



<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *42nd ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2020, Online, South Korea, 27 June 2020 through 19 July 2020*.

Citation for the original published paper:

Strand, A., Gunnarson, M., Britto, R., Usman, M. (2020)

Using a context-aware approach to recommend code reviewers: findings from an industrial case study

In: *Proceedings - International Conference on Software Engineering*, 3381365 (pp. 1-10). IEEE Computer Society

<https://doi.org/10.1145/3377813.3381365>

N.B. When citing this work, cite the original published paper.

© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:bth-20573>

Using a Context-Aware Approach to Recommend Code Reviewers: Findings from an Industrial Case Study

Anton Strand
Ericsson AB, Sweden.
anton.strand@ericsson.com

Ricardo Britto
Ericsson AB, Sweden.
Blekinge Institute of Technology, Sweden.
ricardo.britto@ericsson.com

Markus Gunnarsson
Ericsson AB, Sweden.
markus.gunnarsson@ericsson.com

Muhmmad Usman
Blekinge Institute of Technology, Sweden.
muhammad.usman@bth.se

ABSTRACT

Code reviewing is a commonly used practice in software development. It refers to the process of reviewing new code changes before they are merged with the code base. However, to perform the review, developers are mostly assigned manually to code changes. This may lead to problems such as: a time-consuming selection process, limited pool of known candidates and risk of over-allocation of a few reviewers. To address the above problems, we developed Carrot, a machine learning-based tool to recommend code reviewers. We conducted an improvement case study at Ericsson. We evaluated Carrot using a mixed approach. we evaluated the prediction accuracy using historical data and the metrical Mean Reciprocal Rank (MRR). Furthermore, we deployed the tool in one Ericsson project and evaluated how adequate the recommendations were from the point of view of the tool users and the recommended reviewers. We also asked the opinion of senior developers about the usefulness of the tool. The results show that Carrot can help identify relevant non-obvious reviewers and be of great assistance to new developers. However, there were mixed opinions on Carrot's ability to assist with workload balancing and the decrease code review lead time.

ACM Reference Format:

Anton Strand, Markus Gunnarsson, Ricardo Britto, and Muhmmad Usman. 2019. Using a Context-Aware Approach to Recommend Code Reviewers: Findings from an Industrial Case Study. In *Proceedings of International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Code review is one of the central activities in assuring quality of the code in most software development environments [17, 19, 23, 24]. Modern code review (MCR) is a lightweight approach wherein the review process is conducted with the support of dedicated tools, such as Gerrit and GitHub.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE, May 23–29, 2020, Seoul, South Korea

© 2019 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

In MCR, the code is reviewed before it is pushed to a repository. Integrating MCR in the development process improves maintainability of the software and also helps in reducing bugs [19, 24]. It also helps to share knowledge among the involved developers [3, 17, 19, 21].

Selecting a suitable reviewer in a large project can be both challenging and time-consuming [6, 19, 24, 27]. In general, developers select reviewers from a small pool of people they know [23]. If they are not sure about who should review a given code change, they spend more time trying to identify a suitable review. This is even more critical in the case of newcomers, who do not know everyone in a project and may require even more time to identify adequate reviewers [7].

The limited knowledge about existing code in a project limits the extent to which newcomers can review code. If a newcomer without enough knowledge is asked to review a given code change, it may let defects slip through the review process. Although this may happen even with the experienced reviewers, the more knowledge, the lower the risk of defects not being captured during a code review session [27].

Another consequence of inadequate code reviewer selection is unbalanced review load. When selecting code reviewers from a small group of known developers, it is possible that the same reviewers are selected repeatedly. The repeated assignment of code changes to the same reviewers may lead to a situation wherein just a few developers review most of the code changes.

The previously mentioned problems have two main consequences: i) the time spent finding a reviewer not only costs development time but can also increase the lead time in the review process. A delay in the review process can be seen as a minor inconvenience, but many occurrences could delay the entire project; ii) the risk of choosing unsuitable reviewers may result in more post-delivery defects, due to unsatisfactory code making it through the review process [3, 18].

One way to mitigate the aforementioned problems is by fully or partially automating the code reviewer selection process. This may lead to more effective and efficient code review processes. Furthermore, it may foster balanced review load by considering more developers as potential reviewers.

In this paper, we report the findings from developing, deploying, using, and validating a tool called Carrot, which partially automates

the code reviewer selection process. Carrot recommends code reviewers using a machine learning-based approach (context-aware collaborative filtering).

The following research questions are answered in this paper:

- RQ1 - What are the factors that relate to the suitability of a developer to review a given code change?
- RQ2 - How well does Carrot perform recommendations?
 - RQ2.1 - What is the predictive performance?
 - RQ2.2 - How accurate are the recommendations?
- RQ3 - What is the developer’s perception of the feasibility of using Carrot to recommend code reviewers in large-scale projects?
 - RQ3.1 - How useful is Carrot to recommended reviewers?
 - RQ3.2 - What is Carrot’s ability to help decrease the lead time for reviews?
 - RQ3.3 - What is Carrot’s ability to help balance workload?

We have contributed in the following ways:

- We have deployed Carrot in a real project and used the input from real users to validate its predictive performance and usefulness.
- Carrot is the first code reviewer recommendation tool that uses context-aware collaborative filtering, which has outperformed other recommendation approaches in other domains.
- Finally, Carrot is the first code reviewer recommendation tool that accounts for reviewer workload and makes an attempt to foster balanced review load.

