Front. Comput. Sci., 2025, 19(1): 191202
https://doi.org/10.1007/s11704-023-3256-9

RESEARCH ARTICLE

Code context-based reviewer recommendation

Dawei YUAN[1], Xiao PENG[1], Zijie CHEN[1], Tao ZHANG (✉)[1], Ruijia LEI[2]

1 School of Computer Science and Engineering, Macau University of Science and Technology, Macau 999078, China
2 Faculty of Science, University of Amsterdam, Amsterdam WB 1018, Netherlands

Abstract Code review is a critical process in software development, contributing to the overall quality of the product by identifying errors early. A key aspect of this process is the selection of appropriate reviewers to scrutinize changes made to source code. However, in large-scale open-source projects, selecting the most suitable reviewers for a specific change can be a challenging task. To address this, we introduce the Code Context Based Reviewer Recommendation (CCB-RR), a model that leverages information from changesets to recommend the most suitable reviewers. The model takes into consideration the paths of modified files and the context derived from the changesets, including their titles and descriptions. Additionally, CCB-RR employs KeyBERT to extract the most relevant keywords and compare the semantic similarity across changesets. The model integrates the paths of modified files, keyword information, and the context of code changes to form a comprehensive picture of the changeset. We conducted extensive experiments on four open-source projects, demonstrating the effectiveness of CCB-RR. The model achieved a Top-1 accuracy of 60%, 55%, 51%, and 45% on the Android, OpenStack, QT, and LibreOffice projects respectively. For Mean Reciprocal Rank (MRR), CCB achieved 71%, 62%, 52%, and 68% on the same projects respectively, thereby highlighting its potential for practical application in code reviewer recommendation.

Keywords code reviewer recommendation, code context, pull request

1 Introduction

Software code review is an established best practice in both open source and industrial software projects. It significantly improves programming quality and productivity by enabling formal inspections of design and code. In essence, code review involves developers scrutinizing code changes to assess their quality, identify any potential defects, and make corrections before integration [1]. However, the adoption of code review is not without its challenges due to its time-consuming, complex, and synchronous nature [2]. For example, the Android project experienced over three million submissions between 2012 and 2020, averaging more than a thousand commits per day [3]. Given the sheer scale of such an open-source project, identifying suitable reviewers for Pull Requests from a large pool of potential reviewers presents a significant challenge. Incorrect assignment of code reviewers can exacerbate this issue, prolonging request processing time and reducing project development and maintenance efficiency [4]. With this context in mind, the remainder of this paper focuses on the proposal of a new method for code review which leverages global semantic information to aid programmers in understanding entire pull requests and recommending suitable code reviewers.

However, while code files contain valuable information, redundant information can be detrimental. Code changes represent modified information, but the remaining unmodified lines of code might also provide useful contextual information. Given that files are often large, there is a high potential for redundancy in this contextual information. Consequently, critical information might be lost when converting a long sequence to a fixed-length vector, or it might convey incorrect meanings. In order to make better code reviewer recommendations, we propose a novel deep learning-based approach called **CCB-RR** (Code Context Based Reviewer Recommendation), which incorporates multiple features to recommend reviewers. We divide the model into three sub-networks: File Path Network (utilizing file paths), Context-Aware Network (utilizing contextual and modified information from source files), and Textual Network (utilizing the title and description of pull requests). Owing to the variable size of contextual information, we improve the Context-Aware Network by using KeyBERT [5] to extract keywords from source files and the Byte Pair Encoder (BPE) [6] method to process the code information. In each network, we use the self-attention mechanism module [7] to extract features and obtain global textual information.

Our experiment evaluated CCB-RR on four open-source software systems: Android, OpenStack, Qt, and LibreOffice. The results show that our model improves performance in Top-*k* accuracy and MRR. On average, CCB-RR correctly recommended 87% of reviewers with a Top-10 recommendation. Therefore, on average, CCB-RR achieved a Top-1 accuracy of 55% over the baselines, demonstrating that by using our context-based model, CCB-RR can accurately

Received March 28, 2023; accepted October 13, 2023

E-mail: tazhang@must.edu.mo

recommend code reviewers. Thus, the paper makes the following contributions:

- We introduce a novel code reviewer recommendation approach based on deep learning. Our method represents the first attempt at using context information from source files for automated pull request reviewer recommendation.
- Our design incorporates three sub-networks, each equipped with a self-attention module. These sub-networks are designed to extract features from diverse data types.
- We integrate a comprehensive set of features in our model, including file paths, pull request titles, descriptions, and the potential efforts derived from context analysis and modification information within the source files. This multi-input model is then trained to recommend code reviewers.
- A series of extensive experiments were conducted on a publicly available dataset to evaluate the effectiveness of our CCB-RR model. The results demonstrate that our model substantially outperforms two baseline models. To support open source project, we have made both our dataset and source code accessible to the public. Everyone can access the repository using the keywords `ccb_method` and query the source code with the ID `23664681` on Figshare.

