Github website, 2016. https://github.com/ [accessed 2016-04-19].

[20] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 113–122, Sept 2005.

[21] K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, and H. Iida. Who does what during a code review? datasets of oss peer review repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 49–52. IEEE, May 2013.

[22] C. Hannebauer and V. Gruhn. Implementation of a wiki development environment. In *Proceedings of the the 15th SoMeT_16*, Frontiers in Artificial Intelligence and Applications, 2016.

[23] C. Hannebauer, M. Patalas, and V. Gruhn. Lab package for the evaluation in this study, 2016. https://www.uni-due.de/~hw0433/ [accessed 2016-07-14].

[24] J. W. Hunt and M. D.McIlroy. An algorithm for differential file comparison. Computing Science Technical Report 41, Bell Laboratories, July 1976.

[25] G. Jeong, S. Kim, T. Zimmermann, and K. Yi. Improving code review by predicting reviewers and acceptance of patches. ROSAEC MEMO ROSAEC-2009-006, Research On Software Analysis for Error-free Computing Center, Seoul National University, Sept. 2009.

[26] J. Jiang, J.-H. He, and X.-Y. Chen. Coredevrec: Automatic core member recommendation for contribution evaluation. *Journal of Computer Science and Technology*, 30(5):998–1016, 2015.

[27] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006. ACM.

[28] T. Koponen. Life cycle of defects in open source software projects. In *OSS2006: Open Source Systems (IFIP 2.13)*, pages 195 – 200. Springer, 2006.

[29] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[30] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 131–140, May 2009.

[31] D. W. McDonald and M. S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 231–240, New York, NY, USA, 2000. ACM.

[32] Q. McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

[33] S. Minto and G. Murphy. Recommending emergent teams. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pages 5–5, 2007.

[34] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 503–512, New York, NY, USA, 2002. ACM.

[35] Mozilla Developer Network. How to submit a patch, Feb. 2015. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/How_to_Submit_a_Patch [accessed 2015-08-28].

[36] Mozilla Foundation. Bugzilla@mozilla, May 2015. https://bugzilla.mozilla.org [accessed 2015-08-28].

[37] Mozilla Foundation. Main hg repository for firefox, May 2015. https://hg.mozilla.org/mozilla-central/ [accessed 2015-08-28].

[38] Mozilla Wiki Contributors. Firefox/code review – help, my patch isn't being reviewed, Aug. 2015. https://wiki.mozilla.org/Firefox/Code_Review#Help.2C_my_patch_isn.27t_being_reviewed [accessed 2015-08-24].

[39] T. T. Nguyen, T. Nguyen, E. Duesterwald, T. Klinger, and P. Santhanam. Inferring developer expertise through defect analysis. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1297 –1300, June 2012.

[40] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 9–18, New York, NY, USA, 2009. ACM.

[41] G. Robles, J. Gonzalez-Barahona, and I. Herraiz. Evolution of the core team of developers in libre software projects. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 167 –170, may 2009.

[42] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 121–124, New York, NY, USA, 2008. ACM.

[43] B. D. Sethanandha, B. Massey, and W. Jones. On the need for oss patch contribution tools. In *Proceedings of the Second International Workshop on Building Sustainable Open Source Communities*, 2010.

[44] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 2–11, Piscataway, NJ, USA, 2013. IEEE Press.

[45] I. Steinmacher, I. S. Wiese, A. L. Schwerz, R. L. Roberto, J. E. Ferreira, and M. A. Gerosa. Historical analysis of message contents to recommend issues to open source software contributors. *Revista Eletrônica de Sistemas de Informação*, 2014.

[46] S. Stünkel, C. Hannebauer, and M. Patalas. Expertise Explorer, 2016. https://github.com/paluno/ExpertiseExplorer [accessed 2016-07-14].

[47] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 365–375, New York, NY, USA, 2011. ACM.

[48] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How Do Software Engineers Understand Code Changes? An Exploratory Study in Industry. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*. Research Triangle Park, North Carolina, November 2012.

[49] The Apache Software Foundation. Apache http server project, Apr. 2015. https://httpd.apache.org/ [accessed 2015-08-28].

[50] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pages 119–122, New York, NY, USA, 2014. ACM.

[51] P. Thongtanunam, C. Tantithamthavorn, R. Kula, N. Yoshida, H. Iida, and K.-I. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150, March 2015.

[52] L. Torvalds and G. Moody. Interview: Linus Torvalds – I don't read code any more, Nov. 2012. http://www.h-online.com/open/features/Interview-Linus-Torvalds-I-don-t-read-code-any-more-1748462.html [accessed 2015-08-28].

[53] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pages 1–1, 2007.

[54] K. E. Wiegers. *Peer Reviews in Software – A Practical Guide*. Addison-Wesley Information Technology Series. Addison Wesley, 2002.

[55] Y. Yu, H. Wang, G. Yin, and T. Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[56] M. B. Zanjani, H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2015.

[57] M. Zhou and A. Mockus. What make long term contributors: Willingness and opportunity in oss community. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 518–528, Piscataway, NJ, USA, 2012. IEEE Press.