2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

CORRECT: Code Reviewer Recommendation in GitHub Based on Cross-Project and Technology Experience

Mohammad Masudur Rahman Chanchal K. Roy †Jason A. Collins
University of Saskatchewan, Canada, †Google Inc., USA
{masud.rahman, chanchal.roy}@usask.ca, †jasonco@google.com

ABSTRACT

Peer code review locates common coding rule violations and simple logical errors in the early phases of software development, and thus reduces overall cost. However, in GitHub, identifying an appropriate code reviewer for a pull request is a non-trivial task given that reliable information for reviewer identification is often not readily available. In this paper, we propose a code reviewer recommendation technique that considers not only the relevant cross-project work history (e.g., external library experience) but also the experience of a developer in certain specialized technologies associated with a pull request for determining her expertise as a potential code reviewer. We first motivate our technique using an exploratory study with 10 commercial projects and 10 associated libraries external to those projects. Experiments using 17,115 pull requests from 10 commercial projects and six open source projects show that our technique provides 85%–92% recommendation accuracy, about 86% precision and 79%–81% recall in code reviewer recommendation, which are highly promising. Comparison with the state-of-the-art technique also validates the empirical findings and the superiority of our recommendation technique.

CCS Concepts

•Software and its engineering → Software notations and tools; *Code Review;* Recommendation; **•Collaboration in software development** → Programming teams;

Keywords

Code reviewer recommendation, cross-project experience, specialized technology experience, GitHub, pull request

1. INTRODUCTION

Software development practices have dramatically changed over the last decade, and software projects are now developed not only in a collaborative environment but also in a distributed fashion [21]. GitHub, a collaborative and distributed development framework, promotes pull-request based development where a new developer forks from an existing repository (i.e., project), works on certain module of her interest, and then submits the changed files to the repository using a pull request [9]. One or more expert developers from the base repository, referred by the submitter, then review(s) the code carefully before accepting the changes as a contribution to the codebase. Such code review is reported as highly effective for locating common coding rule violations or for performing simple logical verifications [4, 7, 14]. It also helps identify the issues (e.g., vulnerabilities) in the code in the early phases of development, and thus reduces overall cost for the software project [4, 7]. However, choosing an appropriate developer for code review for a pull request is a significant challenge [5], and to date, GitHub does not provide any support for this. Reliable information on developer's expertise (e.g., technology skill) for the review is often not readily available, and it needs to be carefully mined from the codebase. Thus, reviewer identification task is even more challenging and time-consuming for the novice developers who are less familiar with the codebase as well as the skills of the hundreds of fellow developers. Such challenge is prevalent not only in open source development but also in the industrial environment where a company uses GitHub for commercial development, and encourages developer collaborations such as peer code review.

Fortunately, there have been several studies that recommend code reviewers by analyzing past code review history (e.g., line change history [5], review comments [19, 22]), project directory structure [16, 17], and developer collaboration network [22]. Similarly, studies on expert recommendation for software bugs also exploit different software artifacts [11, 13] and developer communication history [20]. Thus, existing studies mostly rely on the work history of a developer within a particular project and her collaboration history with other developers for determining her expertise. However, no studies consider the cross-project experience or the experience in various specialized technologies of a developer, and thus they fall short in handling certain challenges.

First, in the industry, software developers often reuse software components (e.g., libraries) that are previously developed by themselves for low cost and faster development. Thus, their contributions scatter throughout different projects in the code repositories of the organization, and such contributions are a great proxy to their experience. Unfortunately, the existing studies on code reviewer recommendation completely ignore such information in expertise determination, and their recommendations are merely based on the contri-

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2889160.2889244

bution details within a particular project. Second, underlying tools and technologies of software projects are rapidly changing, and modern projects often involve different specialized and cutting edge technologies such as `map-reduce, task queues, urlfetch, memcache` and `pipeline`. Hence, code reviewers for a pull request are expected to have expertise in such technologies. However, neither mining of the revision history of changed files nor mining of the developer collaboration history, as the existing studies do, might be sufficient enough to ensure that. Thus, a technique that can analyze both relevant cross-project experience and specialized technology experience of a developer for a pull request, is likely to overcome the above challenges.

In this paper, we propose a novel code reviewer recommendation technique–CoRReCT (**Co**de **R**eviewer **Re**commendation based on **C**ross-project and **T**echnology experience), for pull requests at GitHub. It estimates code review expertise of a developer for a pull request by analyzing her past work experience with (1) external software libraries and (2) specialized technologies used by the pull request. Reference to the external libraries (i.e., software units external to the working project) in the code generally suggests one's working experience with such libraries, and we call it *cross-project experience*. Our baseline idea is– *"if a past pull request uses similar external libraries or similar specialized technologies to the current pull request, then the past request is relevant to the current request, and thus, its reviewers are also potential candidates for the code review of the current request"*. We first mine the library and technology information from a pull request using static analysis, and then identify the relevant requests in terms of library and technology similarities from the recently submitted request collection. We then propagate the similarity score for each relevant request to its corresponding code reviewers as a proxy to the shared experience in external libraries and specialized technologies with the current request. Thus, each of the candidates accumulates scores for all relevant requests, and finally, the technique returns a ranked list of code reviewers. While the technique adopts heuristics for ranking, our contribution lies in identifying the appropriate proxies to code review expertise. To the best of our knowledge, ours is the first study that exploits the benefits of using cross-project experience and specialized technology experience (of the developers) in recommending code reviewers for a pull request. We adopt a client-server model for our technique where the client module is packaged as a Google Chrome plug-in, and the server module is hosted as a web service. Both modules are available online [2] for replication or third party use.

In order to motivate cross-project experience and specialized technology experience as a proxy to code review expertise, we first conducted an exploratory study using 10 commercial projects and 10 libraries from the codebase of a local reputed software company (for the sake of anonymity, we call this *ABC*) with more than 150 employees. Experiments with 10 commercial projects and six open source projects totaling 17,115 pull requests show that our technique recommends code reviewers with 85%– 92% recommendation accuracy, about 86% precision and 79%–81% recall. Furthermore, comparison with the state-of-the-art also confirms the empirical findings and superiority of our technique. Thus, we make the following contributions in the paper:

- An exploratory study that not only analyzes the usage of external libraries and specialized technologies in the commercial projects but also investigates their potential for code review.
- Two novel expertise dimensions–*cross-project experience* and *specialized technology experience* for code reviewer recommendation for pull requests at GitHub.
- Comprehensive evaluation of the proposed technique with both commercial and open source projects using popular performance metrics, and comparison with the state-of-the-art.
- Implementation of our recommendation technique as a web service (server) and a Chrome plug-in (client).

