S	
E1	
E2	
For Office Use Only	

Masters Project Final Report

(MCS)

2020

Project Title	Code Reviewer Recommendation system for pull requests
Student Name	Pavani Yashodha De Silva
Registration No. & Index No.	2017/MCS/021 17440216
Supervisor's Name	Dr. Ajantha Athukorale

For Office Use ONLY

Code Reviewer Recommendation system for pull requests

A dissertation submitted for the Degree of Master of Computer Science

Pavani Yashodha De Silva
University of Colombo School of Computing
2020

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: Pavani Yashodha De Silva

Registration Number: 2017/MCS/021

Index Number: 17440216

____

Signature: Date: 21/6/2020

This is to certify that this thesis is based on the work of

Mr./Ms.

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name:

____

Signature: Date:

Abstract

Development of software has been drastically changed in the direction of distributed and collaborative environment. Global contributors are encouraged to remotely contribute to open source projects using the pull based model and continuous integration techniques with extremely low barriers. Since this allows external developers to integrate changes into the central repository, maintaining the code quality of the central code base is considered a critical project activity with high importance. Core developers of a project is responsible for maintaining the central code base. Performing a code review before integrating the changes by external developers improves software quality. Identifying the most apposite reviewers for a pull request review is a challenging activity in a distributed software development environment. Identifying the potential candidate reviewer would improve the reviewing latency and will help to provide constructive feedback on the development. This research is an approach to recommend potential reviewer candidates for a pull request. A similarity measure between the novel pull requests and the available pull requests of the repository based on tile and description similarity (text similarity), file path similarity and activeness of the integrators is used as the basis for the approach. Upon analyzing on the literature, it was revealed that activeness of the integrators is not considered for recommendation of reviewers in state-of-the-art approaches. Most research is based on one factor either on text similarity, file path similarity, expertise of developers or social network and relationships of developers. This approach is a combination of multiple factors and uniquely considers activeness of the integrators towards development of the recommendation algorithm. On submission of a novel pull request, an average integrator score is calculated for each of the integrators of the repository by the similarity between the novel pull request and the older PR's he/she has reviewed. Ranking of the integrators based he integrator score is used for generation of the recommendation list for reviewers. Feature weighting is done, and the accuracy is compared against different weighting combinations. Experimentation is done based on three github repositories - Akka, Bitcoin and Rubocop which are developed on different programming languages. This approach yields and average accuracy of 82% across multiple repositories.

Acknowledgements

I would like to express my sincere gratitude towards my supervisor Dr. Ajantha Athukorale, Deputy Director – University of Colombo School of Computing (UCSC) for his motivation, guidance and support extended throughout the research project.

Appreciation and thanks extend to all the lecturers of **University of Colombo School of Computing (UCSC)** who enlightened me in numerous ways to solve the complex issues we encountered from the initiation of the project and for their encouragement, insightful comments, and hard questions.

I am also grateful to the non-academic staff members and technical staff at the faculty for providing me with the technical requirements needed to achieve ultimate project goal. Without your support this project would not have been successful.

I warmly extend my heartfelt gratitude all friends who helped me in numerous ways throughout this venture to complete the research project.

Finally, I would like to express gratitude to all my family members for their continuous support in this journey.

Thank you all.

Table of Contents

List of Figures

List of Tables

Chapter 1

1.Introduction

1.1 Introduction

Software development has been drastically changed in the direction of distributed development in a collaborative environment recently. Pull based model used in continuous integration process inspires global contributors to contribute to projects with extreme low barriers. This model allows external contributors to propose and integrate changes into a project without having direct access to the central codebase of the project. External developers usually create a fork, work on changes of their interests and when the changes are ready to be merged, request the changed files to be merged into the central project using a pull request [1].

Pull requests management has been identified as a critical project activity. Software quality is among the foremost considerations of software engineering. Projects core team is responsible for maintaining the code quality. Numerous techniques have been adopted to ensure code quality. Among industry accepted standards, peer code reviews have been identified as an effective technique for maintaining code quality. Code review is usually considered as a cost-effective fault detection approach as bugs can be detected early which is when it is less expensive to fix [2]. Projects core team must ensure that pull requests are sensibly reviewed and evaluated before merging a pull request to the main project. Reviewing is a scrutiny of change of code by co-developers to detect and fix defects prior to merging a change of code to a system [3]. Review process helps to identify coding rule violations, simple logical verifications and identify vulnerabilities at early phase of development.

Generally, after a code change a merge request with the changed code lines is submitted for review, a set of code- reviewers will be invited by the author to review the code change. Subsequently fixes will be suggested by the reviewers discussing on the change. Currently a pull request is reviewed by one or more professional developers of the central codebase suggested by the merge request submitter.

1.2 Background and Motivation

With the drastic improvement in distributed development, pull request-based model has been popular. External developers can suggest changes and contribute to a project short of direct access to the central code base of the project. An external person creates a fork on the original repository and study the code base of his area of interest. He implements novel features or fix bugs on his local repository cloned from the latest version of the central code repository. Once a developer has completed his new features or bug fixes, he creates a pull requests to merge his developments to the central repository. Usually the developer himself, or one of the core developers of the project assigns a reviewer to the pull request manually. The reviewer is responsible for discussing the submitted new features and bug fixes with the developer and other reviewers and suggesting updates. Next the contributor updates the merge request upon reviewers' suggestions. Afterwards reviewers discuss the pull request with the updates again. The core team after considering all the opinions of reviewers in conclusion, merges or rejects the pull request. Pull request mechanism used is collaborative and distributed.

Currently in the pull request-based models, a pull request is assigned a reviewer manually by

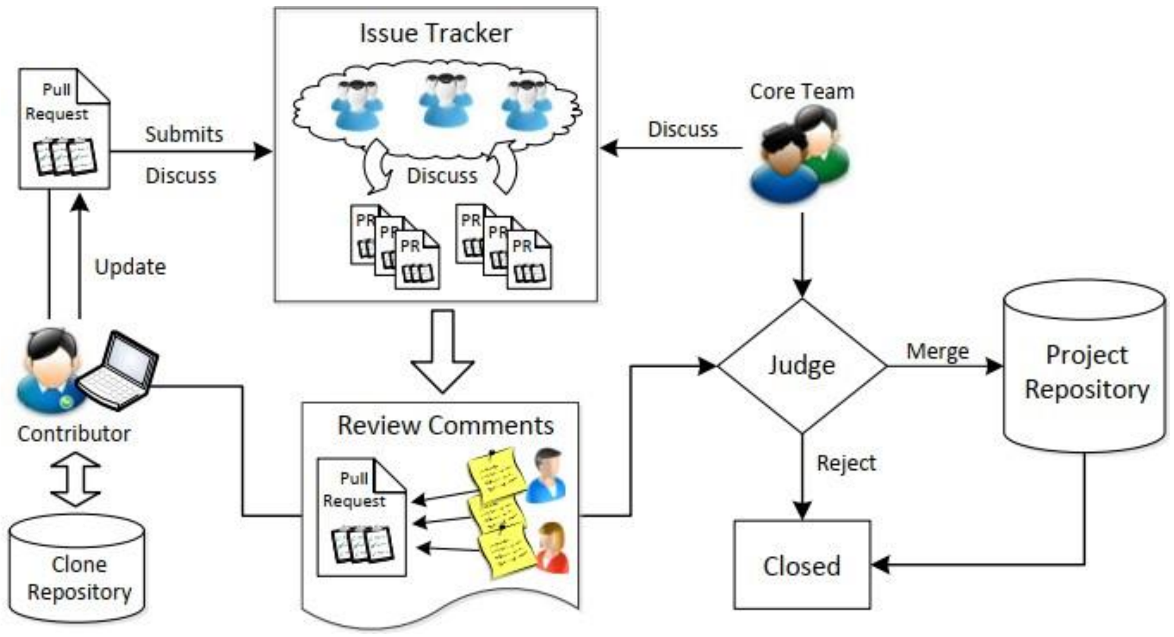


Figure 1: Outline of pull request mechanism

the submitter or by one of the main developers of the project. Assignee is responsible for the review process. The assignee will receive notifications and other developers can participate for discussions using the @mention tag. The developers with @mention tag will receive notifications on the pull request. Two types of comments can be published by the reviewers: general comments on wide-ranging contributions, and code-inline comments for the changes on specific lines.

It is observed that, a change of code which is not immediately reviewed is almost likely not to be reviewed. Weigerber et al. [4] identified that minor changes of code are preferred to be accepted by reviewers unlike heavy code changes. Some code changes are idling in the queue for more than two weeks prior to being reviewed according to Weigerber et al. Rigby and Bird

[5] spot that reviewing time of 50% of reviews average to 30 days. According to Tsay et al. [6] some code changes are waiting for 2 months to be merged to the central repository. Idling happens mostly due to not assigning the targeted or most suitable code reviewer for the merge request or assigning reviewers who are not active in the codebase. By recommending the potential targeted reviewers for a PR idling time could be minimized.

1.3 Problem Domain

Code reviews are expensive as it entails the reviewers to read, comprehend and analysis a code change [7]. Reviewers with profound knowledge of the relevant system code is essential to find the defects of the submitted code change. Determining potential code reviewers for every code change is labor intensive and time consuming for developers as all the code changes must be inspected and reviewed prior integration [8]. Speedy and precise selection of reviewers is the crucial factor of the success of code reviewing process. Reliable information on reviewer expertise is not available readily. Therefor mining of the codebase is necessary to identify the expertise of the developers. Novice developers with less familiarity with the codebase and the skills and expertise of hundreds of developers struggle in reviewer identification in a distributed software development environment. This directly impacts the reviewing time badly.

Discussions among reviewers and timely responsiveness of reviewers affect the pull request evaluation process immensely. Most of the developers criticize on late feedback on their changes and irresponsiveness of the reviewers due to not giving a timely feedback. Commits lacking an assigned reviewer require considerably high time to be integrated to the master repository [9].

With the recent trend of open source development, number of pull requests also increase drastically. Bulk of pull requests is a critical challenge to the integrators in large projects. Consequently, some reviewers can be exhausted due to their expertise on a single aspect of a project. Timely noticing of pull requests by some potential reviewers will not happen.

Manual suggestion of reviewers would lead to communication overhead and delay in processing of pull requests. Thus, suggestion of potential reviewers automatically to a novel pull requests will enhance the efficiency of the pull-based model. Recommending reviewer will increase the effectiveness of the review process because it lowers the time gap in between the submission of a merge-request and the review of it.

1.4 Research contribution

1.4.1 Aim and Objectives

Objectives of the project are aligned with the goal of the research – Implementing an automated pull request reviewer suggestion system.

Objectives are

- Investigation on the potential parameters for recommending apposite reviewers for a pull request.

  There are many attributes that can be considered in recommending the potential reviewers for a pull request. For example

  - ✓ Activeness
  - ✓ Expertise of technologies
  - ✓ File path similarity of recently reviewed pull requests and current changes
  - ✓ Text similarity of recently reviewed pull requests and current changes

  This objective includes studying the literature and similar approaches which has solved the above problem and identifying the most suitable attributes to be used for developing a model to recommend reviewers. Each of the attributes contribute differently to the output of the system.

- Feature extraction and developing a dataset.

  After identifying the potential attributes, features to represent these attributes from github repositories needs to be identified. Thereafter measures need to be taken to extract the identified features and develop a dataset containing the identified attributes. A dataset containing features from multiple projects need to be developed. For example

  - ✓ Activeness can be measured through last commit date or time lapse between suggesting and reviewing the pull request
  - ✓ Expertise of technologies can be measured using the technologies used in pull requests reviewed in the past.

- Developing a system to recommend pull request reviewers.

  After extracting the above-mentioned features, a system is developed using the prior developed dataset. Appropriated weights need to be allocated for each of the features based on the importance of the features for the outcome.