This paper is organized as follows. In Section 2, we provide background information on modern code review, pull requests, language models, and the motivations behind this study. In Section 3, we offer an overview of the proposed architecture and elaborate on its details. In Section 4, we present the results and analysis pertaining to our research questions. Section 5 covers related work in the area of code reviewer recommendation systems. In Section 6, we evaluate the threats to the validity of our research. Finally, Sections 7 and 8 are devoted to concluding the paper and outlining future work.

2 Background

In this section, we introduce background knowledge on the techniques followed by CCB-RR in code reviewer recommendations. In addition, we also introduce the motivation for our work.

2.1 Modern code review

In software development, "pull requests" denote a developer's proposed changes to a project [8]. These proposals are evaluated via the Modern Code Review (MCR), an informal, tool-based approach widely used in today's software development practices [9,10]. The MCR process involves submitting pull requests, having them reviewed and refined, and finally incorporating these changes into the version control system [8]. Platforms like Gerrit, Phabricator, Codestriker, JCR, and ReviewBoard facilitate this process. The goal of MCR is to expedite the development process, enhance security, and ensure software reliability by enabling early detection of potential issues [11].

2.2 Pull requests

When someone wants to make changes to the code, they usually go through a process. First, they create a copy, or a "fork", of the code. They make their changes to this copy, and then start a pull request. This is basically a formal way of suggesting the changes to the project. Next, the appropriate reviewers take a look at these proposed changes. Based on what the reviewers think, the changes either get added into the main project or they get turned down. Pull requests are really important for contributing to public repositories [8], and they're used for more than just submitting patches [12].

Each pull request gets its own page, created by the code review tool, where you can see all the details about it, as shown in Figs. 1 and 2. This includes the current status of the pull request ①, the list of reviewers ②, the title of the pull request ③, when it was last updated ④, a description of the pull request ⑤, and where the file of the pull request can be found ⑥. We chose Gerrit as the review tool mainly because it is widely used in major open-source projects and its features are very similar to other review tools [13].

2.3 Language model

Language models have been pivotal in natural language processing (NLP) tasks over the years, playing significant roles in text data processing tasks [14,15]. In language modeling, we compute the probability of a sequence T of length n, constituted of words $w_1, w_2, ..., w_n$. We express this probability $P(w_1, w_2, ..., w_n)$ using the chain rule of probabilities, which can be written compactly as:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, \ldots, w_{i-1}). \quad (1)$$

Language models are extensively used in numerous applications like machine translation, information retrieval, automatic question answering, and text summarization.

A critical concern in NLP is the out-of-vocabulary (OOV) problem, where words absent from the training set appear in the test set, leading to the language model assigning these words a zero probability. Similarly, a missing subsequence in the training set also results in a zero probability. To mitigate this issue, many techniques employ Laplace Smoothing, expressed as:

Fig. 1 The flowchart illustrating the modern code review process

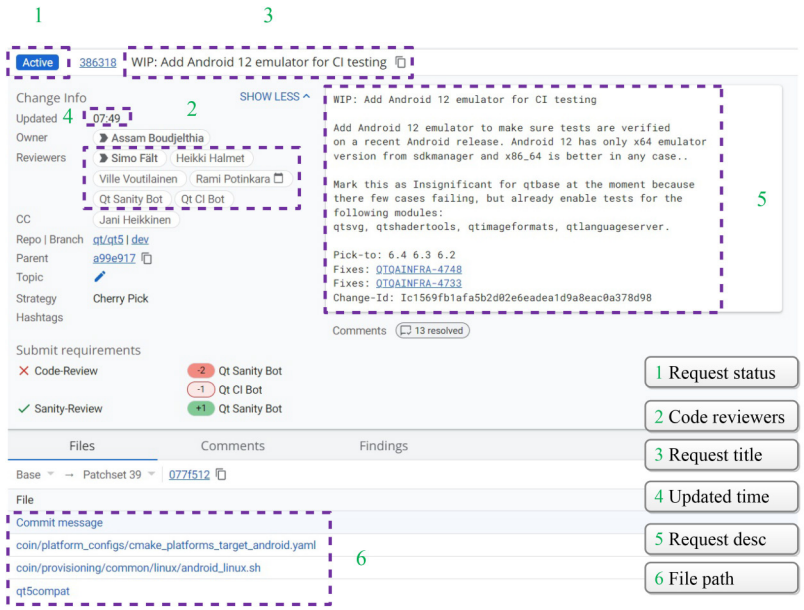

Fig. 2 An example of a pull request

$$P(w_n|w_1,w_2,\ldots,w_{n-1}) = \frac{C(w_1,w_2,\ldots,w_n)+1}{C(w_1,w_2,\ldots,w_{n-1})+V}, \quad (2)$$

where $C(w_1,w_2,\ldots,w_n)$ is the n-gram count in the training set, and V is the vocabulary size.