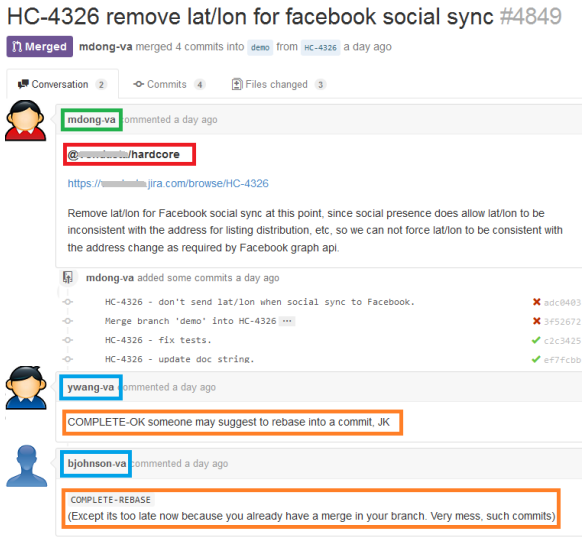

Figure 1: Code review interface at GitHub

The rest of the paper is structured as follows– Section 2 provides an overview on modern code reviw, Section 3 focuses on our conducted exploratory study, and Section 4 describes our proposed recommendation technique. Section 5 discusses the evaluation and validation details, and Section 6 focuses on threats to the validity of our findings. Section 7 discusses related studies from the literature, and finally Section 8 concludes the paper with future work.

2. MODERN CODE REVIEW

Code review refers to a manual assessment of source code that identifies potential defects (e.g., logical errors) and quality problems (e.g., coding rule violations) in the code [6]. In recent years, code review is assisted with different tools, which is less formal and more popular than the traditional review techniques [5]. Such code review is termed as *Modern Code Review* (MCR) [6]. It is widely adopted both by the commercial organizations (e.g., Google, Microsoft) and by the open-source communities (e.g., Android, LibreOffice). Existing code review tools such as *ReviewBot* [5] and *RevFinder* [17] are mostly based on Gerrit, a web-based code review system. GitHub also provides a similar feature for conducting code review by human developers through pull requests where a developer can request her peers for code review during a pull request submission.

For example, developer *mdong-va* (i.e., green box, Fig. 1) requests a development team– *hardcore* (i.e., red box, Fig. 1) for code review during the submission of pull request #4849. Two developers–*ywang-va* and *bjohnson-va* (i.e., blue boxes) from the team analyze the commits associated with the pull request, perform the code review, and then post their feed-

Table 1: ABC Company Projects

Project	#Files[1]	#PR[2]	#PRR[3]	Project	#Files	#PR	#PRR
CS	3,733	4,560	57	SR	2,139	1,927	36
ARM	2,035	969	33	NB	1,524	828	32
SM	2,026	1,291	36	VBC	1,894	1,050	36
VW	1,475	787	16	AA	2,174	1,313	46
MS	2,227	1,156	36	ST	2,676	1,397	28

[1]Source files, [2]Pull requests, [3]Pull request reviewers

Table 2: ABC Company Libraries

Library	#Files[1]	#TC[2]	#TA[3]	Library	#Files	#TC	#TA
vapi	631	727	24	vpubsub	750	428	20
vform	545	893	27	vtest	511	222	18
vbackup	469	236	15	vauth	421	200	19
vlogs	826	243	17	vmonitor	532	213	12
vautil	1294	269	20	vpipeline	1470	228	13

[1]Source files, [2]Total commits, [3]Total authors

Table 3: Specialized Technologies in ABC Projects

Technology	Functionality	Technology	Functionality
`taskqueue`	task scheduling	`deferred`	task scheduling
`mapreduce`	distributed computing	`blobstore`	data storage
`urlfetch`	HTTP communication	`jinja2`	template engine
`search`	item search	`modules`	app. factorization
`ndb`	data storage	`socket`	networking

back using comments (i.e., orange boxes, Fig. 1). Unfortunately, despite assistance from the static analysis tools [5], effective code review still remains a challenge, and identifying an appropriate reviewer is a non-trivial task. To date, both reviewer selection and code review are also performed manually at GitHub. In this research, we thus recommend appropriate developers (e.g., *ywang-va, bjohnson-va*) for such code review task (e.g., Fig. 1) at GitHub.

3. EXPLORATORY STUDY

Since our proposed technique is based on cross-project work history (i.e., external library experience) and specialized technology experience, it is important that we first conduct an exploratory study to find out to what extent such situations in fact occur and whether such information can help recommend code reviewers. We thus conduct an exploratory study with 20 commercial projects and libraries targeting 10 specialized technologies. In our study, we answer three research questions as follows:

- **Exp-RQ$_1$:** How frequently do the commercial software projects reuse libraries from the codebase?
- **Exp-RO$_2$:** Does the experience of a developer with such libraries matter in code reviewer selection?
- **Exp-RQ$_3$:** How frequently do the commercial software projects use specialized technologies?

3.1 Dataset Collection

ABC codebase hosts 30 commercial projects and 21 libraries (according to March, 2015) having various sizes and functionalities. They are based on *Google Cloud* platform and are mostly written in *Python*. In order to perform a meaningful analysis on the codebase, we select (1) the top 10 projects with more than 750 closed pull requests and (2) the top 10 libraries that are used at least 10 times on average in each of those projects for the exploratory study. Table 1 and Table 2 show the details of the selected projects and libraries respectively. We also consider 10 specialized technologies that are at least used five times on average in each of those projects for the study. Table 3 shows the selected technologies, and their specialized functionalities.

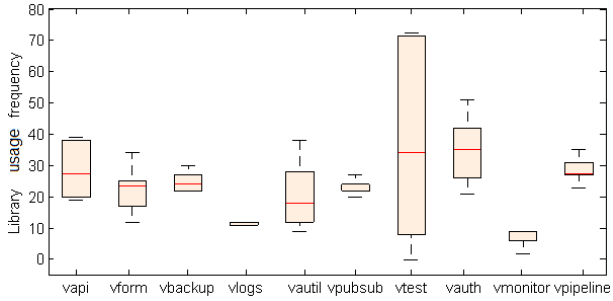

Figure 2: Usage frequency of libraries (Table 2) in ABC Company projects (Table 1)

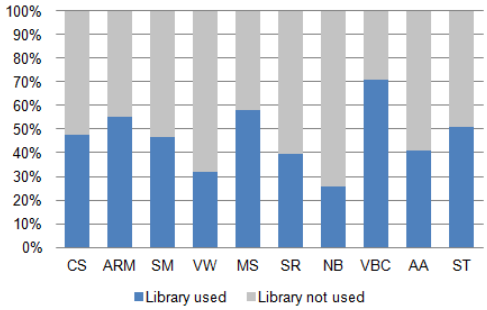

Figure 3: Library usage ratio in pull requests

3.2 Answering Exp-RQ$_1$: Frequency of library use in the commercial projects

Each of the selected projects nearly depends on all of the chosen libraries for their functionalities (e.g., authentication, utility). However, in order to answer Exp-RQ$_1$, we need more focused statistics which are provided using the following analyses both with source files and pull requests.

Library Use in Source Files: In Python programming, external libraries are generally attached to a source file using import statements. We analyze the import statements (e.g., `import vform, from vlogs import AbstractTracer`) from all the source files of each of the selected projects, and extract the library names (e.g., `vform, vlogs`) using a custom-built Python AST parser. The goal is to determine the extent to which a particular external library (e.g., `vform`) is used in different projects.