- Evaluation of the developed model with respect to state-of-the-art techniques.
  Results of the model are compared with the state-of-the-art techniques to validate the performance of this model. Testing dataset developed using repositories of open source projects is used to test the results.

1.5 Scope and limitations

Research is about development of a code reviewer recommendation system for pull requests. On studying literature, I research about the factors which affects identifying a potential candidate for reviewing a pull request. This research investigates on the potential parameters which affects recommending apposite reviewers for a pull request by studying the literature and identifies the factors for developing the system. Feature extraction from the available github repositories to characterize the identified parameters is done. Thereafter identified features are extracted from repositories and a dataset containing the identified attributes is developed for multiple repositories.

The research identifies the features characterizing a pull request such as title, description, file paths of changed files, libraries and technologies used in changed files and develops a similarity between pull requests to recommend reviewers for a pull request reviewing process. Identifying the potential reviewers is done using similarity measures. A similarity algorithm considering file path similarity, text similarity and activeness of the integrators is developed. An average integrator score is assigned to each of the integrators in the repository once a novel pull request is submitted using the similarity measure algorithm. Integrators are ranked based on the score and the top k recommendation list for a pull request id generated. Experimentation of accuracy by assigning weights to the attributes is done to analyses the importance of attributes towards the reviewer recommendation

Research compares the results of the recommendation against the reviewers of the actual PR's and experimentation analyses how the impact of each factor could affect the final recommendation. Three github repositories of different programming languages – Akka, Bitcoin and Rubocop is used for experimentation.

Limitations of this research included not identifying expertise and libraries used by developers on cross project basis. The system is developed only based on a single project therefor cross project evaluation is not focused in this research. Furthermore, the social relations among the reviewers which helps to identify reviewers with the same expertise based on the discussions they have actively participated is not considered in this research.

1.6 Structure of the report

Introduction chapter of the report is structured to identify the research problem and motivation to develop the proposed system. Scope the proposed system and the limitations are also elaborated in this chapter.

Second chapter includes a literature review which detailly analyses the state-of-the-art techniques to solve the identified research problem. The methods, approaches, algorithms and implementation of the state-of-the-art techniques are detailly explained here. The research gap of the current approaches to solve the identified research problem is discussed here. Furthermore, comparison of the state-of-the-art techniques based on their limitations and advantages have been summarized in this chapter.

Third chapter is about a detailed description of the methodology I have adopted to solve the above identified research problem. High level data flow of the proposed system, high level architecture of the system and the approach and algorithms adopted to implement the solution is discussed in detail in this chapter.

Fourth chapter details about the evaluation techniques adopted to appraise the suggested approach against state-of-the-art techniques for automatic reviewer suggestion.

Final chapter discusses about the problems encountered during the research and how I have tried to address the identified problems. Furthermore, this chapter discusses on future research activities which could be based on this research and extensions for the research.

Chapter 2

2. Literature Review

2.1 Introduction

A merge request (pull request) is a checkup of a modification of source code by an independent external contributor to detect and fix defects prior to amalgamating a source code alteration into a system. Managing merge requests has been identified as a critical project activity. Projects core team is responsible for maintenance of the code quality. They must ensure that pull requests are sensibly reviewed and evaluated before merging a pull request to the main project. Review process helps to identify coding rule violations, simple logical verifications and identify vulnerabilities at early phase of development. Currently a pull request is reviewed by a single or multiple expert developer of the central codebase suggested by the pull request submitter. Reviewers with profound knowledge of the relevant system code is essential for the success of this task. Reliable information on reviewer expertise is not available readily. Therefor mining of the codebase is necessary to identify the expertise of the developers. Discussions among reviewers and timely responsiveness of reviewers affect the pull request evaluation process immensely. Most of the developers complain about not getting a timely feedback on their changes. Discovery of apposite code-reviewers to every piece of code transformation is labor- intensive and time-consuming.

With the recent trend of open source development, number of pull requests also increase drastically. Bulk pull requests is a critical challenge to the integrators in large projects. With the radical increase of the pull requests, need for the competent reviewers also increase. The challenge here is to find apposite reviewers in a pool of reviewers. Manual suggestion of reviewers would lead to communication overhead and delay in processing of pull requests. Review latency is the time gap between a pull request being submitted and the time the reviewers start discussing on the pull requests. Review latency of pull requests with a reviewer assignment is much low than pull requests without reviewer assignment. Average gain of a pull request with a reviewer suggested, submitting his first comment on the merge request is 40.8 hours shorter than that without recommendation [3].

Results reveal that code-reviewer assignment problem exists among 4%-30% of reviews. Pull requests with the reviewer assignment problem pointedly consume 12 days extended to approve a code change [3]. Thus, suggestion of potential reviewers automatically to novel pull requests will enhance the efficiency of the pull-based model.

2.2 Methods

Currently many approaches are used to identify potential candidates for recommending reviewers. Machine learning based approaches considering a pull request as a textual document, file path similarity-based approaches, models considering social relations among reviewers, approaches based on expertise of reviewers are among the state-of-the-art techniques.

2.2.1 Text based approaches

Y. Yu et al. base this approach on automatic bug triaging based on mining bug repositories using machine learning techniques [10], [11], [12]. Model considers pull requests as text documents and use machine learning techniques to predict top n reviewers. Each pull request is uniquely recognized using its title, description, and categorized with the names of developers who has at least submitted one comment to the request. After eliminating all non- alphabetic tokens and stop words, rest of the words are stemmed. An individual pull request is represented as a vector space model where an individual element in the vector is a term, and the importance of the pull request represented by the value. If a word frequency in a pull request is high, it is considered more important for the pull request. Vice versa if the same word appears n many pull request the importance of that word for categorizing a pull request is low. Value of a term is represented using Term frequency-inverse document frequency (tf-idf) shown in Equation 1.

$$tfidf(t, pr, P_R) = \log\left(\frac{n_r}{N_{p_r}} + 1\right) * \log\frac{N_{p_r}}{|p_r \in P_R : t \in p_r|} \tag{1}$$

t represents a term, pr notates a pull-request, PR is the corpus of all pull-requests associated with a given project, nt is the occurrences for term t in pr, and Npr represents the total number of terms in pr and total pull-requests in the corpus is notated by NPR [13].

In training machine learning classifiers, each merge request must be labelled with many key words therefor classifiers must possess the capability to handle multi-label classification. Reviewers are ranked on the probability and when the probabilities appear to be same developers are ranked in terms of the number of comments on pull requests on a project submitted by them. SVM classifier is used in this approach for classification.

2.2.2 File path and location-based approaches

P Thongtanunam et al. implements a system for recommending reviewers based on file path similarity algorithm (FPS). It determines the likeness of PR reviews based on the file path location of the changed files. The main assumption this study uses is that files that are in similar locality will be examined and reviewed by the same expertise and experienced developers. The motivation underlying the above assumption is the directory structure of Linux kernel where files with alike functions are generally located in similar or adjacent directories [14].

FPS algorithms selects the potential candidates for the reviews from the reviewers who had analyzed and reviewed files with similar file path locations. This also considers prioritization of time in recommending the top candidates. Inputs for the algorithm are the new request and the number of candidate reviewers to be recommended. And the output is an ordered list of potential reviewers based on file path similarity scores [14]. Algorithm 1 [14] is used by P Thongtanunam et al to find the potential candidate list.

Algorithm 1 Recommend Reviewers (Rn; k)

```
potentialReviewcandidates = list()
pastReviewList = retrievePastReviews(n)
j = 0
for newReview Rp :pastReviewList do
    candidateScore = FPS(Rn;Rp; j)
    for newReviewer i : getPastReviewers(Rp) do
        potentialReviewcandidates [i] = potentialReviewcandidates [i] + score
    end for
    j = j + 1 end for
potentialReviewcandidates:sort()
return potentialReviewcandidates[0 : k]
```

Equation 2 calculates the File Path Function. Past review score (Rp) is the average similarity of every file in Rp (fp) comparing with every file in Rn (fn). The set of file paths of the input review is returned by file path function. Equation 3 is the Similarity (fn; fp) function which computes the similarity between fp and fn. The averaged similarity score is prioritized by j and δ value where δ is a time prioritization factor. (0; 1).

$$FSP(R_n, R_p, m) = \frac{\sum_{\substack{f_n \in Files(R_{n,}) \\ f_p \in Files(p)}} Similarity\,(f_n, f_{p,})}{|\,Files(R_{n,})| * |\,Files(p)|} * \delta^m \tag{2}$$

$$Similarity\,(f_n, f_{p,}) = \frac{commonPath(f_n, f_{p,})}{\max\,(Length(f_n), Length(f_p))} \tag{3}$$

In Equation 4 the commonPath(f_n; f_p) function counts the common directory in both file paths in order. Equation 3 formularizes the count of commonPath(f_n; f_p) where the values of i and j are starting from 0.

$$commonPath(f_{n,}f_{p,}i,j) = \begin{cases} commonPath(f_{n,}f_{p,}i+1, j+1) \\ 0 & otherwise \end{cases} \tag{4}$$

Evaluation of the performance of FPS algorithms has been done based on three distributed Open Source Software (OSS) projects: Android Open Source Project (AOSP), OpenStack, and Qt. Performance for the algorithm is measured differently for recommending one top reviewer, top three reviewers and top five reviewers. 77.97% accuracy has been acquired by the FPS algorithm. This algorithm achieves 77.12% accuracy for AOSP project, 77.97% for OpenStack. However, for Qt which is comparatively large project only 27-36% accuracy has been achieved [14].

Limitation of this algorithms is the poor performance for large projects. Calculation assumed of similar files located in similar or adjacent locality. The file structure of Qt might not correspond with the prior assumption. Furthermore, as potential candidate reviewers were dependent on prior recommended reviewers list, reviewers may be frequently recommended and may be overloaded as workload balancing was not considered [14].

P. Thongtanunam et al. suggests a file location-based code reviewer recommendation approach as Revfinder which uses the resemblance of file paths formerly reviewed to endorse a potential reviewer based on the intuition files that are organized in alike file paths are managed and reviewed by equally skilled code-reviewers [3].

Revfinder uses Code Reviewers Raking Algorithm and combines the top reviewers to find the potential candidates for a novel PR. Main aim of this approach is to endorse reviewers who has priory reviewed almost identical functionality. Equivalence level of priory evaluated file paths is used by the Code-Reviewers Ranking Algorithm to calculate code reviewer scores. State-of-the-art string comparison techniques [15] has been used to calculate review similarity benchmarking scores.

Figure 2 illustrates how Code Reviewer ranking Algorithm calculated the similarity of file paths for a novel PR. As there are multiple string comparison techniques, Thongtanunam et al. finds the top reviewers using multiple string comparison techniques and collectives the diverse output lists into a combined list to minimize the false positives [3].