The reminder of this paper is organized as follows: Section 2 contains the related work. Section 3 presents the research design employed in our research. Section 4 contains the description of Carrot. Section 5 presents and discusses the results of Carrot’s validation. Section 6 presents threats to the validity of our results. Finally, Section 7 presents our conclusions and vision on future work.

2 RELATED WORK

Recently, Badampudi et al. [4] conducted a systematic mapping study on the state of modern code review. While they identified that different topics are covered by existing literature, many papers address the problem of recommending code reviewers. Most papers focus on proposing tools to recommend reviewers (often using machine learning techniques) and validate their approaches using historical data extracted from open source projects.

For example, Thongtanunam et al. [23] proposed an algorithm (FPS) for selecting appropriate reviewers by calculating a similarity value of file paths from previous reviews [23]. In a later paper, Thongtanunam et al. [24] extended their previous work with an improved comparison-functionality and developed a tool called RevFinder.

Xin et al. [25] proposed a tool that uses file’s location as input. They proposed analyzing the available description field to get a better understanding of what a code change is about. Additionally, their approach compares the identical number of components in the file-path string independent of its order.

With a focus on the time it might take to review a patch, Zanjani et al. [27] developed the tool chRev. Their tool combines the relevance of similar files, with the frequency and recency of previous reviews.

Xia et al. [26] proposed an approach to catch implicit relations between reviewers and changes. The basis of the algorithm is to replace a new change request with several similar closed change requests. An explicit score is calculated using the replacements per reviewers based on their participation. Additionally, a score is evaluated from the implicit connection to relevant reviews through what they call an Singular Value Decomposition-like (SVD) lower rank decomposition approach.

Ouni et al. [19] focused on social connections. They the connections between a submitter and reviewer and a reviewer’s knowledge associated with similar code snippets as input. Then, the recommendations are done using a search-based approach (genetic algorithms) to recommend reviewers.

Jiang et al. [10] developed CoreDevRec, a tool that uses file paths as input and a Support Vector Machine-based approach to recommend code reviewers. Their approach is only able to recommend core members of a given project.

A different approach using set operations is suggested by Mikolaj et al. [8] proposed a profile-based approach. A developer’s profile is a set that contains all folders and files that the developer has interacted. To make the recommendations for a given code change, all existing profiles are compared (using Jaccard coefficient and Tversky index) with the files and folders of the change, which results in a score for each developer.

Kovalenko et al. [14] implemented an Information retrieval-based recommendation system that was deployed in two companies (Microsoft and JetBrains). They analyzed the performance of their approach using data from production environments. Furthermore, they evaluated the usefulness of their approach through semi-structured interviews and questionnaires. The results indicate that the recommendations are often perceived as relevant and effort saving.

Peng et al. [20] extracted data associated with open source projects through archival research, interviews, and questionnaires to identify the usefulness of Facebook Mention bot, a code reviewer recommendation tool. They found that most developers that the tool help them saving time to identify adequate reviewers. However, they also found that most developers believe that the tool leads to unbalanced review work load.

The main limitations of existing literature can be summarized as follows:

- Existing literature only uses data from open source projects to validate code reviewer recommendation approaches (the only exception is Kovalenko et al. [14]).
- Most studies have not validated their code reviewer recommendation approaches in production or collected the opinion of developers about those approaches (the exceptions are Kovalenko et al. [14] and Peng et al. [20]).
- The validation of existing approaches are done using accuracy metrics such as precision, recall, and Mean Reciprocal Rank (MRR). They compare a list of reviewers recommended by their proposed approach with the list of actual

reviewers in the extracted data (from open source projects in most cases). If this type of validation is not possible to know whether the actual reviewers were adequate to review the pushed code or if a recommended reviewer not part of the actual reviewers' list was adequate or not.

- Most approaches focus on getting the best reviewer “on paper” and rarely take into account the workload of potential reviewers. A possible implication of this greedy approach is the high review workload of a few developers. Furthermore, the more reviews a developer does, the better match for future reviews a developer will be, leading to a “vicious circle”.
- Context-aware approaches have not been used to recommend code reviewers. This type of approach is able to handle the limitations of traditional recommendation systems (e.g., the cold start problem). It has also been successfully used in other domains (e.g., fashion recommendation [15]).

With the investigation reported in this paper, we aimed at address the aforementioned limitations of existing research. Carrot is recommendation tool that uses a context-aware approach to recommend code reviewers. Our tool also makes an attempt to balance the workload of code reviewers. Furthermore, we have deployed Carrot in a real project in Ericsson. Finally, we evaluated Carrot's performance, usefulness, and feasibility by means of feedback provided by Ericsson developers.

3 RESEARCH DESIGN

In this section, we describe the research design used to conduct the research. We conducted an improvement case study [22], which involved a literature review, semi-structured interviews and questionnaires as data collection methods.

3.1 The case and unit of analysis

The case company is Ericsson¹. The case product is BSS (Business Support System) comprising 30 subsystems/products. The system was deployed in production for the first time in the second semester of 2018 but have been in development for more than half a decade. The development of this product involves more than 1000 employees distributed in centers of excellence located in Sweden, India, Canada, and Brazil.