Fig. 2 shows the box plot of frequency of occurrence in different commercial projects for the selected libraries. We note that *vtest*, a testing library, has a large variance in usage frequency with a median frequency around 35 and a maximum frequency around 70. This means that the library is used to various degrees in different projects for testing purposes. The similar median also goes for *vauth* library (i.e., provides authentication support) whereas the other libraries have a median frequency around 25. Thus, each project is densely connected with most of the libraries, the overall dependency of the projects on the external libraries is remarkable due to their dedicated functionalities.

Library Use in Pull Requests: In order to provide further insight on library use, we study the pull requests of each of the projects. Each of the requests contains one or more commits, and we analyze the changed files in those commits, and look for the occurrences of the selected libraries (Table 2). Fig. 3 shows the fraction of the pull requests that use any of the 10 libraries (Table 2) for each project. We note that about 50% of the requests on average referred to those libraries in their changed files which is undoubtedly a signif-

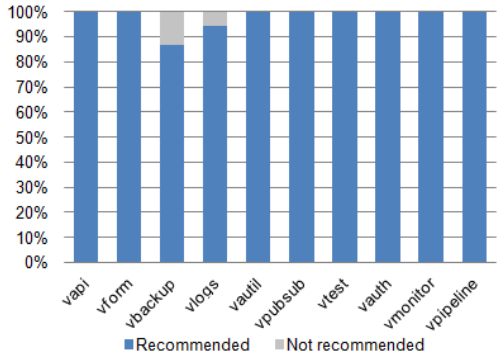

Figure 4: Library authors as code reviewers for relevant pull requests in the selected projects

icant amount. For example, *CS* is a large project containing 4,560 pull requests, and 47.63% of its requests used those libraries which clearly reports an extreme dependency of the projects on the selected libraries.

Thus, to answer the first exploratory research question– Exp-RQ$_1$, commercial projects frequently use the libraries from codebase for their dedicated functionalities (e.g., authentication, testing). We observe that some of the projects refer to a single library even up to 70 times, and about 50% of their pull requests involve one or more of those libraries.

3.3 Answering Exp-RQ$_2$: Role of experience with external libraries in code review

In order to investigate if the experience of a developer with the included libraries into a project does matter or not in the code reviews for the project, we analyze cross-project contributions of the developers. For each selected library from the codebase, we identify the developers who authored at least one of the merged commits, and create an *author list*. We also identify all the pull requests from each of the selected projects that use a certain library in the changed files, and then develop a *reviewer list* by collecting the reviewers of the corresponding requests. We then analyze both the author list and reviewer list, and determine if the library authors are later recommended as code reviewers or not.

Fig. 4 shows the percentage of the authors for each selected library (Table 2) who are recommended as code reviewers for different projects using that library. We note that for each of the selected libraries except *vbackup* and *vlogs*, all authors (i.e., 100%) are later recommended as reviewers for pull requests involving those libraries. For *vbackup* and *vlogs*, such authors are also about 90%. Thus, the finding from our empirical dataset clearly shows that the first-hand working experience (i.e., authorship) of a developer with relevant libraries is greatly valued in code reviewer selection, which necessarily answers our second exploratory research question– Exp-RQ$_2$. Furthermore, the finding also motivates *cross-project experience* (i.e., work experience with external libraries) as an expertise dimension for code review. Since the library author list largely overlaps (i.e., 98% on average) with the reviewer list from relevant pull requests, we mostly exploit such reviewer lists in our technique for code reviewer recommendation.

3.4 Answering Exp-RQ$_3$: Frequency of technology use in commercial projects

Our selected software projects are based on Google cloud

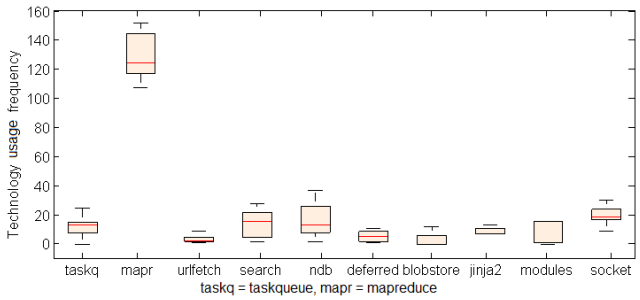

Figure 5: Usage frequency of the selected technologies (Table 3) in ABC Company projects (Table 1)

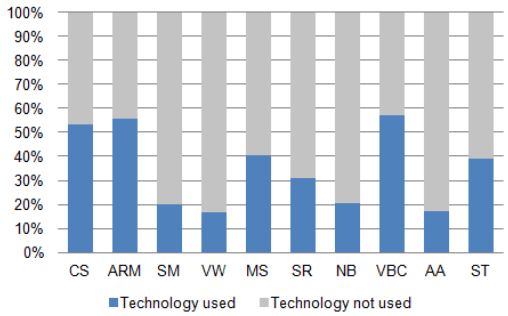

Figure 6: Technology usage ratio in pull requests

platform, and they generally use *Google App Engine (GAE)* technologies (Table 3). In order to better understand the extent to which each of these technologies is used in those projects, we perform our analyses using both source code files and pull requests from the projects.

Technology Use in Source Files: Since *Google App Engine* framework targets a complete hosting solution, its service comes with a set of specialized technologies such as `taskqueue, pipeline, memcache, mapreduce` and so on. Software projects built on that framework generally use those technologies, and include them in the source files using import statements (e.g., `from google.appengine.api import taskqueue`). We analyze such statements from each of the source files from the selected projects, and extract the technologies (e.g., `taskqueue`). The goal is to determine the extent to which each of the selected technologies (Table 3) is used in different software projects.

Fig. 5 shows the five point statistics on the frequency of occurrence for each of the selected technologies in different projects. We note that *mapreduce* is the most widely used technology with a median usage frequency of about 125 and a maximum of 150. This finding suggests that the commercial projects under study heavily use distributed computing, and *mapreduce* is a major building block for them. The other technologies such as *search* and *socket* also have a median frequency around 20 and a maximum frequency around 30. Thus, the overall dependency of each of the selected projects on such technologies is noteworthy given that these technologies provide advanced or specialized computing features.

Technology Use in Pull Requests: In order to further investigate the use of specialized technologies in the projects, we study the pull requests from each project. We analyze the changed files from each of the pull requests, and look for the technologies referred to in the `import` statements. Fig. 6 shows the fraction of the pull requests that refer to any of the 10 selected technologies (Table 3) for each project. We note that about 35.09% of the requests on average used

those technologies in their changed files which is undoubtedly a significant amount. For example, *CS* and *VBC* are two large projects containing 4,560 and 1,050 pull requests respectively, and 53.18% and 56.95% of their requests used the selected technologies. This clearly shows a significant dependency of the selected commercial software projects on the specialized technologies.

Thus, to answer the third exploratory research question– Exp-RQ$_3$, commercial software projects frequently adopt specialized technologies such as Google App Engine technologies. We note that some of the projects refer to a single technology even up to a maximum of 150 times, and about 35% of their pull requests use one or more of the selected technologies. Thus, experience with such specialized technologies is also an important prerequisite for performing code review for the pull requests.