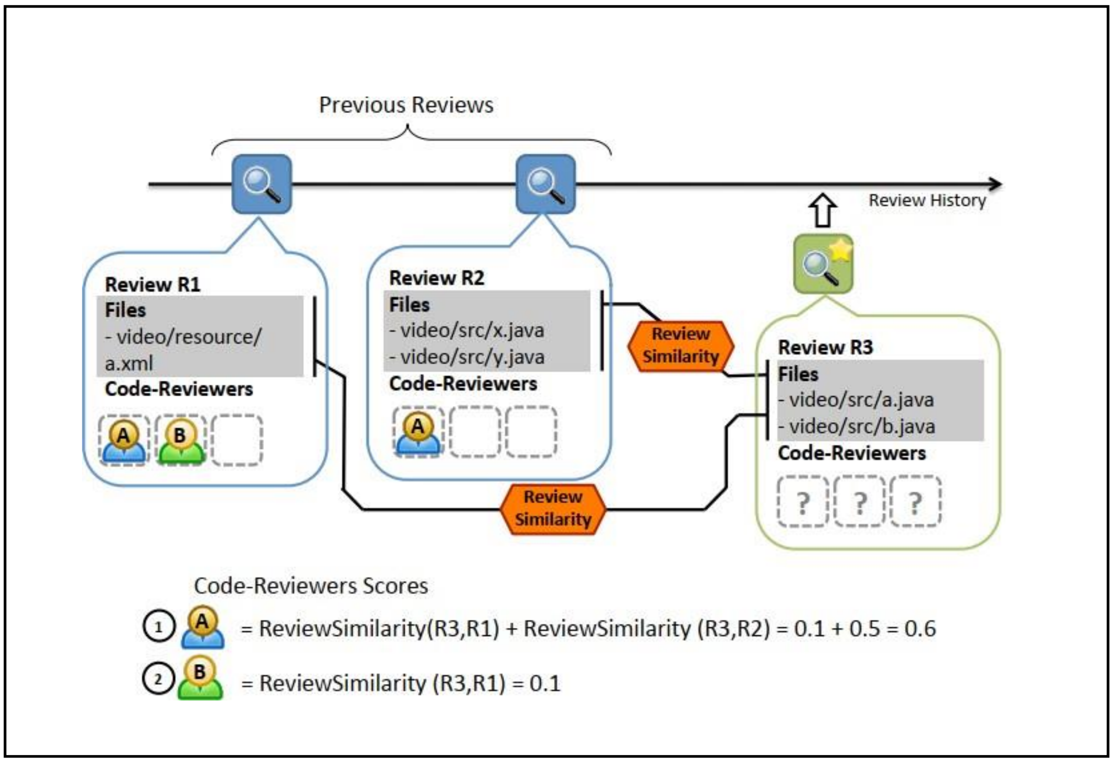


Figure 2: Computation example of Code Reviewer ranking Algorithm [3]

Algorithm 2 [3] is used by Thongtan et al. to find the potential candidate reviewer list by estimating review likeness score between prior reviews and the novel request (Rn). Thereafter resemblance scores are circulated to code reviewers. Review similarity score (calculatedScore R_p) of each past review (R_p), the is an average of file path similarity value of individual file path in R_n and R_p calculated using filePathSimilarity(fn, fp) function in Equation 5.

$$filePathSimilarity\ (f_n, f_p,) = \frac{StringComparison(f_n, f_p,)}{\max\ (Length(f_n), Length(f_p))} \tag{5}$$

File path is split using the delimiter slash and similarity of words is considered. The StringComparison(f_n, f_p) function returns the count of mutual components in both file paths after comparing the individual components of the file paths of f_n and f_p. String comparison techniques Longest Common Substring (LCSubstr), Longest Common Prefix (LCP), Longest Common Subsequence (LCSubseq) and Longest Common Suffix (LCS) are used to find sperate potential candidate lists.

Combination of the lists are used to improve the performance in this approach. Borda Count [16] has been used as the combination technique which is a voting technique that is used for combination of the recommendation lists grounded on the rank. A count is assigned to each

```
Algorithm 2 The Code-Reviewers Ranking Algorithm
Input: Rn : A novel code review
Output: O : Reviewer candidate list
Method:
formerClosedCodeReviews ← Priorly closed review list
formerClosedCodeReviews ← order(formerClosedCodeReviews).by(createdDate)
for singleReview R p ∈ formerClosedCodeReviews do
    File sn ← retrieveFiles(Rn)
    File sp ← retrieveFiles(Rp)
    # Calculate review similarity score between Rn and Rp
    computedSimilarityScore Rp ← 0
    for fn ∈ File sn do
        for fp ∈ File sp do
        calculatedScore Rp ← Score Rp + filePathSimilarity(fn, fp)
    end for
end for
calculatedScore Rp ← calculatedScore Rp / (length(File sn) × length(File sp))
# Proliferate calculated similarity scores of the reviews to code-reviewers who have a
history in reviewing a PR Rp
for Code-Reviewer r : retrieveCodeReviewers(Rp) do
C[r]. reviewerScore ← C[r]. reviewerScore + reviewerScore Rp
end for
end for
return C
```

code reviewer Ck based on its rank in each of the recommendation list using different string comparison techniques. Candidate who sums up to the highest count is the highest ranked reviewer Equation 6 is used to calculate the combination score where total count of candidate code- reviewers with a non-zero score in Ri is Mi and the rank of code-reviewer candidate ck in Ri is $(c_k|R_i)$.

$$Combination(c_k) = \sum_{R_i \in R} M_i - rank(c_k|R_i) \tag{6}$$

Evaluation of Revfinder is performed using 42,045 reviews on four open-source software systems namely Android Open Source Project (AOSP), OpenStack, Qt and LibreOffice. ReviewBot [2] is used as the baseline approach. Top-k accuracy and the Mean Reciprocal Rank (MRR) are used as the evaluation metrics. Top-k accuracy computes the percentage of accurately recommend code reviewers on a PR and the total number of reviews. A mean of reciprocal ranks of accurate code-reviewers recommended through a recommendation list is computed by Mean Reciprocal Rank (MRR). Top k accuracy and MRR of Revfinder approach is higher with respect to all projects. Thotunaamn et al. presents that Revfinder properly suggested 79% of reviews with a top-10 recommendation. Therefor Revfinder outperforms four times more precisely in comparison with ReviewBot [2].

Limitation of this approach is results were only based on four datasets. Experimental results would differ with multiple datasets. Furthermore, code reviewer retirement was not considered in this approach. That is the time sensitivity factor of the pull request reviewing process was not taken into consideration. Workload balance of the reviewers wan not considered as a metric and as a result reviewer might be burdened with workload [3].

Commercialized large software projects assign reviewers based on the file revision history and diff. It is difficult to do this when there are immensely large number of files. V. Balachandran propose a ReviewBot which uses an approach grounded on, line modification history of source code. The line alteration history of a line of code in a file difference of a PR is the list of PR s which was influenced by that line in the history. The 'line' here denotes the line in the patched file found as a result of applying a line in the raw diff data, conflicting line in the raw diff data [2].

Even though calculating file diff of deleted and altered lines are straightforward, inserted lines must be handled differently as they have no PR which deal with these lines in the history. Therefor for newly introduced lines, we use the adjacent prevailing line based on the underlying assumption that the newly introduced lines are corelated to lines of code in its proximity [2].

Reviewer ranking Algorithms is used by ReviewBot to recommend the potential reviewers. ID of PR and the diff revision for all the PR s are the inputs to the algorithm while the output is a list of reviewers arranged in descending order based on user points. Algorithms 3 describes this algorithm in detail.

Algorithm 3 ***Reviewer Rankers Algorithm***

```
id: Code Review Request ID, revision: File Diff
CodeReviewRequest newReq = retrieveReviewRequest(id)
Diff diff request = retrieveDiff (revision)
//Compute review request points
for (FileDiff newFileDiff : diff :retrieveFileDiffs()) do
    if (isNovelFile(newFileDiff )) then
continue
end if
```

```
SetOfRequests ←{}
     for (Line l : fileDiff :retrieveRelaventLines()) do
          lchdreq □ LCHfileDiffreq (l)
       α = initialDiffPoint(fileDiff )
       for (CodeReviewRequest cdr : lchdreq :history()) do
              r.points  = r.points + α
              α = α × δ
              SetOfRequests = SetOfRequests [ frg
        end for
    end for
end for
// Proliferate review request points to user points
CandidateSet = {}
for (ReviewRequest r : SetOfRequests) do
for (Candidate candidate : r :getCandidates()) do
candidate.points = candidate.points + r :points
CandidateSet candidate
end for
end for
potentailReviewers = Collections.
toArray(CandidateSet)
//Sort potential candidate reviewers
return potentailReviewers
```

Limitation of this algorithm is it cannot be used on pull requests with new files. Experimentation was carried out considering RevHistRECO as the baseline approach using two datasets first one being relatively large with 7035 pull requests and second one with 1676 pull requests. Review Bot's top-1 recommendation has accomplished an accuracy rate of 60% for both projects while RevHistRECO has only 34.15% and 47.83%, respectively for top 1 recommendation. ReviewBot achieved 80.85% accuracy rate for top-5 recommendation while RevHistRECO recorded 46.34% only accuracy rate for the first project whereas for the second project, that is the smaller project top 5 recommendation reached the 92.31% and 60.39% for ReviewBot and RevHistRECO respectively [2]. The experimental results reveal the fact the outperforms RevHistRECO in both types of projects.

2.2.3 Social relations-based approaches

Research considers pull request reviewing as a communal activity reliant on negotiations among code reviewers in GitHub, unlike bug fixing and feature enhancements which are only dependent on expertise of a developer which can be learnt from history of his bug fixes. Y. Yu et al. uses a

Comment Network, directly reflecting the interest and relations among developers to analyze the social relations among reviewers and contributors. Experimentation is based on 10 projects with more than 1000 merge requests and the evaluations reveal Comment Network based approach achieves improvements over machine learning based approach.

ML based approach is based on description of a pull request. Preprocessed text of a PR contains identifiers of the code files, variables and functions. For a developer who has submitted many pull requests the corpus of him is small mainly consisting of the above identifiers. Then the classifier biasedly assigns the pull request to him. Social network Based model addressed this problem by recommending the developers who share mutual interests with the contributor of code as reviewers. A network of comments to identify the developers having mutual interests has been build tracing historical comments [13].

A system of comments is constructed for each individual project between contributors and reviewers in a many-to-many model. A developer can coexist as a contributor of multiple pull requests and a reviewer for an already submitted merge request. A weighted directed graph denoted as Gcn = (V; E, W) where V denotes a developer, the relations between developers as E and W representing the importance of the relation. Weight is calculated using Equation 7.

$$W_{ij} = \sum_{r=1}^{k} W(ij,r) = P_{c*} \sum_{r=1}^{k} \sum_{n=1}^{m} \lambda^{n-1} * t(ij,r,n) \tag{7}$$

Overall count of merge requests submitted by vi is denoted by k. Weighted score assigned to a distinct pull request r is represented by w(ij,r). Pc is a pragmatic default value used to approximate the impact of an individual comment on the pull-request, and the total count of comments succumbed by vj in the same pull-request is denoted by m. t(ij;r;n) is a time-sensitive factor of related comment calculated using Equation 8.

$$t(ij,r,n) = \frac{timestamp(ij,r,n) - baseline}{deadline - baseline} \in (0,1] \tag{8}$$

where timestamp(ij,r,n) is the date the reviewer commented on pull request. The baseline and deadline are related to training set selection.

Novel merge requests are categorized into two classes based on the submitter. Pull-requests from Acquaintance Contributors (PAC) denotes a submitter for whom we can at least find one neighbor. Pull-Requests from New Contributors (PNC) denotes developers who has reviewed other pull requests but has not submitted any or a novel contributor as well as a reviewer.

Algorithm 4 Top-k recommendation for PAC [5]

```
Require: Gcn depicts the comment network of a given project; vs repersents the contributor
of a new pull request; topk referes to the number of reviewers of requirement;
Ensure: recSet is a set of sorted reviewers;
Q:enqueue(vs) and recSet  ← null;
repeat
        v ← Q:dequeue and Gcn:RankEdges(v)
        repeat
                if topk = 0 then
                        return recSet
                end If
                vnb ←  Gcn:BestNeighbor(v)
                Q:enqueue(vnb) and Gcn:mark(vnb)
                recSet [ fvnbg and topk = topk – 1
        until Gcn:Neighors(v) all marked
        until Q is empty
return recSet
```

For a PNC recommended reviewer would be the neighbor connected to the contributor in the previous pull requests. Improved breadth first search proposed by Algorithm 4 is suggested by this research to recommend top n reviewers. For a PNC a prediction on reviewers who share common interests is done using the comment network. For novel contributors lacking any connection to the neighbours, mining based on co-occurrences of patterns of pull requests is done. Apriori algorithm is used for rule mining in this approach. A new comer who lacks a node in the comment network, potential reviewers are the most active reviewers in corresponding communities [13].