The products are developed by multiple teams with different responsibilities, often directly connected to a few repositories. The same repository can be *owned* by multiple teams. The teams have from 5 to 8 members and use agile practices in their daily work. There are many differences in how the teams operate in regards to Gerrit changes. The most impacting difference is the way they currently add reviewers; some teams have the policy to add the entire team as reviewers; some go by areas of responsibility (sometimes unknown by a developer); and others choose reviewers more freely inside the same community, (a large pool of developers). Another difference is the specific role of the design lead and how involved they are in all aspects of the code base. In some cases, anyone can review the change, and the design lead just gets involved if there is a need. For other cases, a design lead should (very much preferred)

review a particular part. Depending on how critical a repository is and what the specific change entails, it might be necessary or not.

Additionally, some teams have implemented Gerrit hooks that when triggered, add a pre-made review list. This is often by adding +1 review themselves to tell the other developers the change is ready for review. The different review processes are the main reason that some of the evaluations are split into two for the two subsystems.

3.2 Research Approach

We used the following six step approach to carry out the reported research:

- (1) **Factor identification** - consists of the literature review as well as the complementary interviews with experienced developers to discuss and find suitable factors.
- (2) **Factor selection** - is the process of filtering the identified factors, looking at the availability and complexity; which is based on an exploratory analysis of the available data including the thoughts and ideas collected from the experienced developers.
- (3) **Approach selection** - consists of finding a suitable recommendation approach that can work with the desired factors previously identified. This consists of identifying possible machine learning approaches that are viable and can accept the data.
- (4) **Predictive performance** - it is calculated with the help of Mean Reciprocal Rank (MRR) to give an early sanity check of the chosen algorithm.
- (5) **Recommendation suitability** - it is the feedback collected through a questionnaire when the developer request a recommendation. The developers provide their feedback on the suitability of the recommended candidates.
- (6) **Questionnaire (candidate)** - it is a set of questions asking the recommended candidate reviewers if they see themselves as suitable reviewers for the given change.
- (7) **Questionnaire (feasibility)** - it is the questions for the static validation [9] of the tool, which is used to evaluate the real feasibility of Carrot.

MRR is calculated with the following Equation, where Q means the number of times recommendations were requested (queries) and $rank_i$ refers to the position wherein the first relevant recommended item appears for the i -th query. The result is between 1.0 and 0.0, where 1.0 means that all recommendations had an actual reviewer on the first position and 0.0 means that they did not present any actual reviewer.

$$MRR_{score} = \frac{1}{|Q|} * \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (1)$$

3.3 Data Collection

We used multiple data collection methods in this study including a literature review, semi-structured interviews, repository mining and two questionnaires (see Table 1 for more details).

We describe the details of the interviewees and questionnaire participants in the following:

¹www.ericsson.com

- Interviews - We conducted six semi-structured face-to-face on-site interviews of experienced software developers. The interviewees were selected by the Unit Manager. The average length of the interviews was about half-hour. The experience of the interviewed developers varied from four to ten years (average experience of 7.5 years).
- Questionnaire (accuracy) - To see how the developers using Carrot view its recommendations with regards to accuracy, we administered a questionnaire inside the Carrot tool (see likert scale options against recommended candidates in Figure 1). Through this questionnaire, we collected the opinion of the developers to see how strongly they (dis)agree with Carrot’s recommendations. During the validation period consisting of two weeks, Carrot was used to make 47 recommendation requests.
- Questionnaire (candidate) - It was aimed to collect the feedback of the candidates recommended by Carrot as reviewers to ascertain the suitability of the recommendations. After deployment and testing of Carrot, this questionnaire was sent to a sample of 36 developers that were recommended by Carrot as reviewers for the ten randomly selected changes. In response, 26 developers answered the questionnaire.
- Questionnaire (feasibility) - To test the feasibility of the Carrot as a recommendation tool, the developers who used Carrot during the testing period were asked to provide their feedback. This questionnaire was answered by 8 developers.

3.4 Data Analysis

To analyze our data, we compiled a factor list for RQ1 from the literature review and the interviews. We used historical data to control Carrot’s recommendation for RQ2.1. Lastly, we used basic descriptive statistics to identify the trends in the questionnaires’ responses corresponding to RQ2.2 and RQ3.

4 CARROT OVERVIEW

Carrot is a recommendation system that utilizes a hybrid approach (context aware) for performing recommendations. To use it, the user provides a valid code change id. As a result, Carrot provides a list with the top 5 reviewers and their respective review workload. It also allows the users to provide feedback on the recommendations. In the remainder of this section, we provide more details about Carrot architecture and its associated micro services.

4.1 Architecture

Carrot was developed using python 3.6 and is composed by five microservices, which are deployed using Docker containers, uWSGI, and Nginx. The tool can be used either via the available Flask-based GUI (see Figure 1) or REST API calls.

Figure 2 shows an overview of Carrot’s architecture. The dotted lines represent data channels, and the other lines are bi-directional communication channels for requests and response data. The blue boxes represent the microservices, while the red box represents the intermediate storage where the ML model is located (a pickle object²).