4. CORRECT: PROPOSED TECHNIQUE

Since findings from the exploratory study (Section 3) suggest that experience with external software libraries or specialized technologies is a useful proxy to code review expertise, we exploit such information in our proposed technique. Fig. 7 shows the schematic diagram of our proposed technique– *CORRECT* for code reviewer recommendation for pull requests at GitHub. In our technique, we analyze the review history of past pull requests from a project, identify the relevant pull requests in terms of external library or specialized technology similarity, and then recommend code reviewers from those requests. This section first explains our code reviewer ranking algorithm, and then discusses the implementation details of our developed prototype.

4.1 An Overview of CORRECT

CORRECT analyzes past pull requests and their corresponding review history from a project for ranking and then recommending code reviewers for an incoming pull request. We believe that the developers who have reviewing experience on similar (i.e., relevant) pull requests are suitable candidates for reviewing that request [5, 17]. In this research, we hypothesize similarity between two pull requests based on their shared libraries (e.g., `vapi, vform`) and adopted technologies (e.g., `taskqueue, ndb`) in the changed files. Thus, any pull request, R_i, can be considered as a combination of the tokens for external libraries (L_{ext}) and specialized technologies ($T_{special}$) used in the request.

$$R_i = \{L_i \mid L_i \epsilon L_{ext}\} \cup \{T_i \mid T_i \epsilon T_{special}\}$$

Two pull requests (R_i, R_{i-1}) can be considered similar if they share a set of external software libraries or specialized technologies in the changed files.

$$R_i \sim R_{i-1} \mid (\{L_i\} \cap \{L_{i-1}\}) \neq \phi \vee (\{T_i\} \cap \{T_{i-1}\}) \neq \phi$$

We consider 30 (i.e., best performing heuristic count) previously closed (i.e., merged, rejected) and similar pull requests from the code review history for analysis. It should be noted that GitHub stores the pull requests with an incremental index, and we use that index for collecting the past requests from a project. Our experiments also suggest that relevant requests are mostly found in a chunk of consecutive requests, and thus, we choose a list of consecutive requests from the most recent history. We then estimate similarity degree between the current request (R_c) and each of the collected past requests (R_i) using cosine similarity measure. We collect the library or technology names from each pull request, consider them as a *bag of tokens* (i.e., a collection of tokens with no fixed order), and decompose each token having dotted (e.g., `app.views.filters.vff`) or underscored (e.g., `datetime_utils`) structures. We then prepare a combined set of tokens, C, from the two sets corresponding to the two pull requests and calculate *cosine similarity*, $CS(R_c, R_i)$, as follows.

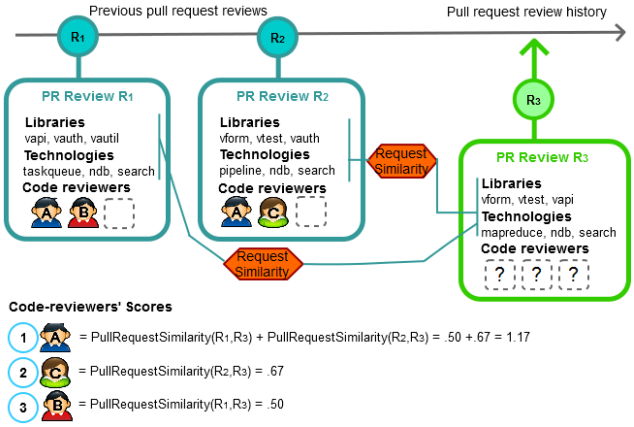

Figure 7: Schematic diagram of the proposed technique– CORRECT

$$CS(R_c, R_i) = \frac{\sum_{k=1}^{n} C_{ck} \times C_{ik}}{\sqrt{\sum_{k=1}^{n} C_{ck}^2} \times \sqrt{\sum_{k=1}^{n} C_{ik}^2}} \quad (1)$$

Here, C_{ck} represents frequency of k^{th} token from C in set R_c (i.e., token set from current request), and C_{ik} represents that frequency in set R_i (i.e., token set from the past request). This measure values from zero (i.e., complete dissimilarity in libraries and technologies) to one (i.e., complete similarity). We then propagate the similarity estimates (as a proxy to review expertise) to the corresponding code reviewers (LR) of the past requests (R_i).

$$LR[r] = \sum_{i=1}^{N} CS(R_c, R_i) \mid reviews(r, R_i) \wedge (c > i) \quad (2)$$

Thus, according to our proposed idea, the developers who have more experience on the attached external libraries (i.e., cross-project experience) and the adopted specialized technologies in the changed files of R_c, are more appropriate for code review than the ones having less experience.

Example: Let us consider R_3 (Fig. 7) is a pull request to be submitted, and the submitter is looking for one or more code reviewers for the request. R_1 and R_2 are two past requests similar to R_3 containing one or more changed files. From Fig. 7, we note that each of R_1 and R_2 includes three libraries, adopts three specialized technologies, and is reviewed by a different set of developers. Similarly, R_3 also includes three libraries from the codebase and adopts three specialized technologies in the changed files. In order to recommend reviewers for R_3, CORRECT first determines the *cosine similarity* (Equation 1) between libraries and technologies of R_3 and those of R_1 and R_2. It then applies those scores (Equation 2) to the corresponding reviewers of R_1 and R_2. Thus, the developers who have the most review experience with similar past requests, bubble up in the ranked list for code reviewers. From Fig. 7, we see that reviewer A scores the top (i.e., 1.17) in the list according to our ranking

Algorithm 1 Code Reviewer Ranking Algorithm

```
1: procedure CORRECT(R_n)                    ▷ R_n: new pull request
2:     LR ← {}                               ▷ list of code reviewers
3:     ▷ VA libraries and specialized technologies used
4:     libtech ← getLibTechTokens(R_n)
5:     ▷ Collecting previously closed pull requests
6:     pastPRequests ← getAllPastPRequests(R_n)
7:     pastPRequests ← getRecentPRs(pastPRequests)
8:     ▷ Accessing & analyzing each pull request
9:     for PullRequest R_i ∈ pastPRequests do
10:        libtech_i ← getLibTechTokens(R_i)
11:        ▷ Calculate similarity score between two requests
12:        S_cos ← CosineSimilarity(libtech, libtech_i)
13:        ▷ Assigning scores to corresponding reviewers
14:        pastReviewers ← getPRReviewers(R_i)
15:        for PR-Reviewer r ∈ pastReviewers do
16:            LR[r].score ← LR[r].score + S_cos
17:        end for
18:    end for
19:    ▷ Creating ranked list of reviewers
20:    RLR ←sortReviewersByScore(LR)
21:    return RLR
22: end procedure
23: procedure GETLIBTECHTOKENS(R_p)
24:    ▷ R_p: pull request
25:    ▷ Collecting VA libraries included
26:    libs ← getVALibTokens(R_p)
27:    ▷ Collecting specialized technologies adopted
28:    techs ← getSpecTechTokens(R_p)
29:    ▷ Returning combined token list
30:    return concatTokens(libs, techs)
31: end procedure
```

algorithm, and thus, A is recommended as the code reviewer for R_3. We recommend the top five code reviewers [5, 17] from such a ranked list for a pull request.