The n-gram model, based on the Markov chain, employs maximum likelihood estimation to assess a sentence's plausibility. It offers benefits such as ease of parameter training and computational simplicity but suffers from an increased computational complexity due to the inclusion of all the information from the first n–1 words. Furthermore, it struggles with capturing long-term dependencies and generalizing capabilities, resulting in OOV issues when data is sparse. Despite these drawbacks, it excels in interpretability and simplicity.

On the other hand, the neural network language model, which assigns a vector (word vector) to each word in a continuous space, provides superior generalization capabilities and reduces data sparsity issues. The model assesses the constrained relationship between the current word's probability and its preceding n–1 words as:

$$P(w_n|w_1,w_2,\ldots,w_{n-1}). \quad (3)$$

However, it is limited by the extent of preceding information it can contain.

Language models have gradually evolved and diverged into statistical language models and neural network language models. While early statistical language models facilitated swift computer processing, they struggled with measuring semantic similarity between words and learning contextual information from the text. The emergence of neural network language models introduced approaches such as Word2vec [16] and Glove [17], where each word has a fixed vector representation, with semantically similar words possessing similar vectors. These represented vectors can be used for training in neural networks, paving the way for pretrained language models. As deep learning techniques have advanced, more potent feature extractors like ELMo [18], Transformer [7], T5 [19], XLNET [20], RoBERTa [21], ALBERT [22], and GPT- 3 [23] have been introduced, continually enhancing the effectiveness of pre-trained models and improving the performance of natural language processing tasks.

3 Approach

The process of recommending code reviewers is a critical component of the code review workflow. As such, the selection of appropriate reviewers forms the foundation of recommendation strategies. Previous studies [3,4,24] have proposed a plethora of approaches, largely based on data mining and machine learning algorithms, to facilitate code reviewer recommendations. However, these methods, despite their ability to recommend reviewers, often fail to fully leverage the information embedded within the pull request. This oversight can lead to recommended reviewers not performing as anticipated. A pull request contains valuable data such as the modified code path, code context, and text description, all of which can be utilized to establish connections between the reviewer and the pull request. With the use of deep learning, we aim to characterize reviewers from these three perspectives. This approach allows us to simulate reviewers, gaining insights through these three features, thereby enhancing our ability to rectify bugs.

3.1 Motivation

The necessity for tailored code reviews stems from the fact that different reviewers have varying areas of expertise within a range of code procedures. For instance, some reviewers may excel in the implementation of web procedures, while others are proficient in database reading and writing procedures. These specialized code procedures are typically found in distinct file paths within the pull request, allowing us to identify and assign suitable reviewers. Earlier methods such as REVFINDER [4] and TIE [1] leveraged file path similarity and description textual information respectively to recommend reviewers. However, these approaches do not fully capture the

essence of a pull request, which includes code-modified file path, title, and description that collectively represent the functionality of the source code. Our research revealed that many files had been modified due to version iterations. Therefore, the historical data regarding these changes should also be considered in reviewer recommendation. Moreover, it is insufficient to rely solely on textual information to decide reviewer recommendation. We should leverage the historical reviewer recommendation experience to anticipate future reviewer selection. This calls for a reference to historical reviewer recommendation records. A programmer modifies the code file based on the pull request description, which includes deleted and newly modified lines of code. File content, once refined, can effectively address path change and description correspondence. Moreover, studies [25] reveal that code contextual information plays a vital role in the reviewer's decision-making process during a code review session. Hence, our proposed approach initiates with the information in the code file to recommend code reviewers.

3.2 Overview of approach

Our research questions and experiment results are presented in this section, along with a comparison with existing baselines. We propose three research questions:

- RQ1: How effective is our proposed CCB-RR in recommending reviewers?
- RQ2: How does CCB-RR compare with different sub-networks?
- RQ3: How does keyword extraction with KeyBERT perform in our model?

Figure 3 provides an overview of CCB-RR, comprising three principal sub-networks: *File Path Network*, *Aware Network*, and *Textual Network*. These sub-networks extract features from different data types: modified file paths, modified source files with context information, and textual details of the pull requests. To extract textual features, we employ the word2vec technique to represent text information in a low-dimensional vector space. This results in semantically similar words being close in vector space. The word2vec technique uses statistical strategies to predict the current word based on context information. These extracted features serve as inputs to the self-attention network to obtain contextual representations, which are then combined by a fully connected layer for prediction.