²docs.python.org/3/library/pickle.html

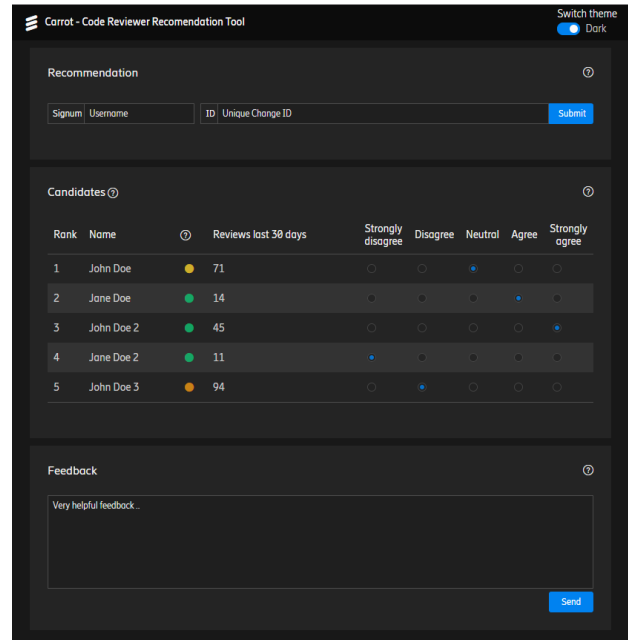


Figure 1: The Carrot web interface.

The **Core** microservice orchestrates the operation of the other services. It also handles the functionality used to obtain the feedback from Carrot’s users about the recommendations.

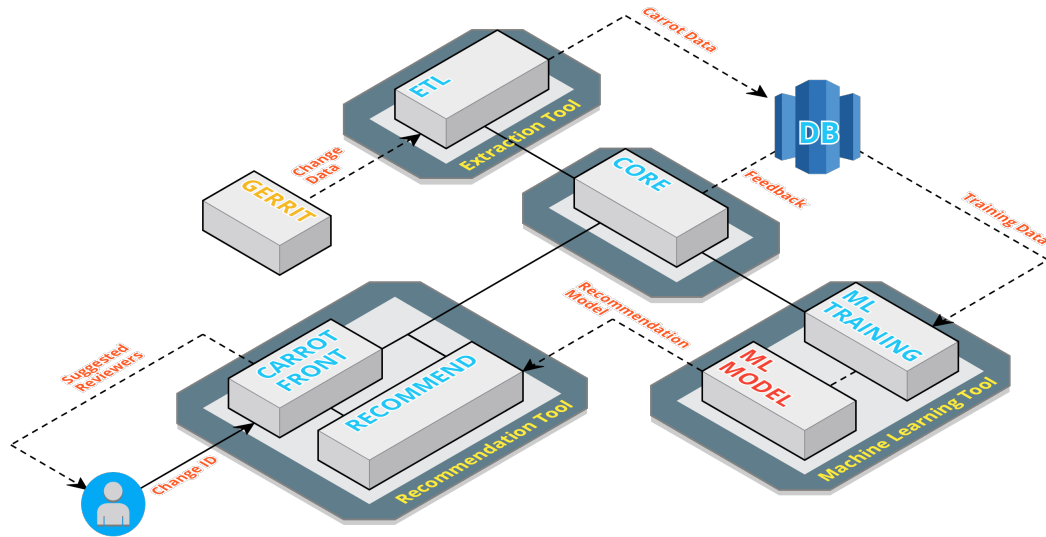
The **ETL** microservice performs the data extraction, transformation and loading of the data. It has two responsibilities: i) extract data (code changes and the associated authors and reviewers) from the version control system (Gerrit in our case) every day and stores it in a database; ii) extract the data (from Gerrit) associated with an input code change that a user expects recommendations from Carrot. Note that this module also transforms the data to the format required by the ML microservice.

The **ML** microservice uses the data extracted by the ETL microservice and trains (once per day) a model to recommend code reviewers (see Section for more details). The trained model is stored as a pickle file. Note that Carrot considers both code changes and code reviews to train the model. Carrot accounts for code review data explicitly since the ETL microservice extracts this type of data through the Gerrit API. Code changes are accounted for by means of the self-reviews done by the developers. In the studied case, the author of a given code change is required to review its own code in addition to other developers. As a consequence, even if a developer has not reviewed a file modified by some other developer, if s/he has modified the file before, it will have self-reviewed his/her change and this will be considered when training the ML model.

Recommend is the microservice that loads the trained model to perform a recommendation for a given code change. It loads the trained model and requests the ETL (via the Core) to obtain the data (file paths and repository) associated with the given code change id. After, it splits the file paths into directories and file extensions. The repository name, along with the folders and file extensions are provided as input to the trained model. As a result,

Table 1: Data collection methods used in the study

Data source	Description	Questions
Literature review	Code review factors identified in related and previous work.	RQ1
Semi-structured interviews	Complementary information on the factor identification from literature. Includes additional identified factors as well as ideas and limitations on the already discovered ones.	RQ1
Repository mining	The data from Gerrit required to perform the initial exploratory analysis, but also the factor selection for Carrot. This data is also responsible for training the recommendation model.	RQ2.1
Questionnaire (candidate)	Feedback on the suitability of the recommendation from the candidate's point of view.	RQ2.2
Questionnaire (accuracy)	A scale of the suitability of the recommendations from the developer requesting the recommendation.	RQ2.2
Questionnaire (feasibility)	Feedback on the usefulness of Carrot from the developer's point of view.	RQ3

**Figure 2: Carrot's architecture.**

the trained model predicts a score for each developer. This score reflects the suitability of each developer to review the given code change. Then, the developers are sorted by the calculated scores (descending order).