4.2 Code Reviewer Ranking Algorithm

The pseudo-code of our proposed ranking algorithm– CORRECT is given in Algorithm 1. The algorithm takes a pull request–R_n as an input and returns a ranked list of code reviewers–RLR as the output. First, the algorithm extracts the *external software libraries* included and the *specialized technologies* used in the changed files of R_n (Line 4) by invoking another procedure–`getlibTechTokens(`R_p`)` (Line 23 to Line 30). It then collects the most recent and previously closed pull requests from the project (Line 6 and Line 7). Our iterative experiments on ABC dataset suggest that past 30 pull requests (in contrast to all requests from the history [17]) are enough to sufficiently recommend the code reviewers. We thus collect the top 30 pull requests from the most recent request history. The algorithm then browses through each (R_i) of the past requests, extracts their libraries and technologies, and determines similarity (S_{cos}) between R_i and R_n using cosine similarity measure (Line 9 to Line 12). It then collects the corresponding reviewers of R_i, and assigns the score–S_{cos} to each of those reviewers. Thus, the frequent reviewers from the similar (i.e., relevant) pull requests get the maximum scores (Line 14 to Line 17). Since the exploratory study (Section 3) suggests library experience and technology experience as useful proxies to code review expertise, and we identify relevant past requests using that information, the heuristic expertise scores (S_{cos}) are generally propagated to the frequent and expert reviewers. The reviewers' names and their scores are stored in a key-value pair list–LR. Once the outer loop (Line 9 to Line 18) terminates, LR is sorted by score, and top five code reviewers (as suggested by literature [5, 17] and *ABC* developers) from the ranked list–RLR are recommended for R_n (Line 20 and Line 21).

Table 4: Experimental Dataset (ABC Company)

Project	#TPR[1]	#SPR[2]	#RPR[3]	Project	#TPR	#SPR	#RPR
CS	4,560	3,370	5	SR	1,927	1,771	2
ARM	969	867	2	NB	828	731	2
SM	1,291	1,199	2	VBC	1,050	906	2
VW	787	768	1	AA	1,313	1,159	2
MS	1,156	1,092	2	ST	1,397	1,218	2

[1]Total pull requests, [2]Selected pull requests, [3]Reviewers per request

Table 5: Experimental Dataset (Open Source)

Project	Lang.[1]	#PR[2]	#PRR[3]	Project	Lang.	#PR	#PRR
Beets	Python	476	44	St2	Python	548	14
Orientdb	Java	283	22	Okhttp	Java	650	49
Rubocop	Ruby	860	86	Vagrant	Ruby	1,217	546

[1] Programming language [2]Total pull requests, [3]Total pull request reviewers

4.3 CORRECT's Architecture

We adopt a client-server architecture in the implementation of our code reviewer recommendation technique. It has two parts–*CORRECT server* and *CORRECT client*. The server is hosted as a web service, and it has API access to ABC code repositories at GitHub. The client is implemented as a Google Chrome plug-in, and it can request the server for recommendation. Once the client plug-in captures necessary information (e.g., details of the branch to be merged) from a pull request to be submitted, it encodes the information and sends to the server using an AJAX call. The server analyzes the recommendation request, executes the proposed recommendation algorithm (Algorithm 1), and then returns a ranked list of code reviewers to the client. Both client and server modules are available online for replication or third party use [2].

5. EXPERIMENT

One of the most effective ways for evaluating a code reviewer recommendation technique is to consult with actual code reviews and the reviewers assigned for them from a codebase. We evaluate our technique using the real code reviews data from ABC codebase. In particular, we use 13,081 pull requests and their code review details from ABC Company as our oracle in evaluating CORRECT against a number of popular performance metrics. In order to further validate our findings and demonstrate its superiority, we experiment with six open source systems of three different programming languages, and compare with the state-of-the-art technique. We particularly answer the following research questions through our conducted experiments:

- **RQ_1:** How does our technique–CORRECT perform in terms of the state of the art performance metrics?
- **RQ_2:** Does CORRECT outperform the state of the art technique for reviewer recommendation?
- **RQ_3:** Does CORRECT perform equally on both private and public codebase?
- **RQ_4:** Does CORRECT show bias to any of the development frameworks?

Table 6: Experimental Results with Individual Subject Systems (ABC Company)

	CS	ARM	SM	VW	MS	SR	NB	VBC	AA	ST	Avg.
Top-K Accuracy	93.92%	96.31%	96.75%	97.92%	94.51%	97.85%	68.81%	88.52%	89.82%	97.13%	**92.15%**
Mean Reciprocal Rank (MRR)	0.73	0.66	0.63	0.89	0.65	0.71	0.47	0.57	0.62	0.74	**0.67**
Mean Precision (MP)	75.01%	91.90%	90.06%	97.79%	86.29%	93.66%	65.27%	82.81%	82.53%	93.94%	**85.93%**
Mean Recall (MR)	65.56%	87.85%	85.80%	96.79%	83.25%	89.14%	60.71%	78.22%	75.94%	90.64%	**81.39%**

Table 7: External Library Similarity & Specialized Technology Similarity

	Library Similarity			Technology Similarity			Combined Similarity		
	Top-1	**Top-3**	**Top-5**	**Top-1**	**Top-3**	**Top-5**	**Top-1**	**Top-3**	**Top-5**
Top-K Accuracy	51.10%	83.57%	92.02%	46.55%	82.18%	91.83%	49.58%	83.75%	92.15%
MRR	0.51	0.66	0.67	0.47	0.62	0.64	0.50	0.65	0.67
MP	51.10%	65.93%	85.28%	46.55%	62.99%	83.93%	49.58%	65.98%	85.93%
MR	24.10%	58.34%	80.77%	21.99%	55.77%	79.50%	23.29%	58.43%	81.39%

5.1 Experimental Dataset

We use 10 projects (Table 4) from ABC codebase for our experiments. It should be noted that these projects were also selected for the exploratory study (Section 3). Since they met certain important constraints (i.e., details in Section 3.1) for that study, they are also suitable subject systems for our evaluation. It should also be noted that the experiments involve actual recommendation of code reviewers, and choice of the same systems does not have any impacts on the evaluation since there is no direct relation with the exploratory study. Rather, choosing the same systems confirms the findings of the exploratory study and vice versa. The selected systems are based on Google cloud platform, and they focus on different business functions such as business reputation management, social marketing, sales and brand analytics. Each of these projects is chosen carefully, and project related data are collected using Git Bash and GitHub API. We apply a sliding window approach [15] (i.e., window size=30) in selecting past pull requests from the request history to collect candidate reviewers for each current request. In *ABC Company*, one generally assigns code reviewers for a pull request by referring one or more peers in the message body of the request (e.g., @smelnyk-va). We collect such developer references (i.e., recommended reviewers), and make a *reviewer set* for each pull request. Besides, we also identify the developers who posted feedback against the request (i.e., actual reviewers) from the comment history of the request (Fig. 1), and they are appended to the reviewer set. We then use such set as the *gold reviewer set* for each of the pull requests in our experiments.