Experimentation is carried out using dataset provided by Gousios et al. [17,18] evaluated on 10 projects with more than 1000 pull requests. Comment Network based approach achieves 78% accuracy in respect to the baseline. Approaches has been run on different projects including rails, bitcoin, jquery, phantomjs and homebrew.

Few developers seem to be assigned to review pull requests frequently as they have been submitting comments actively. Therefor the workload of these reviewers is increased.

Y. Yu et al. considers information retrieval and analyzing social relations to recommend reviewers for a pull request in another research where he considers word-based semantics of pull requests and social associations and connections of contributors [19] as the two key perceptions. Y. Yu et al. assumes the commenting and reviewing history of a reviewer can be used to find the expertise of a developer. For a novel PR, a contributor who has a commented in the frequent past for similar PR request is considered as a potential candidate. Similarly, the developers with similar interest are considered to have social relations [19].

2.2.4 Technology experience and expertise-based approaches

CORRECT approach is based on relevant cross project work history and expertise of developers with respect to technologies in recommending potential candidates for a PR [20]. Information on the expertise of candidate reviewers need to be mined form the codebase. Prevailing studies only emphasize on a project and disregard the expertise of the reviewers. Furthermore, the fundamental tools and technologies are subjected to rapid change and mining of the comment history, history of changed files or developer association history will not cater to the requirements.

This approach considers the experience with respect to external libraries and technologies used by the developers referred to as cross project experience. The intuition behind this is the pull requests using similar external libraries and technologies are relevant to each other. Thus, the reviewers who has reviewed similar pull requests are potential candidate reviewers for a novel pull request [20].

CORRECT is based on the hypothesis that likeness of two requests are computed on their shared libraries and shared technologies in the modified files Degree of similarity is measured using cosine similarity. Technologies and libraries with respect to a pull request is collected as a bag of words decomposing tokens with a delimiter from it. Similarity between the two bag of tokens are measured using cosine similarity using Equation 9.

$$CS\ (R_{c,}R_{i,}) = \frac{\sum_{k=1}^{n} C_{ck} * C_{ik}}{\sqrt{\sum_{k=1}^{n} C_{ck}^{2}} * \sqrt{\sum_{k=1}^{n} C_{ik}^{2}}} \tag{9}$$

Here, Cck corresponds to frequency of kth token from C in set Rc and Cik represents that frequency in set Ri. Code Reviewer Ranking Algorithm used in this approach input a novel pull requests and outputs a potential reviewer list. Firstly, it extracts the external libraries and technologies used in the revised files of the pull request. Then it uses a most recent set of pull requests from pull request history, extracts their libraries and technologies and calculated the similarity between the novel request and the pass requests and assigns a score to each request. Corresponding reviewers of the past requests are assigned a score where the frequent reviewers with high cosine similarity assigned a higher score [20].

Experimentation is based on 10 commercial projects with 17,115 pull requests and six open source projects. CORRECT achieves 85% - 92% with respect to recommendation accuracy, about 86% precision and 79% - 81% recall in code reviewer recommendation. Top-K Accuracy and Mean Reciprocal Rank (MRR) are used as performance evaluation metrics.

Many of the projects used for experimentation are medium sized projects but as testing on many projects was carried out and did not crash the algorithms is robust. Experimentation is carried on Python; Ruby and Java platforms and the system was not biased to any platform.

2.2.5 Profile based approaches

M. Fejzer et al. suggests maintaining a profile for each reviewer which is a multiset of all paths reviewed by the correspondent contributor. Whenever a novel pull request is reviewed by the contributor his profile gets updated. A similarity function is computed between profiles and the novel pull requests. M. Fejzer et al. states that state-of-the-art techniques of analyzing through all the history of commits and comments are impractical as the highly time consuming and resource utilization is very high [1]. To address this problem M. Fejzer et al. suggest a model grounded on profiles of the code reviewers. Profile is updated whenever a comment is added by the reviewer on a commit. That is for each novel pull request, the system scans for the reviewer profiles and not the commit history which saves time.

Hash tables has been used to implement reviewers' profiles since it has average constant time for insertions and scanning using hash function to plot distinct word in the profile for multiplicity. All the file paths of the commits of a reviewer are extracted, tokenized into tokens and added to the hash table. Multiset based representation (m(Ct)) of the commits is computed with O(K) complexity where K is the number of tokes of path segments.

For a novel pull request, a comparison between its multiset-based representation along with profiles of every reviewers is done suing an algorithm to identify the potential candidate to review the PR. Jaccard coefficient and the Tversky index are used as similarity functions. Jaccard coefficient computes the fraction of overlying elements belonging to two sets. Tversky index computes the variant to a prototype [21]. Tversky index is used as the main similarity function in this approach due to the opportunity of fine-tuning the importance ratio between a review and a profile. When a novel pull request arrives, a mapping it to its multiset-based representation m(Ct+1) is done. Then the m(Ct+1) is compared to profiles of reviewers. Top n potential reviewers are suggested by calculating the similarity of the profiles and m(Ct+1).

Experimentation is done using Android, LibreOffice, OpenStack and Qt projects. Precision, the recall and F-measure has been used as evaluation metrics [22,23]. Approach of M. Fejzer et al. with Tversky index as the similarity function achieves improved precision to-recall ratio and higher F-measure than all supplementary methods. Thus, experimentation results reveal that fact that this approach use significantly less computing power and memory therefore mainly advantageous to be used with bulky repositories such as Github [1].

2.3 Comparison of state-of-the-art techniques

Table 1: Comparison of the state-of-the-art techniques

Approach	Significance	Algorithm	Limitations	Advantages
Text based approaches	Considers PR as a text document. Based on title, description, and categorized with the names of developers who has at least submitted one comment to the request	Machine learning approaches.	Social relations of the reviewers have been ignored.	
File location-based approaches	Considers the file path of the changed files. Assumes that files in similar locality will be examined and reviewed by the same expertise and experienced developers.	File path similarity algorithms. String comparison techniques.	Poor performance for large projects. Time consuming. Reviewers may be overloaded with workload.	
Social relations-based approaches	Considers the social relations of the reviewers	Apriori algorithm	Frequently commenting reviewers can be overloaded	Timestamp of the comments has been considered.
Expertise based approaches	Concerns the libraries, technologies and expertise of the reviewers	Cosine similarity	Social relations of the developers have been ignored.	Cross projects are considered.
Profile based approach	Builds a profile considering the file paths of the changed files.	Jaccard coefficient and the Tversky index are used as		Significantly less computing

		similarity functions.		power and memory. Can be used with large projects

Table 2 summarizes the comparison of state-of-the-art techniques to sole the reserch problem. Considering file path-based approaches, Revfinder performs better than ReviewBot because Revfiner tracks changes in code history at file path level while ReviewBot tracks code change history at line level source code. Finding reviewers at line level is best for projects with recurrent changes. Files are not changed frequently [23]. 70% - 90% of code lines are uniquely changed at a single time and then left untouched, signifying that code review system lacks line-level history [3]. Therefor ReviewBot functionality is restricted.

From the literature review it is evident that most of the approaches are concerned on text-based techniques. Similarity is evaluated based on the title or description similarity of PR s. Some of the approaches considered the file path similarity of the changed files. Some approaches are concerned about the social relations among developers. Few approaches are concerned about the expertise of the developers derived from the libraries they have used priorly in their prior development and reviewing. None of the approaches have considered the activeness of the reviewer for recommending reviewers. Only a very few researches are based on combination of text based approached and file path similarity approaches.

On the above inference I have decided to proceed on the combined approach of text based and file path similarity-based approaches along with the activeness of the developers for my research.

Chapter 3

3. Methodology

3.1 Introduction

Methodology focusses on the way in which the research question has been addressed promptly with the knowledge gained through the analysis of similar literature. Through this chapter a solution is proposed to the identified research gap in the recent literature. The conceptual approach is explained in terms of input, output and process for the modules. Flow of inputs and outputs between the individual modules is identified through this chapter. Furthermore, provides an overview of the implementation methodology adopted during the project to achieve the research goal.

3.2 Proposed research solution

This research aims at identifying the features of a pull requests, reviewer expertise and how these features could be used to calculate a similarity between the reviewers and pull requests. Research includes studying literature to investigate on potential attributes to be considered on suggesting the top reviewers for a pull request (Eg: Activeness of the reviewer, File path similarity, Text similarity).

After identifying the potential candidate attributes, investigating on the available datasets on pull requests of multiple projects and identifying the potential features to be extracted to develop a dataset is done. Afterwards features are extracted, and a dataset of multiple repositories associated with a combination of programming languages is developed.

A similarity measure is used to identify the potential reviewers for a novel PR based on the history if reviewing in the repository. An attribute weight is allocated for each of the attributes to identify the prominent attributes. Using a ranking algorithm, the potential reviewer candidates are ranked based on weighted attributes, the system will notify the integrators who have been identified as the potential reviewers.

At the end of the project an automated system to recommend reviewers of a pull request is developed. The high-level data flows of the system are depicted in Figure 3.

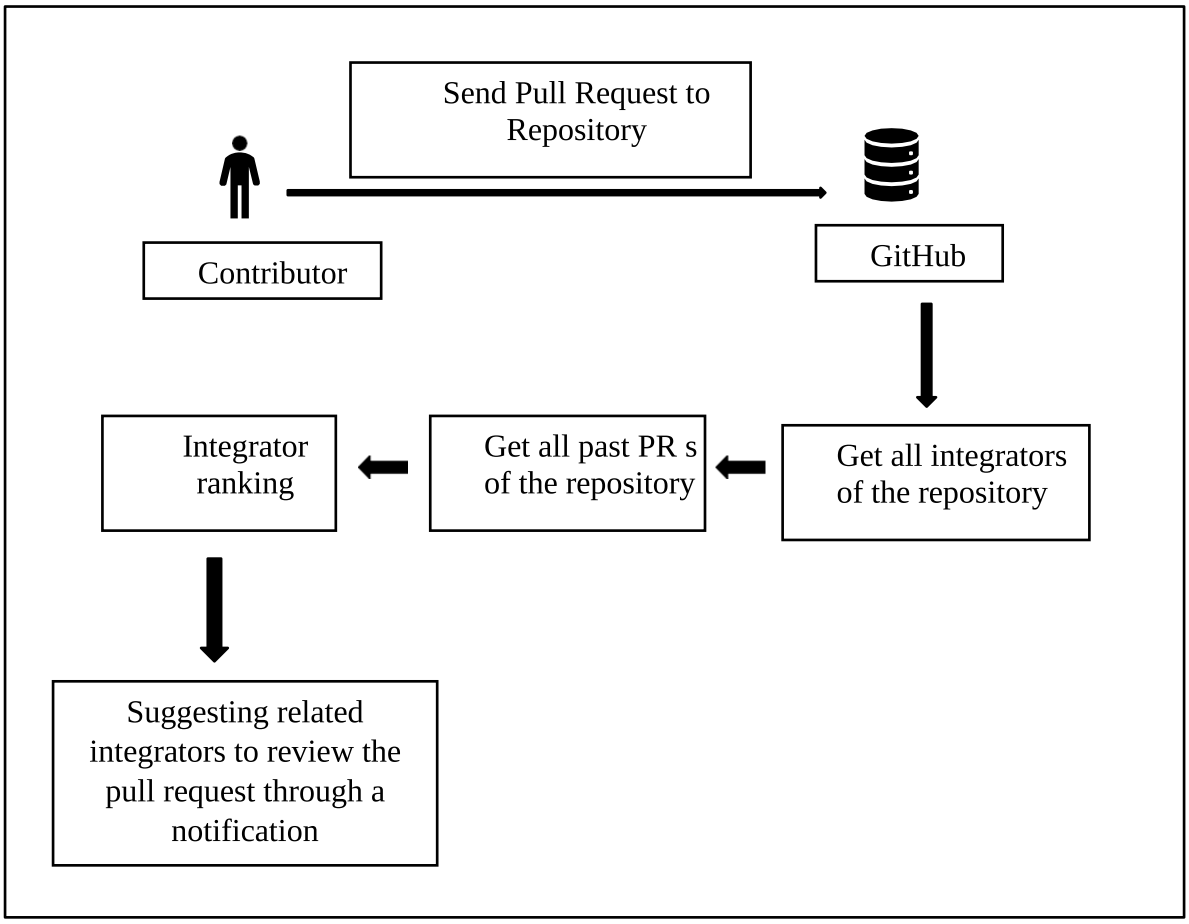

Send Pull Request to Repository
Contributor
GitHub
Get all integrators of the repository
Get all past PR s of the repository
Integrator ranking
Suggesting related integrators to review the pull request through a notification