The final step carried out by Recommend is the calculation of the code review workload associated with the developers. The workload is measured in two ways: i) the number of reviews conducted

in the last 30 days by each developer; ii) how many standard deviations the number of reviews conducted by a developer is from the mean number of reviews conducted by all developers in the last six months. Candidates with less than 0 reviews in the last 30 days are removed from the list. This is done to avoid recommending someone that may be on vacation or on leave. Finally, Carrot recommends the top five developers from the resulting list.

The **Carrot-Front** microservice provides the front-end of Carrot. This microservice enables the users to request code reviewer recommendations, along with an indication of their respective review workloads (a review counter and a color indicator). The color indicator can have three different values: i) red, if the review workload is equal or above three standard deviations from the mean (high load); yellow, if the review workload is between two and three standard deviations from the mean (medium load); iii) green otherwise (low load). These thresholds are commonly used to identify abnormalities in data.

Another responsibility of Carrot-Front is to provide a feedback form for Carrot’s users. The users can grade the correctness of each recommended reviewer using a 5-point scale, wherein 1 means strongly disagree and 5 means strongly agree. The users can also provide free text-feedback. Note that we used the data gathered through this feedback form to evaluate Carrot.

4.2 Recommendation Engine

The recommendations provided by Carrot are calculated using a context-aware approach. More specifically, we implemented the LightFM algorithm [15] using the python library provided by the author³. LightFM combines both collaborative filtering [1] and context-based filtering [1]. We decided to implement this hybrid approach because it handles well the so-called cold start problem and sparse interactions [15]. Furthermore, it has presented good performance [15] in recommendation contexts similar to the one addressed in this paper. When recommending code reviewers, the cold start problem happens when new files are added to the repositories, i.e. it is not possible to directly link the new files to the potential reviewers. Furthermore, the interaction between files and potential reviewers is sparse since not all files are modified or reviewed by all reviewers.

Collaborative filtering uses the collaborative power of the interactions between users (code changes in our case) and items (potential reviewers in our case provided by to make recommendations. It is often the case that not all developers have reviewed all files in a repository. This means that the matrix that relates code changes and developers is sparse (there are many empty cells/relationships). The matrix, called interaction matrix, has dimension $c \times d$, where c means the number of unique changes and d means the number of unique developers who have interacted with at least one of the files in the changes.

Collaborative filtering methods are able to obtain the missing interactions because the existing interactions are frequently highly correlated. The interactions can be obtained via memory-based methods [1] or model-based methods [1]. LightFM uses model-based latent factor collaborative featuring, wherein where the interaction matrix is factorized into two matrices (change latent vector and developer latent vector).

Pure collaborative filtering struggles with the code start problem. LightFM handles this problem by accounting for the meta-data associated with the changes. In Carrot, we used the following meta-data features: directory, repositories, and file extension (see Section 5). The changes are related to the meta-data through a binary matrix of dimension $c \times (d + r + e)$, where c means the number of unique

changes, d means the number of unique directories, r means the number of unique repositories, and e means the number of unique file extensions. In this matrix, 1 means that a given feature is present in a change, while 0 means the absence.

To calculate the score (\widehat{s}_{ik}) that reflects how suitable is a developer k to review code change i , LightFM uses Equation , where q_i means the latent representation of change i , p_k means the latent representation of developer k , b_i represents the bias term of code change i , and b_k means the bias term of developer k . The parameters of this equation are obtained by an optimization approach (more specifically, stochastic gradient descent), which aims at maximising the likelihood of the data conditional on the parameters.

$$\widehat{s}_{ik} = q_i \cdot p_k + b_i + b_k \quad (2)$$

LightFM has eight hyperparameters. Table 2 shows the hyperparameter values we used in Carrot, which are the default values recommended by Kula [15].

Table 2: The hyperparameters used in Carrot.

Hyperparameter	Value
Epsilon	1.09e-08
Learning rate	0.02
Learning schedule	adadelta
Loss	warp
Max sampled	22
No components	40
Rho	5.51e-09
User alpha	8.24e-09

5 RESULTS AND DISCUSSION

In this section, we show the results associated with our research questions.

5.1 RQ1 - Review Factors

Table 3 summarizes the data collected through the literature review and the interviews. The results show that the two prominent factors are: **path usage** and **repository activity**. The **usage of time** and **activeness** is also cited in multiple studies and interviews. The categorization used for the factors is fairly rough, since it would not be viable to use every single option as it’s own category. For example the path similarity under file consists of simple character count, common directory count and more advanced algorithms. The category and factor is a bucket for types that in some way analyze the file paths.

The factors used as input for Carrot are in bold letters in Table 3. To select these factors, we used the following criteria:

- C1 - Is explicitly discussed in the interviews.
- C2 - Is talked about in reviewed literature.
- C3 - Is available through automated extraction (does not require manual input).
- C4 - The extraction complexity is reasonable (extracting the required data is not a non trivial task).

³github.com/lyst/lightfm

Table 3: Factors from literature review, exploratory analysis and interviews.