We also select six open source projects for our experiment written in three different programming languages–*Python, Java* and *Ruby*. Each project had at least 275 closed pull requests, and the corresponding gold reviewer set was developed by following the similar steps above. Table 5 shows the summary statistics of the selected open source projects.

5.2 Performance Metrics

Since our technique focuses on recommendation, we choose two relevant performance metrics for evaluation from the corresponding literature [5, 17, 22]. We also choose two metrics from information retrieval domain due to the inclination of this technique to this domain.

Top-K Accuracy: It refers to the percentage of the pull requests for which at least one reviewer is correctly recommended within the Top-K results by a recommendation technique. Top-K Accuracy can be defined as follows:

$$Top{-}KAccuracy(R) = \frac{\sum_{pr \in R} isCorrect(pr, Top{-}K)}{|R|}\%$$

Here, $isCorrect(pr, Top{-}K)$ returns a value 1 if there exists at least one reviewer from the gold set in the $Top{-}K$ results, and returns 0 otherwise. R denotes the set of all pull requests. The higher the accuracy, the better the technique.

Mean Reciprocal Rank (MRR): Reciprocal rank (RR) refers to the multiplicative inverse of the rank of the first correct result in the ranked list by a recommendation technique. Mean Reciprocal Rank (MRR) averages such measures for all pull requests. It can be defined as follows:

$$MRR(R) = \frac{1}{|R|} \sum_{pr \in R} \frac{1}{rank(pr)}$$

Here, $rank(pr)$ returns the rank of the first correct answer from a ranked list. MRR can take a maximum value of 1. The higher the MRR value, the better the technique.

Mean Precision (MP): It refers to the percentage of code reviewers which are correctly recommended for a pull request by a technique. Mean Precision (MP) averages such measures for all requests from the dataset.

Mean Recall (MR): It refers to the percentage of gold set reviewers which are correctly recommended for a pull request by a recommendation technique. Mean Recall (MR) averages such measures for all requests from the dataset.

5.3 Evaluation with ABC Systems

We evaluate our technique using a collection of 13,081 pull requests from 10 subject systems and four state of the art performance metrics as described in Section 5.2. We apply a sliding window based selection (window size=30) of past pull requests from the request history for each current request, and collect the code reviewer candidates. Then the candidates are ranked by our technique based on their experience both on the external libraries and the specialized technologies used in the current request. We then compare the ranked reviewer list with corresponding *gold reviewer set* for each of the requests. Table 6 and Table 7 summarize the performance details of our technique. In this section, we discuss our evaluation results, and answer RQ_1.

Table 6 shows the performance of our technique for each of the individual subject systems. In this case, only top five reviewers are considered, and we note that the technique provides a recommendation accuracy around 90% or above for all the systems except *NB*. This suggests a Top-K Accuracy of 92.15% for our technique which is highly promising according to relevant literature [5, 17, 19]. Our technique also provides a Mean Reciprocal Rank of 0.67 which is quite high [5]. More interestingly, on average, the recommendation technique returns results with 85.93% precision and 81.39% recall which suggests its greater potential for recommendation. Thus, our technique performs significantly well

Table 8: Comparison between CORRECT and RevFinder using all Systems (ABC Company)

Technique	Metric	Top-1	Top-3	Top-5
RevFinder [17]	Top-K Accuracy	54.48%	76.29%	80.72%
	MRR	0.54	0.64	0.65
	MP	54.48%	64.42%	77.24%
	MR	25.83%	57.17%	73.27%
CORRECT	Top-K Accuracy	49.58%	83.75%	**92.15%**
	MRR	0.50	0.65	**0.67**
	MP	49.58%	65.98%	**85.93%**
	MR	23.29%	58.43%	**81.39%**

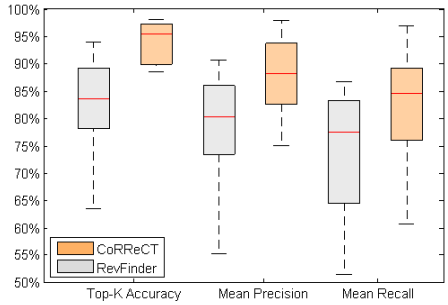

Figure 8: Comparison between CORRECT and RevFinder using Box plots (ABC systems)

in terms of all four state-of-the-art performance metrics, and the findings clearly answer our first research question, $\mathbf{RQ_1}$.

Table 7 shows the performance of our technique for three different cases involving the use of similarity metrics such as *external library similarity* and *specialized technology similarity*. The goal is to demonstrate the effectiveness of those metrics as a proxy for relevance between two pull requests which provides the basis for our reviewer recommendation. We consider Top-1, Top-3 and Top-5 reviewers recommended by our technique and determine the performance for each case. From Table 7, we note that the technique performs almost identically except for the Top-1 case when both metrics are considered in isolation. It provides about 92% Top-K Accuracy and above 80% precision and recall for both metrics with Top-5 results considered. This suggests that both similarity metrics are effective proxies for the relevance between two pull requests which drives our recommendation. Our recommendation algorithm (Line 11 to Line 17, Algorithm 1) exploits such similarities for estimating developer's expertise for code review which includes cross-project experience (i.e., library experience) and specific technology experience. Combing both metrics marginally improves the performance of our technique which also justifies the combination.

Our technique leverages recency of pull requests largely for code reviewer recommendation, and we also investigate its rationale using experiments. We conduct the experiments (1) using recent 30 pull requests and (2) using all available pull requests. CORRECT provides 17%-20% more accuracy for the first setting with better precision, better recall and better reciprocal rank, and thus, our choice of recent pull requests for historical learning is possibly also justified.

5.4 Comparison with Existing Techniques

In order to further validate the performance of our technique, we compare with– RevFinder [17], the state-of-the-art technique for code reviewer recommendation which outperformed earlier techniques as shown in their experiments. It considers File Path Similarity [16] for identifying relevant reviews first and then the code reviewers. We collect authors' implementation of the competing technique, and evaluate its performance on our experimental dataset (Table 4). Table 8, Fig. 8 and Fig. 9 summarize the comparative analyses between our technique and RevFinder.