Figure 3: High level data flow diagram of the system

3.3 High level design of the system

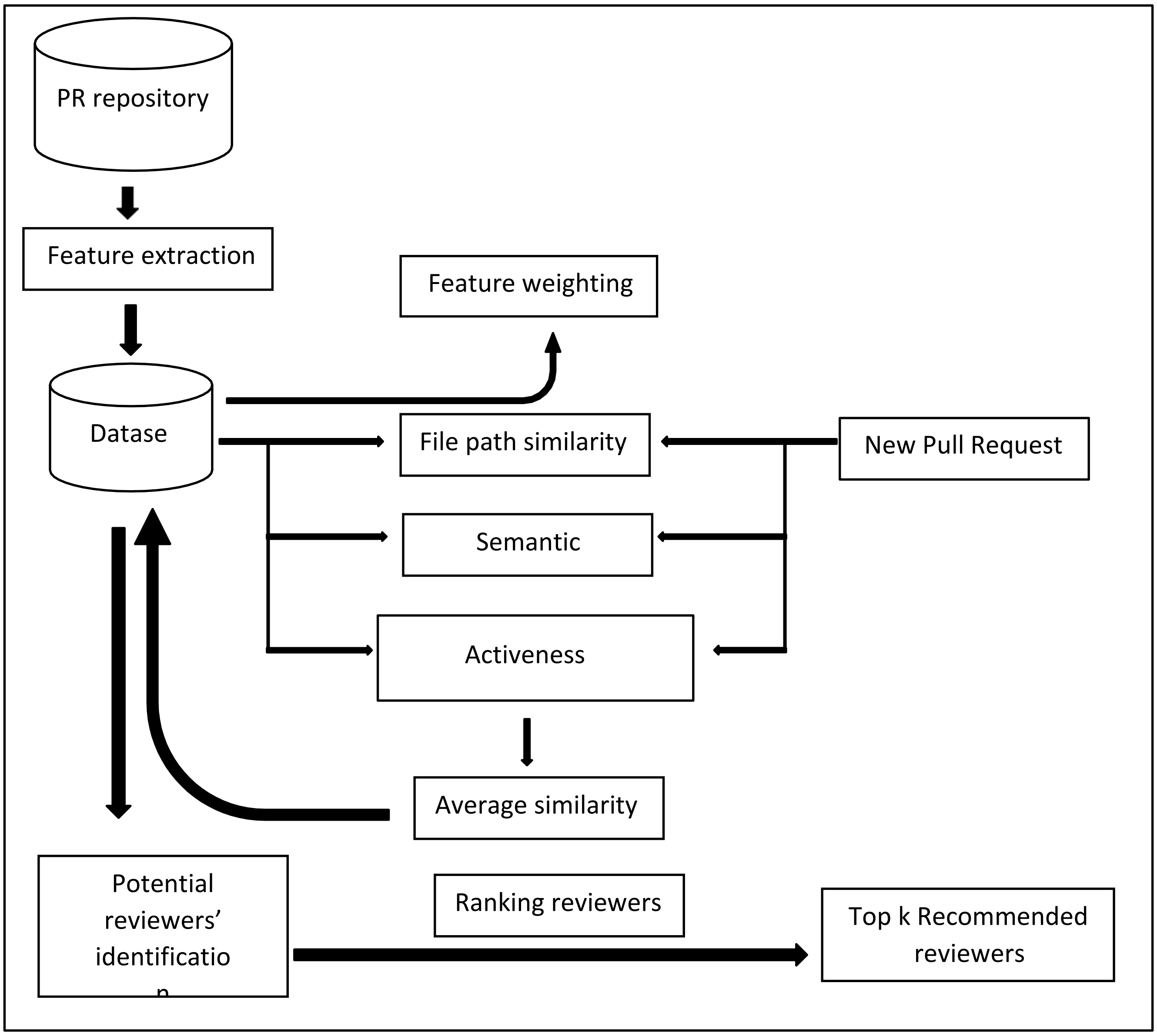


Figure 4: High level architecture of the system

Figure 4 diagrammatically depicts the high-level architecture of the proposed solution.

3.4 Feature extraction and dataset collection

For the development of a dataset, relevant attributes featuring a pull request need to be extracted from a pull request repository. A pull request is characterized by multiple textual attributes such as

- Title
- Description
- Reviewer
- Commented developers

- File paths of the changed files
- Libraries used
- Technologies

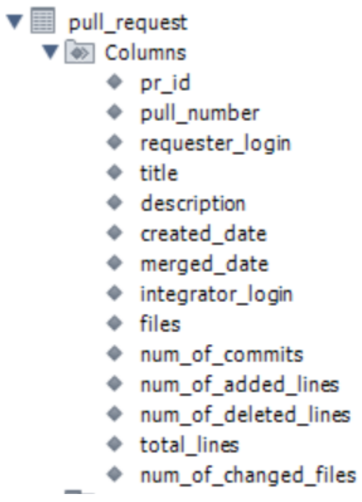
Furthermore, reviewer is featured by numerical attributes such as

- Activeness
- Review latency
- Number of pull request reviewed
- Number of comments

From the above attributes featuring a pull request and a reviewer I have chosen title, description, reviewer, file paths of changed files and activeness for this research on the basis that the experimentation on the analysis of impact of the attributes for reviewer recommendation in the literature reveals that above attributes have a higher impact [24]. Each of the above attributes needs to be extracted from a pull request database. Title and description are represented as bag of words. Libraries and technologies used on the changed files are also represented as bag of words. File paths of the changed files are extracted and represented in the database.

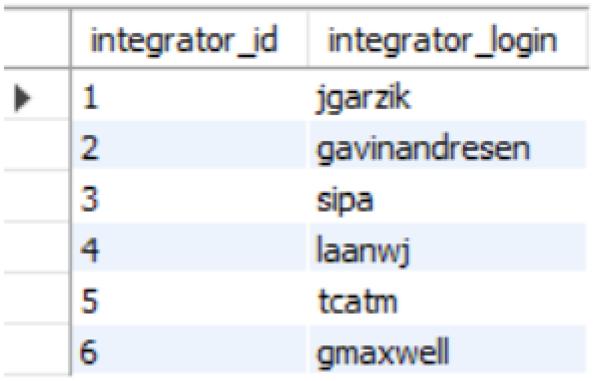
Activeness is an attribute measured by the date of the developers last commit. Review latency is a measure of the duration between PR created date and merged date.

A dataset consisting of pull request with the above mention textual and numerical attributes is created. A database with two entities – Pull request and Integrator is developed.



```
▼ pull_request
    ▼ Columns
        ◆ pr_id
        ◆ pull_number
        ◆ requester_login
        ◆ title
        ◆ description
        ◆ created_date
        ◆ merged_date
        ◆ integrator_login
        ◆ files
        ◆ num_of_commits
        ◆ num_of_added_lines
        ◆ num_of_deleted_lines
        ◆ total_lines
        ◆ num_of_changed_files
```

Figure 5: Data model of pull request entity



integrator_id	integrator_login
1	jgarzik
2	gavinandresen
3	sipa
4	laanwj
5	tcatm
6	gmaxwell

Figure 6: Integrator table

3.5 Recommending reviewers for a PR - Real time processing

3.5.1 Similarity calculation

The research is experimentation on the fact that similarity between pull requests is characterized by semantic similarity between pull requests in title, description and similarity of changed file paths.

Semantic similarity of titles and descriptions

The tiles and descriptions of pull requests are extracted and indexed. A vector space model is used for indexing. Standard preprocessing by tokenization, stemming and stop words removal is done on title and description. Porter stemming algorithm [25] is used for stemming. Preprocessed text is transformed to multidimensional vector computable in Vector space Model. Each dimension of the vector represents a distinct word in the corpus of the text build by all pull requests. TD-IDF model [26] is used to calculate the value of $W_{j,n}$ which denotes the weight of the nth entry in the vector of jth text. Equation 10 is used for calculation.

$$W_{j,n} = tf_{j,n} \times id_{fn} \tag{10}$$

The term frequency which is the frequency of nth term appearing in the jth text is represented by $tf_{j,n.}$. Distinguishing characteristic of a term is signified by $f_{j,n}$ which is the inverse term frequency.

Cosine Similarity [27] is used to measure the similarity between two PRs represented as a collection of texts in vectors after transformation. Equation 11 is used for calculation of two titles and description represented as bag of words.

$$Sim(i,j) = \frac{TextVec_i \, . TextVec_j}{|TextVec_i||TextVec_j|}$$

$$= \frac{\sum_{m=1}^{m=v} W_{i,m} * W_{j,m}}{\sqrt{\sum_{m=1}^{m=v} W^2{}_{i,m}}\ \sqrt{\sum_{m=1}^{m=v} W^2{}_{j,m}}} \tag{11}$$

Two similarities between two pull-requests is calculated based on similarity between title and similarity between descriptions.

Similarity between file paths of changed files

File path similarity function [3] is used to measure the similarity between the changed files of two PRs.

$$Similarity\ (f_{n,}f_{p,}) = \frac{commonPath(f_{n,}f_{p,})}{\max\,(Length(f_n),Length(f_p))} \tag{3}$$

The StringComparison(fn, fp) function compares components of file paths and returns the common components. In this research an average of scores is calculated using Longest Common Substring (LCSubstr), Longest Common Prefix (LCP), Longest Common Subsequence (LCSubseq) and Longest Common Suffix (LCS) methods have been used. The reason behind the use of an average score is that the combination of the results of individual techniques has been successfully shown to improve the performance in the data mining and software engineering domains [28, 29].

File Path Similarity Algorithm [14] computes a score for a past review (R_p) from an average of similarity of every file in R_p (f_p) comparing with every file in Rn (f_n). File returns the array of file paths of the novel PR. The Similarity (f_n,f_p) function computes the likeness between f_p and f_n.

$$FSP(R_{n,}R_{p,}m) = \frac{\sum_{\substack{f_{n} \in Files(R_{n,})\\ f_p \in Files(p)}} Similarity\ (f_{n,}f_{p,})}{|\,Files(R_{n,})| * |\,Files(p)|} *\ \delta^m \tag{2}$$

Activeness of the reviewer

Activeness of the reviewer is calculated as the difference between the new PR created date and the merged date of the last PR reviewed by the reviewer. Time is decaying over a lambda constant. Equation 12 is used for calculating the activeness of an integrator.

$$Activeness = New\ PR\ created\ date - Last\ PR\ merged\ date$$

(12)

3.5.2 Recommending reviewers for a novel PR

When a new PR is submitted, for each of the integrators in the database, an average score based on the similarity of all PR's reviewed by each integrator is calculated. Similarity between the new PR and the old PR's is calculated as an average score of file path similarity, text similarity and activeness. All the calculated scores are standardized into one frame and stored in the database.

$$\text{Average integrator score} = \frac{\sum_{i=1}^{n} Text\ similarity + File\ pathe\ similarity + Activeness}{n} \tag{13}$$

For each of the integrators a score is calculated by averaging the score of priorly reviewed PRs. Equation 13 is used to calculate the average integrator score where n is the number of PR's reviewed by the integrator. Ranking of the reviewers is done based on the average integrator scores of each integrator.