Category	Factor	Description	Int. Freq	Lit. Freq	Source
Time	Time limit	Limit length of history analyzed and used.	3	2	[16, 25]
	Freshness	Consider new as higher weight or other way around.	1	3	[12, 13, 26]
	Activeness	Time period		2	[11, 27]
		Frequency.		2	[19, 27]
		Recency	3	3	[12, 19, 27]
File	Path	Path comparison or usage in one way or another.	4	7	[11, 12, 16, 23–26]
	Modifications	Who have changed the line previously.	4	1	[5]
Commit	Title	Semantic analysis of commit title.		1	[11]
	Description	Semantic analysis of commit message.		1	[25]
	Change type	Is it a bug, a re-factoring, how complex is it. All these things affect who should make the review.	2		
Social	Network	Build network with comments.	1	2	[11, 19]
	Feedback	Find developers that give helpful and positive feedback.	1	2	[2, 12]
Repository	Activity	The contribution made, number of commits, comments and reviews.	4	4	[12, 13, 16, 27]

- C5 - The analysis complexity is reasonable (data size and comparison is within reason).
 C6 - Is a non punishing attribute (relatively unbiased).
 C7 - The data is reliable (the meaning of the data is trustworthy and does not change).

While it is clear that the tool could be improved in terms of MRR, the goal is not to aim for only a good MRR (instead also consider non-obvious candidates). The score can be seen as a form of baseline or sanity check to establish that the algorithm is on the right track.

5.2 RQ2 - Carrot Performance Evaluation

In this section, we present the results associated with RQ2.

5.2.1 RQ2.1: Evaluation using historical data. To answer RQ2.1, we calculated the MRR. The calculation was performed on the last six months of data (up until 2019-05-03). The model was trained on two years of data with the end date before the testing data. The training and testing data were strictly separated, with both sets being represented in the format of an interaction matrix. The interactions in the testing matrix are the changes that recommendations are performed on with an actual interaction being counted as a positive value in the MRR.

The MRR score of the recommendation system was calculated as 0.37 on the historical data of six months. This score is equivalent to an average rank of 2.7 in the recommendation list. In other words, on average Carrot presents an actual reviewer in rank 2 or 3 of the list of recommendations.

5.2.2 RQ2.2: Perceived accuracy. We evaluated the perceived accuracy of Carrot from two perspectives: developers who used Carrot to find relevant reviewers and candidates recommended by Carrot.

The results (see Table 4) show that the developers, who used Carrot to find relevant reviewers, are in agreement with 78% (strongly agree - 53%, agree - 23%) of Carrot's first recommendations. The results also show that the strong agreement reduces as we move from the first to the fifth recommendation. The higher agreement with the first recommendation is understandable as the tool aims to recommend the best suitable candidate as the top candidate. The results also show a noticeable level of disagreement with Carrot recommendations. The disagreement may be due to the difficulty in recommending the relevant candidate for specialized components. Carrot may not have found the relevant reviewer due to faulty implicit relations used by the algorithm or a failure to draw some connection.

Table 4: Developers’ perceptions regarding Carrot’s recommendations.

Survey question	Strongly agree	Agree	Neutral	Disagree	Strongly disagree
<i>RQ2.3: How accurate are the recommendations? - perception of developers selecting code reviewers (N = 47)</i>					
Agreement with Carrot’s first recommendation	53%	23%	13%	9%	2%
Agreement with Carrot’s second recommendation	47%	19%	13%	17%	4%
Agreement with Carrot’s third recommendation	28%	26%	23%	21%	2%
Agreement with Carrot’s fourth recommendation	34%	19%	21%	23%	2%
Agreement with Carrot’s fifth recommendation	43%	15%	23%	17%	2%

Percentages in the table are rounded off to the nearest ten.

Table 5 presents the perceptions of the candidates recommended by Carrot. The results show that a majority of the recommended candidates in our questionnaire sample (96% in total) felt comfortable in reviewing the given change, indicating a very high suitability of the recommendations. To further elaborate, the candidates were also asked to inform if they have previously worked on the same file. The results show that 74% of the candidates shared that they had worked with the same file before. For example, see some quotes below:

“Already worked in this part of code.”

“It is in the repository that I am the owner of.”

“It’s front-end related code and it’s the area I sit most with.”

It is important to note that when files are added for the first time, they would not have direct dependencies at that stage. Therefore, besides experience of working in the same file, we also asked candidates to share if they have previously worked on similar files. We found that 89% of the candidates in our sample had worked with similar files before the given change. Furthermore, all candidates noted (both strongly agree and agree responses together) that they were familiar with the language used in the change, and they could also understand what the code aimed to do. See, for example, some quotes from the recommended candidates:

“I’m familiar with the use case, and I have a good understanding of the contents of the change.”

“I’m familiar with the background and understand the contents of the change.”

“I’m very familiar with the code area affected by the change.”

Looking at the candidates responses together, the results show that majority of the respondents had positive perception about Carrot’s recommendations.

5.3 RQ3 - Carrot feasibility

The question aims to validate the feasibility of Carrot as a suitable code reviewer recommendation tool. We performed this validation by sending a questionnaire (see Table 6) to experienced reviewers. Eight reviewers ranked Carrot with regards to its usefulness and ability to improve the lead time and workload.

5.3.1 RQ3.1. The results (see Table 6) show that the majority of the respondents (75%) agree or strongly agree that Carrot is useful to select relevant reviewers. Some reviewers may not be obvious, but are still relevant to review the change. It is relatively more difficult to find such reviewers. We asked the participants to judge

Carrot’s ability to find such non-obvious but relevant reviewers. The results show that over 60% respondents think that Carrot helped in identifying non-obvious reviewers that are still relevant to review the change. Furthermore, about 87% of the respondents have a positive perception (both agree and strongly agree) about Carrot’s ability to help new developers in finding the relevant reviewers.