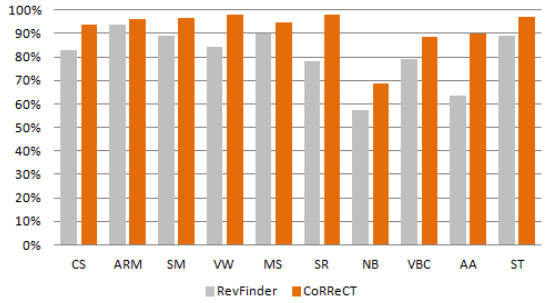

Figure 9: Comparison between CORRECT and RevFinder using individual systems (ABC)

Table 8 shows performance of the two recommendation techniques for Top-1, Top-3 and Top-5 results considered. We see that our technique outperforms the competing technique in terms for all four performance metrics for Top-3 and Top-5 cases. For example, RevFinder provides a maximum of 80.72% recommendation accuracy with 77.24% precision and 73.27% recall which are comparable to the authors reported performance (i.e., 79% Top-K Accuracy). On the other hand, our technique– CORRECT provides a 92.15% accuracy with 85.93% precision and 81.39% recall which are significantly higher. RevFinder performs relatively better only when Top-1 result (Table 8) is considered. However, Top-1 recommendation is rarely used in practice. Rather, more than one code reviewers are generally recommended [5, 19]. RevFinder determines the relevance of pull requests based on merely directory structures of the changed files, which might not be always effective as suggested by our findings. In contrast, our technique defines such relevance based on external software library or specialized technology similarity, a semantic level similarity, and the empirical findings report the potential of our technique.

Fig. 8 provides further insights on the performance of the two techniques using box plots. We collect accuracy, precision and recall of the techniques for each of the subject systems (Table 4), and derive five-point statistics for those performance metrics. We see that RevFinder provides a median accuracy around 85%, a median precision of 80% and a median recall between 75% to 80%. On the other hand, our technique provides a median accuracy over 95%, a median precision about 90% and a median recall about 85%. Besides, the extreme limits (i.e., maximum, minimum) are also higher for our technique than the counterparts.

Fig. 9 further validates our technique for individual systems. We see that CORRECT outperforms RevFinder in terms of Top-K Accuracy for each of the subject systems when Top-5 recommended reviewers are considered. We perform *Mann-Whitney U (MWU) test* [3] and *Cohen's d test* [1] on the accuracy measures for checking significance and effect size respectively. We found that the accuracy of our technique is significantly (p-value=0.003) higher than that of RevFinder. The second test returns $Cohen's\ d > 1.0$ and $Glass\ \triangle = 0.961$ which suggest that the effect size is large, i.e., the recommendation accuracy of our technique is largely higher than that of RevFinder.

Thus, each of the analyses above shows that our technique outperforms the state of the art–RevFinder which was found

Table 9: Comparison using Open Source Systems

Technique	Metric	Top-1	Top-3	Top-5
RevFinder [17]	Top-K Accuracy	48.89%	60.87%	62.90%
	MRR	0.49	0.54	0.55
	MP	48.89%	59.67%	62.57%
	MR	41.80%	55.86%	58.63%
CORRECT	Top-K Accuracy	57.72%	81.01%	**85.20%**
	MRR	0.58	0.68	**0.69**
	MP	57.72%	79.20%	**84.76%**
	MR	48.87%	73.44%	**78.73%**

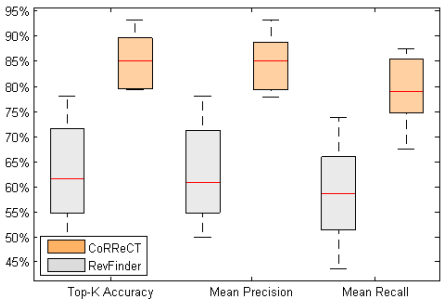

Figure 10: Comparison using open source systems

to be superior to earlier techniques [17]. This clearly answers **RQ$_2$**. Our experimental results suggest that library or technology information is probably more effective than source file path [17] for pull request relevance which probably led to our better performance.

One might wonder why we did not compare with another relevant technique– ReviewBot [5] from the literature that exploits line change history of source code for reviewer recommendation. We made that choice due to two appealing reasons. First, RevFinder outperforms ReviewBot by a large margin. Second, 70%–90% of the source code lines in the project are generally changed only once [17]. Thus, there is an inherent lack of sufficient line-level history, and therefore, the performance of ReviewBot is limited.

5.5 Experiments with Open Source Projects

Our recommendation technique– CORRECT is found highly promising with an organizational codebase which is closed source and based on *Python*. In order to further validate and generalize our findings, we conduct an experiment with six open source projects from GitHub written in three different programming languages.

Table 9 shows performance details of our technique with the open source projects, and compares with the state of the art technique– RevFinder. We see that RevFinder provides a maximum recommendation accuracy of 62.90% with 62.57% precision and 58.63% recall whereas our technique– CORRECT has 85.20% accuracy with 84.76% precision and 78.73% recall. The remaining metric–MRR is also higher for our technique. From the box plot in Fig. 10, we also see that the accuracy, precision and recall of CORRECT are significantly higher. Thus, our technique outperforms the state-of-the-art technique for open source projects as well.

Table 10 shows performance of our technique for each of the individual open source projects. We compare the performance for open source projects with that of closed source projects (Table 6) using MWU test and Cohen's d test. Although Top-K accuracy is slightly higher for commercial projects, the remaining metrics– precision, recall and reciprocal rank are comparable. For example, in the cases of precision and recall, we got p-value=0.239, $Cohen's\ d = 0.142$, $Glass\ \triangle = 0.190$ and p-value=.209, $Cohen's\ d = .276$, $Glass\ \triangle = 0.357$ respectively which suggest that such performance measures by our technique for both project types– *open source* and *closed source* are not statistically different. Thus, the findings answer **RQ$_3$**, i.e., CORRECT performs almost equally for both private and public codebases.

Table 10: Performance of CORRECT on Different Programming Languages (Open Source Systems)

	Python		Java		Ruby	
	Beets	**St2**	**Okhttp**	**Orientdb**	**Rubocop**	**Vagrant**
TKA[1]	93.06%	79.20%	88.77%	81.27%	89.53%	79.38%
MRR	0.82	0.49	0.61	0.76	0.76	0.71
MP	93.06%	77.85%	88.69%	81.27%	88.49%	79.17%
MR	87.36%	74.54%	85.33%	76.27%	81.49%	67.36%

[1]Top-K Accuracy

Table 10 also demonstrates how our reviewer recommendation technique performs with projects using various programming languages– *Python, Java* and *Ruby*. We did not notice any particular bias to any of the languages, and our technique provides nearly 90% accuracy with at least one project from each of the languages. The other performance metrics are also promising and competitive with those with *ABC* projects. Thus, the findings also answer **RQ$_4$** which suggests that our recommendation technique is not biased to any particular programming language (e.g., *Python*).

6. THREATS TO VALIDITY

Threats to internal validity relate to experimental errors and biases [23]. Many of our subject systems (except CS) are medium sized, and they contain about 1.1K pull requests on average. However, we consider not only more (i.e., 10 commercial and 6 open source) systems than any of the existing studies [5, 17] but also conduct experiments with 17K pull requests. Besides, we performed *stress testing* using 3.3K pull requests from a large open source system–*Mozilla Zamboni* to deliberately make our technique failed. However, our technique passed the test, and did not crash. Thus, the technique is robust enough, and the dataset might be also sufficient enough to generalize our findings.

Threats to external validity relate to the generalizablity of a technique. We first experimented using only Python projects from a private codebase which might raise the generalizability concern. In order to handle the threat, we adapt our technique for two more platforms– Java and Ruby. From our findings with 16 (12 python, 2 Java and 2 Ruby) subject systems from three different languages, we didn't notice any bias of our technique to any project type (i.e., open source, closed source) or any language (i.e., Python, Java, Ruby).