When a PR which is already processed is added to the system to find the integrators, system navigates through the database to find the recommendation list. Calculation process will not happen twice to increase the efficiency.

3.5.3 Assigning weighted scores for attributes

A PR is characterized by multiple attributes. These attributes are assigned weights based on its importance for the reviewer assignment. Supervised learning approach is used for training a model for feature weighting. In this research decision tree learning algorithm [30] is used for supervised learning to rank the features based on is importance. Decision tree algorithm is the most effective method of predicting a value of a target variable based on several input features [31].

Decision Tree algorithm calculates the entropy of a class using Equation 14

$$Entropy = \sum_{i=1}^{s} p_i \log {}^{1}/_{p_i} \tag{14}$$

Information gain is calculated using Equation 15 and the split with the maximum gain is chosen as the splitting criteria.

$$Gain(D,S) = Entropy(D) - \sum_{i=1}^{s} P(D_i)\, Entropy(D_i) \tag{15}$$

Weights of the training data set is initialized using the posterior probabilities of each class. Count of incidences of each attribute value A_{ij} is taken, to calculate $P(A_{ij})$ for each attribute,

A_i. Probability P (A_{ij} | C_j) is estimated by counting the occurrence of attribute value in class C_j in the training data. For every value of attributes, P (A_{ij} | C_j) are determined. Initialization of the weights of the attributes is done by multiplying the probabilities of distinct attribute values from the training instances using these probabilities.

For training instance ei having independent attribute values {A_{i1}, A_{i2},…,A_{ip}} as we have already calculated P(A_{ik} | C_j), for each class C_j and attribute A_{ik} , P(e_i |C_j) is estimated by Equation 16.

$$P(e_i \,|C_j) = P(C_j) \prod_{k=1-\rightarrow p} P(A_{ij}|C_j) \quad (16)$$

Likelihood of e_i in each class is calculated to initialize weights for attributes. The posterior probability P (C_j | e_i) is calculated for each class. Afterwards the weight of the training instance is assigned with the maximum posterior probability for that training instance.

A weighted score is assigned to each of the attribute's activeness, file path similarity and text similarity to prioritize each of the factors. Experimentation is carried on which factors affect mostly in highly accurate recommendation list generation. Weights can be varied by the system and observations on how each factor affects the result is analyzed.

Default weighted scores are assigned based on the above method. Defaults scores are set as Activeness- 0.7, File path similarity – 0.1, Text similarity – 0.2

3.5.4 Accuracy calculation

Accuracy calculation is done assuming the reviewers in the real code base as the ground truth. This is done to experiment on difference weight combinations and analyze how it could affect the overall accuracy of the system.

Accuracy calculation allows to set an offset which the starting PR id from the database and sets a window frame which is the set of records the system considers for accuracy calculation. For each of the records starting from the offset, top k recommendation lists are generated for each PR. If the actual reviewer of the code base for a specific PR record is within the top k generated list for each PR, a score of 1 is assigned. Average accuracy score is generated likewise for all combinations of weighted attributes.

Chapter 4

4. Evaluation and Results

4.1 Research hypothesis and research questions

The research focusses on development of a pull request reviewer recommendation system. It identifies the features characterizing a pull request such as title, description, file paths of changed files, libraries and technologies used in changed files and develops a similarity between pull requests to recommend reviewers for a pull request reviewing process.

Identifying the potential reviewers is done using similarity measures. Ranking of reviewers is based on a ranking algorithm considering the weights of attributes characterizing the potential reviewers such as expertise, activeness, review latency, number of comments.

The goal of our experiential study is to assess the effectiveness of the suggested approach in terms of accuracy in ranking of reviewers for a pull request. With the intension of achieving the above research goal, we discourse the subsequent research questions.

Question 1: Does the suggested system precisely recommend code reviewers?

We propose our approach to suggest the appropriate code reviewers since similarity of title, description, file paths of changes files would lead to better accuracy.

Question 2: Does the suggested approach performs better in ranking of recommended code-reviewers?

Recommending most apposite code-reviewers in the top rankings effortlessly will comfort the developer and will also avoid intrusive disparate code-reviewers. The above research goal is set to evaluate the performance of our approach in ranking the reviewers.

Question 3: Does the suggested approach outperform or performs similarly with the state-of-the-art techniques for reviewer recommendation?

Currently the state-of-the-art techniques for reviewer recommendation includes either similarity measurements on file paths, profile-based recommendation techniques, technology expertise, social relations etc. This research aims at finding whether the combination of all the parametric would outperform the current techniques.

4.2 Evaluation approach

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 [20]. Therefor the evaluation process is a mathematical calculation-based method on real repositories on different projects.

4.3 Dataset collection

Evaluation approach is based on 3 open source repositories which have received over 1000 pull-requests. These projects are of different languages – a combination of multiple languages. Bitcon repository is based mainly on C++, Python and C languages while akka repository is based on Scala and Java. Pull request data is collected from GitHub using GitHub API. They are popular and widely used. Pull requests lacking reviews will be discarded. Some of the pull requests of core developers are reviewed by themselves which deviates from normal behavior. Therefor these pull requests are also discarded. For each project, developers who have reviewed others' pull requests are identified as the reviewer candidates. Table 3 summarizes the statistics of the experimental opens source repositories.

Table 2: Summarization of experimental dataset

Project	Language	Total Pull requests	Total PR Reviewers
Akka	C++, Python	3842	19
Bitcoin	Java, Scala	5104	38
Rubocop	Ruby	5000	19

4.4 Evaluation metrics

As the research focusses on recommendation we use the following evaluation metrics for performance evaluation of the proposed system.

Top-K Accuracy:

Percentage of pull requests with at least one reviewer precise recommendation within the Top-k recommendations by a technique. The top-k accuracy can be calculated using Equation 17 where R is a collection of reviews,

$$\text{Top-k Accuracy (R)} = \frac{\sum_{r \in R} Correct\ top{-}k\ recmmendations}{R} * \ 100\% \qquad (17)$$

Mean Reciprocal Rank (MRR):

Average of mutual ranks of correct code-reviewers in a recommendation list. Equation 18 calculates MRR where R is a collection of reviews. rank(candidates(r)) returns the first rank of actual code-reviewers in the recommended potential candidate list candidates(r). If there are no true positive code-reviewers in the recommendation list,1/(rank(candidates(r))) will return 0. Preferably, a method with faultless ranking will reach an MRR of 1.

$$\text{MRR}=\frac{1}{|R|}\sum_{r\in R}\frac{1}{\text{rank(candidates(r))}} \tag{18}$$

4.5 Experimentation results

Experimentation is conducted on 3 open source projects and the results achieved is summarized below. Using a sliding window-based approach of window size=100, I have collected a candidate reviewer list from prior merge requests. Candidates are ranked using our approach and the results are evaluated and summarized. Table 3 summarizes the performance of our approach on different projects.

Table 3: Summarization of results of the research approach

Project	**Top K accuracy**	**MRR**
Akka	82%	0.718
Bitcoin	88.3%	0.822
Rubocop	84.7%	0.763

Accuracy for each of the repositories is calculated for different weight combinations varying the window size. Results for Akka repository with a window size of 100 is visualized as below in Table 4.

Table 4: Accuracy calculation for weight combinations for AKKA repository

File path similarity	Text similarity	Activeness	Top 1 accuracy	Top 3 accuracy	Top 5 accuracy	MRR
0.1	0.1	0.8	0.51	0.92	1	0.713
0.1	0.2	0.7	0.53	0.92	1	0.717
0.1	0.3	0.6	0.52	0.92	1	0.708
0.1	0.4	0.5	0.5	0.92	1	0.698
0.1	0.5	0.4	0.51	0.92	1	0.703
0.1	0.6	0.3	0.49	0.92	1	0.693

0.1	0.7	0.2	0.49	0.92	1	0.693
0.1	0.8	0.1	0.5	0.93	1	0.703
0.2	0.1	0.7	0.52	0.92	1	0.717
0.2	0.2	0.6	0.52	0.92	1	0.71
0.2	0.3	0.5	0.5	0.92	1	0.7
0.2	0.4	0.4	0.51	0.92	1	0.703
0.2	0.5	0.3	0.5	0.93	1	0.698
0.2	0.6	0.2	0.49	0.93	1	0.694
0.2	0.7	0.1	0.48	0.93	1	0.694
0.3	0.2	0.5	0.5	0.93	1	0.703
0.3	0.1	0.6	0.52	0.92	1	0.71
0.3	0.3	0.4	0.53	0.93	1	0.716
0.3	0.4	0.3	0.51	0.93	1	0.703
0.3	0.5	0.2	0.49	0.93	1	0.696
0.3	0.6	0.1	0.48	0.93	1	0.693
0.4	0.1	0.5	0.51	0.93	1	0.706
0.4	0.2	0.4	0.53	0.93	1	0.718
0.4	0.3	0.3	0.51	0.93	1	0.706
0.4	0.4	0.2	0.48	0.93	1	0.691
0.4	0.5	0.1	0.48	0.93	1	0.689
0.5	0.1	0.4	0.5	0.93	1	0.701
0.5	0.2	0.3	0.51	0.93	1	0.706
0.5	0.3	0.2	0.49	0.93	1	0.693
0.5	0.4	0.1	0.46	0.93	1	0.679
0.6	0.1	0.3	0.48	0.93	1	0.686
0.6	0.2	0.2	0.48	0.93	1	0.681
0.6	0.3	0.1	0.45	0.93	1	0.673
0.7	0.1	0.2	0.47	0.93	1	0.684
0.7	0.2	0.1	0.47	0.93	1	0.678
0.8	0.1	0.1	0.49	0.93	1	0.686

From the above experimentation it was revealed that highest accuracy is gained with a weight combination of the following weights. Table 5 summarizes the feature weight combinations achiving highest accuracy on multiple repositories.

Table 5: Summarization od feature weight combinations of highest accuracy

Repository	File path similarity	Text similarity	Activeness	MRR
Akka	0.4	0.2	0.4	0.718
	0.2	0.1	0.7	0.716
	0.3	0.3	0.4	0.715
Bitcoin	0.3	0.1	0.6	0.822
	0.2	0.2	0.6	0.818
	0.2	0.1	0.7	0.810
Rubucop	0.3	0.3	0.4	0.763
	0.2	0.1	0.7	0.758
	0.4	0.2	0.4	0.741

Comparison of results obtained for multiple window sizes is summarized below in Table 6.

Table 6: Comparison of accuracy against dataset window sizes

Repository	Window size	Top 1 Accuracy	Top 3 accuracy	Top 5 Accuracy	MRR
Akka	30	0.63	1	1	0.78
	50	0.54	1	1	0.73
	100	0.53	0.93	1	0.72
Bitcoin	30	0.47	1	1	0.72
	50	0.54	1	1	0.76
	100	0.68	0.97	1	0.82
Rubocop	30	0.69	1	1	0.84
	50	0.59	1	1	0.77
	100	0.57	0.94	1	0.76

4.6 Discussion

After analyzing the results of Table 4 it can be revealed that highest accuracy is obtained with weighted attribute combination of File path similarity – 40%, Text similarity – 20% and Activeness – 40%. Table 5 summarizes the results of multiple datasets and from the results it can be inferred that higher weight factors for activeness always contributes towards gaining higher accuracy scores. File path similarity is contributing next towards higher scores. Therefor it can be inferred that importance of factors towards gaining higher accuracy rates in recommendation of code reviewers can be rated as activeness > file path similarity > text similarity.