5.3.2 RQ3.2. The results (see Table 6) show that the majority of the respondents do not think that Carrot helps in decreasing the lead time for code reviews. There is currently no known explanation for this sentiment. An important note is that the tool in its current use requires a developer to manually enter the web page, copy and past the change ID and then copy one or more names into Gerrit to add them as reviewers.

5.3.3 RQ3.3. The results (see Table 6) show that only half of the respondents agree that Carrot is helpful in balancing the workload of reviewers. No direct explanations were provided in the answers, but comments like the following one indicate that some changes need to be reviewed by a specific set of developers (i.e. not many developers to balance among).

“Bumps are primarily handled by build masters, unless they contain special changes.”

Each team has a design lead, who is an experienced developer having good knowledge about the product’s architecture. It would not be possible to balance for the changes that needed to be reviewed by the design leads (similar to build masters).

6 THREATS TO VALIDITY AND LIMITATIONS

The validity threats associated with our investigation are discussed using the categories reliability, internal, construct and external validity described by Runeson and Höst [22].

Reliability is about the extent to which the collected data and analysis are dependent on the specific researchers[22]. We involved multiple researchers during all phases of the study to minimize the potential reliability related validity concerns. The first two authors developed the case study protocol, interview guide and questionnaire iteratively based on the feedback of the third author. Data collection was performed by the first two authors. The fourth author reviewed and contributed to the data analysis, which was also lead by the first two authors.

Internal validity relates to the confounding factors that could impact the validity of the result [22]. We identified the factors using a literature review and interviews. It is possible that we missed out

Table 5: Recommended candidates’ perceptions regarding Carrot’s recommendations.

Survey question	Strongly agree	Agree	Disagree	Strongly disagree
<i>RQ2.3: How accurate are the recommendations? - perception of reviewers recommended by Carrot (N = 26)</i>				
I feel comfortable reviewing the given change	63%	33%	-	4%
I have previously worked with the files in the change	N/A	74%	26%	N/A
I have previously worked with files similar to the ones in the change	N/A	89%	11%	N/A
I am familiar with the language used in the change	74%	26%	-	-
I understand what the code aims to do in the change	67%	33%	-	-
Percentages in the table are rounded off to the nearest ten.				
N/A means the response option (i.e., strongly agree or disagree) is not relevant for this question.				

Table 6: Experienced reviewers’ perception regarding Carrot’s feasibility.

Survey question	Strongly agree	Agree	Neutral	Disagree	Strongly disagree
<i>RQ3.1: How useful is Carrot to recommend reviewers? (N=8)</i>					
Carrot is useful to select adequate code reviewers	25%	50%	-	12.5%	12.5%
Carrot can recommend non-obvious relevant reviewers	-	62.5%	-	37.5%	-
Carrot will help new developers to find relevant reviewers	12.5%	75%	-	12.5%	-
<i>RQ3.2: What is Carrot’s ability to help decrease the lead time for reviews? (N=8)</i>					
Carrot will help reduce lead time for the code reviews	0	37.5%	0	50%	12.5%
<i>RQ3.3: What is Carrot’s ability to help balance the workload of reviewers? (N=8)</i>					
Carrot will help to balance the reviewers’ workload	0	50%	0	37.5%	12.5%

some relevant studies in the literature review. We performed interviews of six developers having varying experience level to gather as many relevant factors as possible. We used three questionnaires to collect data from the study participants during the study. Furthermore, the questionnaires were sent to only relevant respondents to avoid getting incorrect answers. For example, the questionnaire about recommended candidate’s perceptions about Carrot’s feasibility, was only sent to the candidates that were recommended by Carrot.

External validity concerns the extent that the findings can be generalized and how interesting they are to people outside of the investigated case [22]. We performed the case study in a particular context (i.e., at Ericsson), and therefore the results are not generalizable beyond the studied case. However, our empirical approach that we used to evaluate Carrot would be interesting to other researchers working in the same area.

Construct validity concerns the extent of how well the operational measures represent and answer the research questions [22]. A possible threat is the misinterpretation of the questions in the interviews. At the start of the interviews we clarified the purpose of the interview and how the information will be used. Regarding the questionnaire, We used an iterative process to design and improve the questionnaires instrument in order to avoid ambiguities. However, it is still possible that some respondents misunderstood some parts in our questionnaires.

7 CONCLUSIONS AND FUTURE WORK

In this work, we conducted an improvement case study to address challenges associated with identifying relevant code reviewers. We developed a recommendation tool Carrot using a machine learning based approach (context aware collaborative filtering). We deployed

the tool for a testing period of two week during which Carrot was used to recommend reviewers for a total of 47 changes. For majority of these changes, the involved developers were in agreement with Carrot’s first recommendations. Furthermore, most of the recommended candidates shared that they have either worked in the same or similar file before or understand what the code aims to do. Overall, most of them felt comfortable in reviewing the change. During the static validation with 8 experienced code reviewers, Carrot was not found to be helpful in reducing the lead time and balancing the workload of the reviewers. In our future work we aim to conduct a long-term study to investigate the impact of using Carrot on the lead time of the code reviews and also its ability to balance the workload of the reviewers. We also to plan to integrate Carrot in the code review tool Gerrit, which is used in Ericsson, to further improve its usefulness, and to see if it helps in reducing the code review lead time.