Threats to construct validity relate to suitability of evaluation metrics [23]. We use Top-K Accuracy and Reciprocal Rank which are widely used by relevant literature [5, 17, 22]. The remaining two metrics are well known in information retrieval, and our technique is also aligned with this domain. This confirms no or little threats to construct validity.

7. RELATED WORK

Code Reviewer Recommendation: Existing studies recommend code reviewers by analyzing code review history– line change history [5] and past review comments [19, 22], project directory structure [16, 17], and developer collaboration network [22]. Balachandran [5] propose a recommendation technique–ReviewBot that analyzes change history of

the affected lines in a review request. However, existing findings show that most of the lines are generally changed only once [17] which makes the line change history really scarce and thus, the performance of ReviewBot is limited. Thongtanunam et al. propose another technique– RevFinder [17] that identifies relevant review requests using File Path Similarity (FPS) [16], and then recommends reviewers from those requests for a review request at hand. RevFinder also outperformed existing techniques including ReviewBot [17]. On the other hand, CORRECT identifies relevant pull requests using *external library similarity* and *specialized technology similarity* which are found to be more effective than File Path Similarity[17] for estimating relevance between pull requests, and thus for reviewer recommendation. In our comparative studies (Section 5.4), we show that our technique outperformed RevFinder with statistically significant performance improvements. Another recent work [19] applied machine learning on past review comments and File Path Similarity [17]. It thus suffers from similar issues as of RevFinder such as pull request relevance issue, and that the learned models could be biased to the subject systems under study.

The remaining technique–Yu et al. [22] analyzes past review comments and developer collaboration network for reviewer recommendation. While we use library and technology similarity between pull requests for determining relevant past requests, they use review comment similarity (i.e., textual similarity) for the same purpose. Besides, their idea is still not properly evaluated or validated.

Expert Recommendation: Kintab et al. [12] propose an expert recommendation system that exploits *code similarity* for estimating expertise of a developer on a code fragment of interest. Similar technique is applied by da Trindade et al. [8] where they develop a communication network among documents, source code and developers, and recommend dominant developers as experts. Yang [20] studies the developer network using code review relationship, and identify core and peripheral developers using different network properties. There exist several studies in the domain of *bug triaging* that analyze duplicate bug reports [11] or apply IR-based traceability [13] techniques for recommending experts for bug fixation. Several studies are also conducted on expert user recommendation at Stack Overflow that analyze cross-domain contributions [18] or question difficulty [10] for expertise estimation. While these expert recommendation techniques are somewhat similar to ours, their context of recommendation is different and thus, comparing ours with them is not feasible. Of course, we introduced two novel and effective expertise paradigms (*cross-project experience* and *specialized technology experience*) which were not exploited by any of the recommendation systems. This makes our technique significantly different from all of them.

8. CONCLUSION & FUTURE WORK

To summarize, we propose a novel technique– CORRECT for code reviewer recommendation for pull requests at GitHub. It heuristically captures the experience of a developer with the external libraries (i.e., cross-project experience) and specialized technologies used in a pull request for reviewer recommendation. Experiments using 13,081 pull requests from 10 subject systems from an organizational code repository show that our technique recommends code reviewers with 92.15% Top-5 accuracy, 85.93% precision and 81.39% recall which are highly promising. Experiments using 4,034 pull requests from six open source projects suggest that our technique performs well both for open source and closed source projects, and is not biased towards any programming languages. Comparison with one of the state of the art techniques also demonstrates the superiority of our technique. In future, we plan to work on handling more concurrent recommendation requests as suggested by ABC developers.

References

[1] Cohen's d Test. URL https://researchrundowns.wordpress.com/quantitative-methods/effect-size.

[2] CoRReCT Portal. URL http://www.usask.ca/~mor543/correct.

[3] Mann-Whitney U Test. URL http://www.socscistatistics.com/tests/mannwhitney.

[4] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proc. ICSE*, pages 712–721, 2013.

[5] V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proc. ICSE*, pages 931–940, 2013.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern Code Reviews in Open-Source Projects: Which Problems do they Fix? In *Proc. MSR*, pages 202–211, 2014.

[7] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *Proc. FSE*, pages 257–268, 2014.

[8] C.C. da Trindade, Y.A.M. Barbosa, A.K.O. Moraes, J.O. de Albuquerque, and S.R.L. Meira. An Expert Recommender System to Distributed Software Development: Requirements, Project and Preliminary Results. In *Proc. SBSC*, pages 161–168, 2009.

[9] G. Gousios, M. Pinzger, and A. v. Deursen. An Exploratory Study of the Pull-based Software Development Model. In *Proc. ICSE*, pages 345–355, 2014.

[10] B. V. Hanrahan, G. Convertino, and L. Nelson. Modeling Problem Difficulty and Expertise in Stackoverflow. In *Proc. CSCW*, pages 91–94, 2012.

[11] K. Kevic, S.C. Muller, T. Fritz, and H.C. Gall. Collaborative Bug Triaging using Textual Similarities and Change Set Analysis. In *Proc. CHASE*, pages 17–24, 2013.

[12] G. Kintab, C. K. Roy, and G. Mccalla. Recommending Software Experts using Code Similarity and Social Heuristics. In *Proc. CASCON*, page to appear, 2014.

[13] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging Incoming Change Requests: Bug or Commit History, or Code Authorship? In *Proc. ICSM*, pages 451–460, 2012.

[14] P.C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D.M. German. Contemporary Peer Review in Action: Lessons from Open Source Development. *TSE*, 29(6):56–61, 2012.

[15] Y. Shi. An Improved Collaborative Filtering Recommendation Method based on Timestamp. In *Proc. ICACT*, pages 784–788, 2014.

[16] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida. Improving Code Review Effectiveness Through Reviewer Recommendations. In *Proc. CHASE*, pages 119–122, 2014.

[17] P. Thongtanunam, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who Should Review my Code ? In *Proc. SANER*, pages 141–150, 2015.

[18] R.l Venkataramani, A.l Gupta, A. Asadullah, B. Muddu, and V. Bhat. Discovery of Technical Expertise from Open Source Code Repositories. In *Proc. WWW*, pages 97–98, 2013.

[19] X. Xia, D. Lo, X. Wang, and Xiaohu Y. Who Should Review this Change? Putting Text and File Location Analyses Together for More Accurate Recommendations. In *Proc. ICSME*, 2015.

[20] Xin Yang. Social Network Analysis in Open Source Software Peer Review. In *Proc. FSE*, pages 820–822, 2014.

[21] Y. Ye, K. Nakakoji, and Y. Yamamoto. Reducing the Cost of Communication and Coordination in Distributed Software Development. In *Proc. SEAFOOD*, volume 4716 of *Lecture Notes in Computer Science*, pages 152–169. 2007.

[22] Y. Yu, H. Wang, G. Yin, and C. Ling. Reviewer Recommender of Pull-Requests in GitHub. In *Proc. ICSME*, pages 609–612, 2014.

[23] T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-Specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.