Table 6 summarizes the evaluation results of multiple repositories. In almost all of the results obtained Top 5 accuracy is 1 which concludes that the Top 5 recommenders always included the real PR reviewer in the dataset. Top 3 accuracy is also at higher scores whereas Top 1 accuracy is between 50%-70%.

On considering threats to validity Almost all our experimental projects are medium sized projects. Extending the research on bulk projects would enhance the stress testing of the system. I have examined the system across multiple languages. During the experimentation on multiple languages any biasness towards a language was not observed which generalize our finding on the approach.

On consideration of construct validity non availability of retirement details of code reviewers will affect the accuracy of reviewer recommendation as we consider the pull request review history for potential candidate list formulation. Furthermore, as we does not consider the workload balancing factor, reviewers may be overburdened with review requests.

Chapter 5

5. Conclusion and Future work

5.1 Introduction

This chapter wraps up this documentation on the research by summarizing the final quest thoughts, comments and future work to enhance research on this study. This research is an effort to explore an efficient approach for recommendation of reviewers for pull request reviewing process. The research study was supported out with three core steps comprehensive literature review, analysis of state-of-the-art methods and algorithms for reviewer recommendation and experimentation on the suggested research approach for its performance and evaluation against the real-world repositories for its accuracy.

5.2 Problem addressed and solution proposed

Pull request reviewing is a critical activity in project management as we step towards the collaborative and open source development culture. Any third-party developer can fix bus or introduce new features to an open source project. Core developers of the project is responsible for maintaining code quality. For this purpose, pull requests submission and code reviewing has been encouraged. An external developer makes a clone from the repository, do some changes on code and submits the changes as a Pull request to the central repository. The person who submits the pull request is responsible for assignment of a reviewer for reviewing the pull request.

This is the place where the research fixes in. The person who submits the pull request is not familiar with the prior code reviewers and file structures of the repository. A new submitter finds it hard to assign the pull request to an apposite developer. The new developer is not familiarized with the expertise of languages and libraries of the existing integrators. He will have to dig into the repository to find the expertise technologies. Assignment of the pull request to wrong reviewer would result in the PR idling for long time. If the reviewer is not active PR will be left unattended for long time and will not be picked by some other reviewer. Therefor the research is about developing a pull request reviewer recommendation system. Objectives of the research includes investigation on the potential parameters for recommending apposite reviewers for a pull request by studying the literature, feature extraction and developing a dataset characterizing a pull requests after identification of the features, development of the recommendation system and experimentation and evaluation of the developed system against real world repositories.

On submission of a new PR, the system analyses the history of the repository and calculates a similarity between the submitted new PR and the already reviewed PRs. Based on text similarity, file path similarity and activeness of the integrators, the system ranks the integrators and recommends the top 5 integrators for reviewing a pull request.

Furthermore, for the purpose of evaluation and experimentation accuracy is calculated for the suggested integrators for the PRs. Weights are assigned for identified attributes characterizing pull request and integrator. Experimentation is carried out on the weight assignment to yield the highest accuracy. Three github repositories each with more than 3000 pull requests Akka, Bitcoin and Rubocon is used for experimentation and dataset development. System has received an average accuracy level around 80% - 85%.

5.3 Future work

Research conducted does not consider social networking of the reviewers. Experimentation on using social relationships and networking of reviewers for enhancing the performance of reviewer recommendation by identifying reviewers with the same expertise based on the discussions they have actively participated using their comment network can be carried out as future extension to this research.

This research does not identify expertise and libraries used by developers on cross project basis. Cross project references an expertise on libraries and technologies of reviewers is not focused in this research.

Workload balancing among integrators is not considered in this research. As a result, an active integrator could be overloaded with PR reviews. Considering the work load and indicating it as hint on recommending reviewers about the unattended PR s left for an integrator would enhance the performance of the system.

Further research could be carried out on handling concurrent recommendation requests on real time. In theoretical aspect mixing of this approach with Convolutional Neural Network recommendation-based approach will achieve better accuracy and is worthy of further experimentation.

References

[1] F. Mikołaj, P. Piotr & S. Krzysztof. (2017). Profile based recommendation of code reviewers. Journal of Intelligent Information Systems. 10.1007/s10844-017-0484-1. 14

[2] V. Balachandran, “Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation,” in ICSE ’13, 2013, pp. 931–940 9

[3] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida and K. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, 2015, pp. 141-150. doi: 10.1109/SANER.2015.7081824

[4] P. Weiß gerber, D. Neu, and S. Diehl, “Small Patches GetIn !” in MSR’08, 2008, pp. 67–75

[5] P. C. Rigby and C. Bird, “Convergent Contemporary Software Peer Review Practices,” in ESEC/FSE 2013, 2013, pp. 202–212

[6] J. Tsay, L. Dabbish, and J. Herbsleb, “Let’s Talk About It: Evaluating Contributions through Discussion in GitHub,” in FSE’14, 2014, pp. 144–154

[7] P. C. Rigby and M.-A. Storey, “Understanding broadcast-based peer review on open source software projects,” in ICSE’11, 2011, pp. 541–550.

[8] V. Mashayekhi, J. Drake, W.-T. Tsai, and J. Riedl, “Distributed, collaborative software inspection,” IEEE Software, vol. 10, no. 5, pp. 66–75, 1993.

[9] Drake, J., Mashayekhi, V., Riedl, J., & Tsai, W.T. (1991). “A distributed collaborative software inspection tool: Design, prototype, and early trial. In Proceedings of the 30th aerospace sciences conference”

[10] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in Proceedings of the 28th International Conference on Software Engineering,ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 361–370.

[11] G. Jeong, S. Kim, and T. Zimmermann, “Improving bug triage with bug tossing graphs,” in Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE ’09. New York, NY, USA: ACM, 2009, pp. 111–120.

[12] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, “Automated, highlyaccurate, bug assignment using machine learning and tossing graphs,” Journal of Systems and Software, vol. 85, no. 10, pp. 2275–2292, 2012

[13] Yu, Yue & Wang, Huaimin & Yin, Gang & Ling, Charles. (2014). Who Should Review this Pull-Request: Reviewer Recommendation to Expedite Crowd Collaboration? 335-342. 10.1109/APSEC.2014.57.

[14] P Thongtanunam et al., “Improving code review effectiveness through reviewer recommendations”, Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, June 02-03, 2014, Hyderabad, India [doi:10.1145/2593702.2593705]

[15] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, 1997

[16] R. Ranawana and V. Palade, “Multi-Classifier Systems - Review and a Roadmap for Developers,” IJHIS, vol. 3, no. 1, pp. 1–41, 2006.

[17] G. Gousios and A. Zaidman, “A dataset for pull request research,” in MSR ’14: Proceedings of the 11th Working Conference on Mining Software Repositories, may 2014,

[18] G. Gousios, “The ghtorrent dataset and tool suite,” in Proceedings of the 10th Working Conference on Mining Software Repositories, ser. MSR’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.

[19] Yu, Yue. (2014). Reviewer Recommender of Pull-Request in GitHub. 10.1109/ICSME.2014.107.

[20] M. M. Rahman, C. K. Roy & J. A. Collins (2016). CORRECT: Code Reviewer Recommendation in GitHub Based on Cross-Project and Technology Experience. 10.1145/2889160.2889244.

[21] Terra, R., Brunet, J., Miranda, L.F., Valente, M.T., Serey, D., Castilho, D., & da Silva Bigonha, R. (2013). Measuring the structural similarity between source code entities (S). In The 25th international conference on software engineering and knowledge engineering, Boston, MA, USA, June 27-29, 2013 (pp. 753–758). Knowledge Systems Institute Graduate School.

[22] Lee, J.B., Ihara, A., Monden, A., & Matsumoto, K. (2013). Patch reviewer recommendation in OSS projects. In Muenchaisri, P., & Rothermel, G. (Eds.) 20th Asia-Pacific software engineering conference, APSEC.

[23] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, “Expert Recommendation with Usage Expertise,” in ICSM’09, 2009, pp. 535–538

[24] D. M. Soares, M. L. de L. Junior, A. Plastino, and L. Murta “What Factors Influence the Reviewer Assignment to Pull Requests?" Information and Software Technology. 98. 10.1016/j.infsof.2018.01.015, (2018)

[25] M. F Porter. 1997. An algorithm for sufx stripping. Morgan Kaufmann Publishers Inc. 130– 137 pages

[26] Chengnian Sun, D Lo, Xiaoyin Wang, and Jing Jiang. 2010. “A discriminative model approach for accurate duplicate bug report retrieval” In ACM/IEEE International Conference on Software Engineering. 45–54

[27] Paul Kantor. 1999. Foundations of Statistical Natural Language Processing. MIT Press, pags. 91–92 pages.

[28] J. Kittler, I. C. Society, M. Hatef, R. P. W. Duin, and J. Matas, “On Combining Classifiers,” IEEE TPAMI, vol. 20, no. 3, pp. 226–239, 1998.

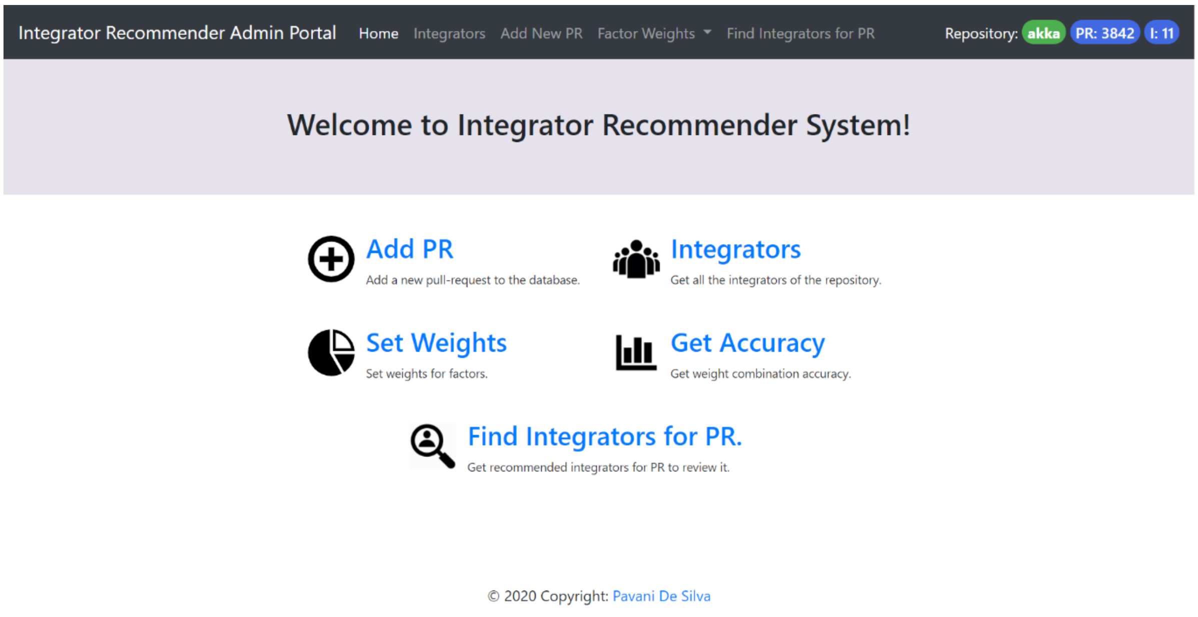
[29] T. K. Ho, J. Hull, S. N. Srihari, and S. Member, “Decision Combination in Multiple Classifier Systems,” IEEE TPAMI, vol. 16, no. 1, pp. 66–75, 1994

[30] A. Colin, 1996, “Building Decision Trees with the ID3 Algorithm", Dr. Dobbs Journal, June 1996.

[31] L. Rokach, O. Maimon. “Data mining with decision trees: theory and applications”. World Scientific Pub Co Inc. ISBN 978-9812771711, 2008.