ACKNOWLEDGMENTS

The authors are very thankful to all Ericsson developers who provided helped us identifying the relevant factors to recommend adequate reviewers and also provided their feedback in relation to Carrot.

REFERENCES

- [1] Charu C. Aggarwal. 2016. *Recommender Systems: The Textbook* (1st ed.). Springer Publishing Company, Incorporated.
- [2] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. SentiCR: A customized sentiment analysis tool for code review interactions. *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), 106–111. <https://doi.org/10.1109/ASE.2017.8115623>
- [3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. *Proceedings - International Conference on Software Engineering* (2013), 712–721. <https://doi.org/10.1109/ICSE.2013.6606617>

- [4] Deepika Badampudi, Ricardo Britto, and Michael Unterkalmsteiner. 2019. Modern Code Reviews - Preliminary Results of a Systematic Mapping Study. In *Proceedings of the Evaluation and Assessment on Software Engineering (EASE '19)*. ACM, New York, NY, USA, 340–345. <https://doi.org/10.1145/3319008.3319354>
- [5] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. *Proceedings - International Conference on Software Engineering* (2013), 931–940. <https://doi.org/10.1109/ICSE.2013.6606642>
- [6] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: which problems do they fix? *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014* (2014), 202–211. <https://doi.org/10.1145/2597073.2597082>
- [7] Ricardo Britto, Daniela S. Cruzes, Darja Smitte, and Aivars Sablis. 2018. Onboarding software developers and teams in three globally distributed legacy projects: A multi-case study. *Journal of Software: Evolution and Process* 30, 4 (2018), 1–17. <https://doi.org/10.1002/smr.1921>
- [8] Mikolaj Fejzer, Piotr Przymus, and Krzysztof Stencel. 2018. Profile based recommendation of code reviewers. *Journal of Intelligent Information Systems* 50, 3 (2018), 597–619. <https://doi.org/10.1007/s10844-017-0484-1>
- [9] T. Gorschek, P. Garre, S. Larsson, and C. Wohlin. 2006. A Model for Technology Transfer in Practice. *IEEE Software* 23, 6 (Nov 2006), 88–95. <https://doi.org/10.1109/MS.2006.147>
- [10] Jing Jiang, Jia Huan He, and Xue Yuan Chen. 2015. CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation. *Journal of Computer Science and Technology* 30, 5 (2015), 998–1016. <https://doi.org/10.1007/s11390-015-1577-3>
- [11] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. 2017. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. *Information and Software Technology* 84 (2017), 48–62. <https://doi.org/10.1016/j.infsof.2016.10.006>
- [12] Aleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. *ICSE: International Conference on Software Engineering* (2016), 1028–1038. <https://doi.org/10.1145/2884781.2884840>
- [13] Aleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating code review quality: Do people and participation matter? *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings* (2015), 111–120. <https://doi.org/10.1109/ICSM.2015.7332457>
- [14] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasyukov, Christian Bird, and Alberto Bacchelli. 2018. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering* (2018).
- [15] Maciej Kula. 2015. Metadata Embeddings for User and Item Cold-start Recommendations. In *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th {ACM} Conference on Recommender Systems (RecSys 2015)*, Vienna, Austria, September 16–20, 2015. (*CEUR Workshop Proceedings*), Toine Bogers and Marijn Koolen (Eds.), Vol. 1448. CEUR-WS.org, 14–21. <http://ceur-ws.org/Vol-1448/paper4.pdf>
- [16] Jakub Lipčák. 2017. Optimal Recommendations for Source Code Reviews. *Masaryk University* (2017).
- [17] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality Categories and Subject Descriptors. *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)* (2014), 192–201. <https://doi.org/10.1145/2597073.2597076>
- [18] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189. <https://doi.org/10.1007/s10664-015-9381-9>
- [19] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2017. Search-based peer reviewers recommendation in modern code review. *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (2017), 367–377. <https://doi.org/10.1109/ICSME.2016.65>
- [20] Zhenhui Peng, Jeehoon Yoo, Meng Xia, Sunghun Kim, and Xiaojuan Ma. 2018. Exploring How Software Developers Work with Mention Bot in GitHub. In *Proceedings 6th International Symposium of Chinese CHI*. ACM, 152–155.
- [21] Peter C. Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* (2013), 202. <https://doi.org/10.1145/2491411.2491444>
- [22] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. <https://doi.org/10.1007/s10664-008-9102-8> arXiv:gr-qc/9809069v1
- [23] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 2014. Improving code review effectiveness through reviewer recommendations. *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering - CHASE 2014* (2014), 119–122. <https://doi.org/10.1145/2593702.2593705>
- [24] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken Ichi Matsumoto. 2015. 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 2015 - Proceedings* (2015), 141–150. <https://doi.org/10.1109/SANER.2015.7081824>
- [25] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings* (2015), 261–270. <https://doi.org/10.1109/ICSM.2015.7332472>
- [26] Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, and Xudong Liu. 2017. A hybrid approach to code reviewer recommendation with collaborative filtering. *SoftwareMining 2017 - Proceedings of the 2017 6th IEEE/ACM International Workshop on Software Mining, co-located with ASE 2017* (2017), 24–31. <https://doi.org/10.1109/SOFTWAREMINING.2017.8100850>
- [27] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543. <https://doi.org/10.1109/TSE.2015.2500238>