APPENDIX

Appendix A – Home Screen of the System



Integrator Recommender Admin Portal Home Integrators Add New PR Factor Weights Find Integrators for PR Repository: akka PR: 3842 I: 11

Welcome to Integrator Recommender System!

Add PR
Add a new pull-request to the database.

Integrators
Get all the integrators of the repository.

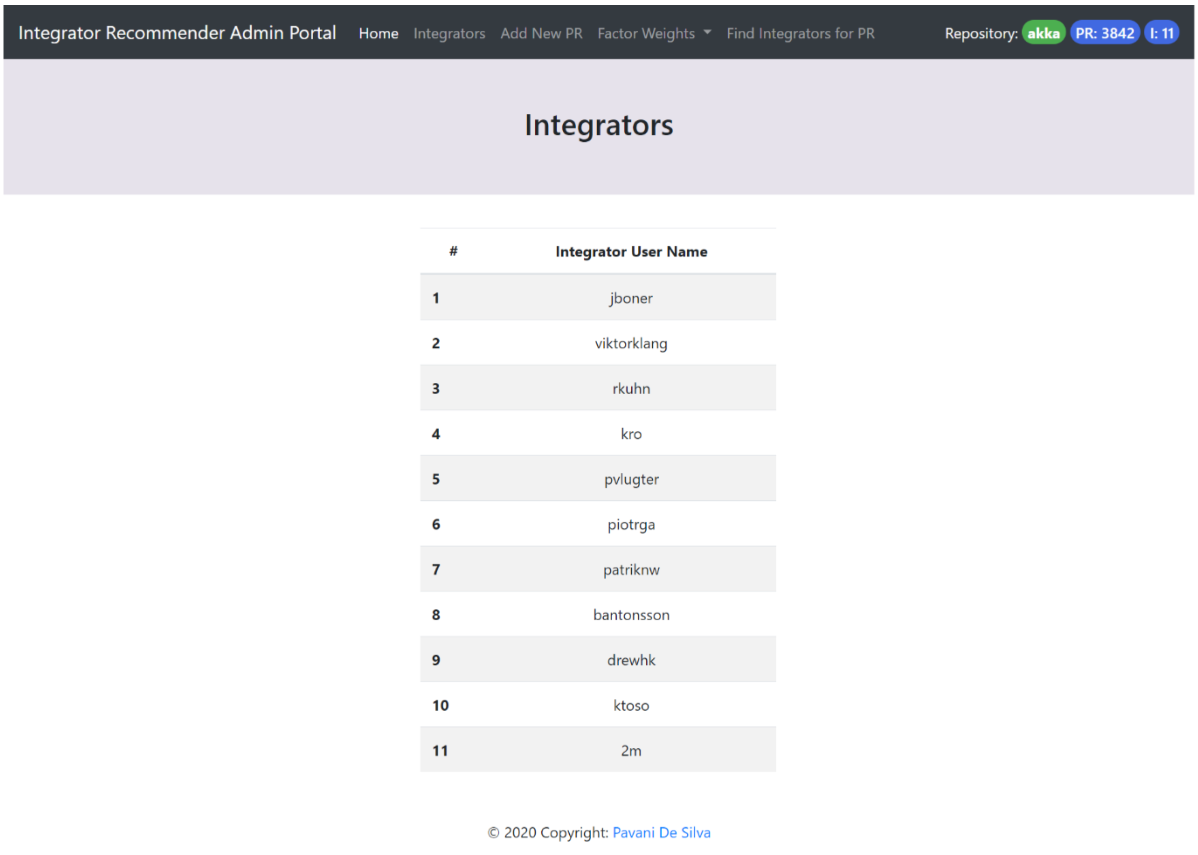
Set Weights
Set weights for factors.

Get Accuracy
Get weight combination accuracy.

Find Integrators for PR.
Get recommended integrators for PR to review it.

© 2020 Copyright: Pavani De Silva

Appendix B – All integrators of the repository



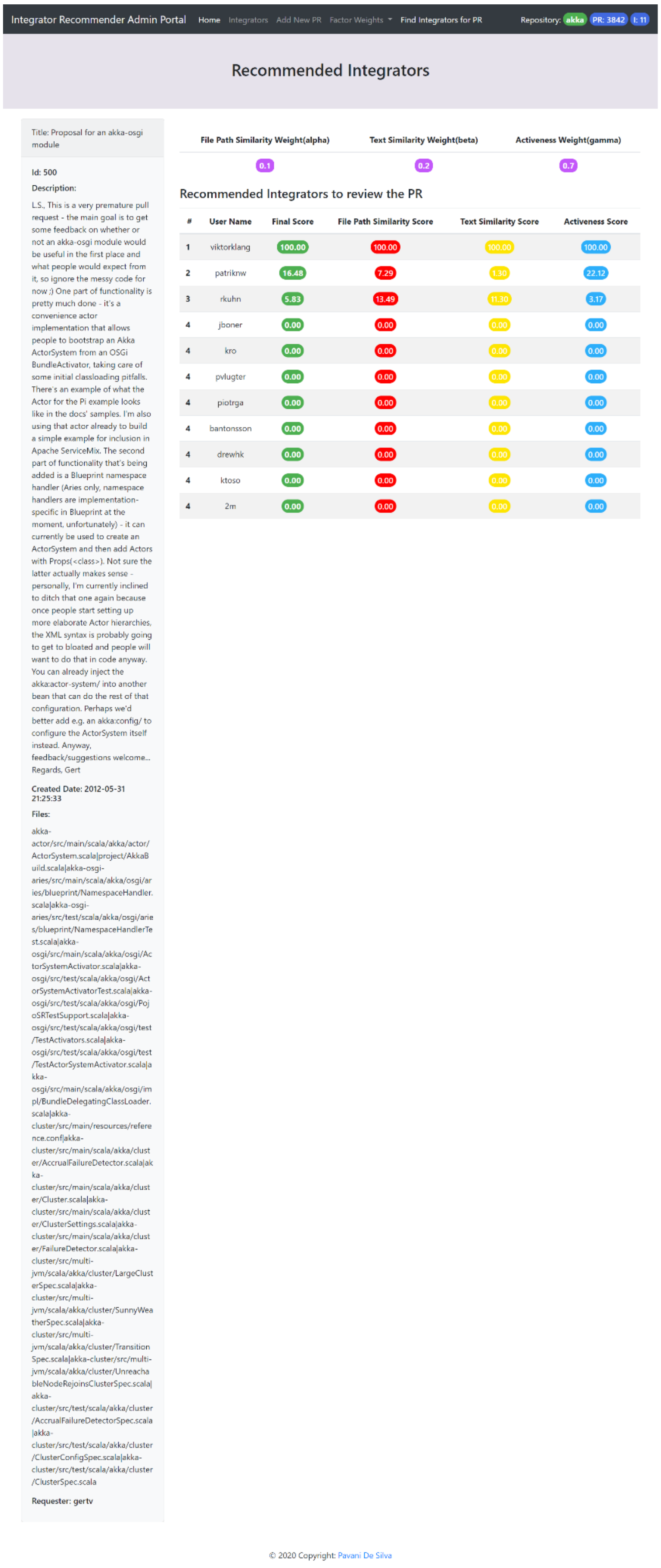
Integrator Recommender Admin Portal Home Integrators Add New PR Factor Weights Find Integrators for PR Repository: akka PR: 3842 I: 11

Integrators

#	Integrator User Name
1	jboner
2	viktorklang
3	rkuhn
4	kro
5	pvlugter
6	piotrga
7	patriknw
8	bantonsson
9	drewhk
10	ktoso
11	2m

© 2020 Copyright: Pavani De Silva

Appendix C – Recommendation of reviewers for PR



Recommended Integrators

Title: Proposal for an akka-osgi module

Id: 500

Description:

L.S., This is a very premature pull request - the main goal is to get some feedback on whether or not an akka-osgi module would be useful in the first place and what people would expect from it, so ignore the messy code for now ;) One part of functionality is pretty much done - it's a convenience actor implementation that allows people to bootstrap an Akka ActorSystem from an OSGi BundleActivator, taking care of some initial classloading pitfalls. There's an example of what the Actor for the Pi example looks like in the docs' samples. I'm also using that actor already to build a simple example for inclusion in Apache ServiceMix. The second part of functionality that's being added is a Blueprint namespace handler (Aries only, namespace handlers are implementation-specific in Blueprint at the moment, unfortunately) - it can currently be used to create an ActorSystem and then add Actors with Props(<class>). Not sure the latter actually makes sense - personally, I'm currently inclined to ditch that one again because once people start setting up more elaborate Actor hierarchies, the XML syntax is probably going to get to bloated and people will want to do that in code anyway. You can already inject the akka:actor-system/ into another bean that can do the rest of that configuration. Perhaps we'd better add e.g. an akka:config/ to configure the ActorSystem itself instead. Anyway, feedback/suggestions welcome... Regards, Gert

Created Date: 2012-05-31 21:25:33

Files:

akka-actor/src/main/scala/akka/actor/ActorSystem.scala|project/AkkaBuild.scala|akka-osgi-aries/src/main/scala/akka/osgi/aries/blueprint/NamespaceHandler.scala|akka-osgi-aries/src/test/scala/akka/osgi/aries/blueprint/NamespaceHandlerTest.scala|akka-osgi/src/main/scala/akka/osgi/ActorSystemActivator.scala|akka-osgi/src/test/scala/akka/osgi/ActorSystemActivatorTest.scala|akka-osgi/src/test/scala/akka/osgi/PojoSRTestSupport.scala|akka-osgi/src/test/scala/akka/osgi/test/TestActivators.scala|akka-osgi/src/test/scala/akka/osgi/test/TestActorSystemActivator.scala|akka-osgi/src/main/scala/akka/osgi/impl/BundleDelegatingClassLoader.scala|akka-cluster/src/main/resources/reference.conf|akka-cluster/src/main/scala/akka/cluster/AccrualFailureDetector.scala|akka-cluster/src/main/scala/akka/cluster/Cluster.scala|akka-cluster/src/main/scala/akka/cluster/ClusterSettings.scala|akka-cluster/src/main/scala/akka/cluster/FailureDetector.scala|akka-cluster/src/multi-jvm/scala/akka/cluster/LargeClusterSpec.scala|akka-cluster/src/multi-jvm/scala/akka/cluster/SunnyWeatherSpec.scala|akka-cluster/src/multi-jvm/scala/akka/cluster/TransitionSpec.scala|akka-cluster/src/multi-jvm/scala/akka/cluster/UnreachableNodeRejoinsClusterSpec.scala|akka-cluster/src/test/scala/akka/cluster/AccrualFailureDetectorSpec.scala|akka-cluster/src/test/scala/akka/cluster/ClusterConfigSpec.scala|akka-cluster/src/test/scala/akka/cluster/ClusterSpec.scala

Requester: gertv

File Path Similarity Weight(alpha)	Text Similarity Weight(beta)	Activeness Weight(gamma)
0.1	0.2	0.7

Recommended Integrators to review the PR

#	User Name	Final Score	File Path Similarity Score	Text Similarity Score	Activeness Score
1	viktorklang	100.00	100.00	100.00	100.00
2	patriknw	16.48	7.29	1.30	22.12
3	rkuhn	5.83	13.49	11.30	3.17
4	jboner	0.00	0.00	0.00	0.00
4	kro	0.00	0.00	0.00	0.00
4	pvlugter	0.00	0.00	0.00	0.00
4	piotrga	0.00	0.00	0.00	0.00
4	bantonsson	0.00	0.00	0.00	0.00
4	drewhk	0.00	0.00	0.00	0.00
4	ktoso	0.00	0.00	0.00	0.00
4	2m	0.00	0.00	0.00	0.00

© 2020 Copyright: Pavani De Silva