Our approach uses an encoder to capture code review information features. The encoder structure comprises six layers, each containing self-attention and fully connected components. Our approach primarily extracts features from the review information, such as file path, source code, and description. We use word embedding technology to extract textual information and the KeyBERT technique to extract keywords from the source code. The features of these keywords are then captured using embedding technology. Subsequently, a self-attention-based encoder captures these features, and a supervised training method is used to train our model. The encoder model connects to a Prediction feed-forward network (FFN) layer at the end, which uses the ReLU activation function and outputs the classification through a linear projection layer.

3.3 File path network feature extraction

The File Path Network focuses on the file location to choose potential reviewers [1,8,26,27,28]. Prior research [1] indicates that file paths provide helpful, albeit somewhat concealed, information for recommending appropriate code reviewers. This information includes similarities between modified file paths and those of previously reviewed files, along with the count of shared directories and file names.

While these metrics are helpful, our approach prioritizes the extraction of global file path information. We employ a self-attention network [7] to extract textual features from file path corpora. After segmenting the file path into tokens by separating special characters and camel-case words, we use the word embedding technique to convert tokens into dense vectors. A self-attention network is then used to capture the features of these file paths. This process is illustrated as shown in Fig. 2. The file path is broken down into individual tokens as follows:

src/corelib/global/qcompilerdetection.h

This is segmented into "**src**", "**corelib**", "**global**", "**q**", "**compiler**", "**detection**", and "**h**". Similar directory structures in file paths often suggest analogous functionalities in the code, which implies that the assigned reviewers could possess comparable skills and review habits.

3.4 Context-aware network

Considering the entire context information for source files might result in an unreasonable use of storage and computational resources due to the extensive information they

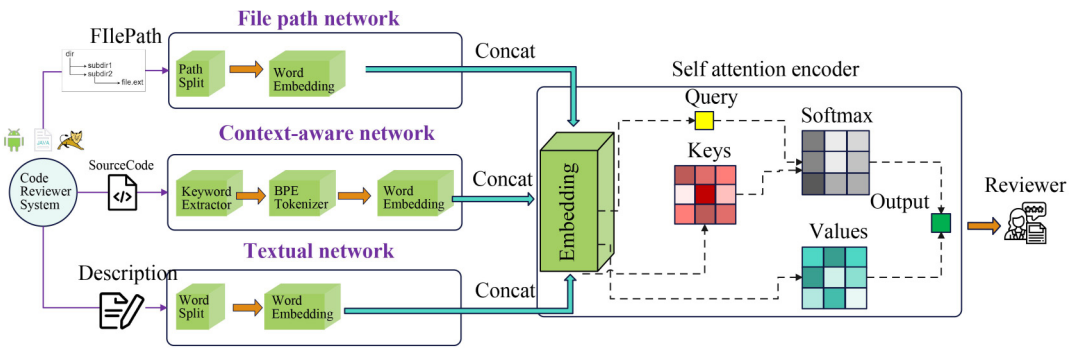

Fig. 3 An overview of CCB-RR

contain. Moreover, the diversity of programming languages in the modified files impedes the extraction of content information using traditional code parsing methods.

Moreover, keyword extraction, an automatic extraction of single or multiple words that represent critical aspects of a text document's content [29,30], is employed in our approach. We use an algorithm called KeyBERT [5] to capture the context and modified information of source files. KeyBERT, based on BERT [31] (Bidirectional Encoder Representation from Transformers), captures bidirectional features from unlabeled text.

Once we obtain the keywords, they are segmented into tokens. This step is crucial as the keywords, which are summaries of the source code, vary from natural language text. Using standard word-splitting methods may result in a large word list that does not handle low-frequency words effectively, leading to out-of-vocabulary (OOV) problems [6]. To address this issue, we use the Byte Pair Encoding (BPE) algorithm to handle low-frequency words by splitting them into subwords. This method has shown promising results in Natural Language Processing tasks [21,32]. We use the self-attention network to capture information after splitting the vocabulary into subwords, aiming to obtain vectors with context information.

3.5 Textual network

Textual information in the pull request, such as details about source code bugs, is closely related to the code review process [1]. In the context of Modern Code Review (MCR), this textual information bridges the gap between code reviewers and developers, offering an understanding of critical issues and changes made. The title and description of a pull request provide a multi-level understanding of its content. For instance, the pull request description often details information about faulty code and its location, thereby helping to recommend suitable code reviewers [1]. Thus, the pull request's description is a valuable guide in the Code Change-Based Reviewer Recommendation (CCB-RR) task. We split the title and description text into tokens, then apply word embedding techniques to create dense vectors that capture their semantic properties. These vectors are fed into a self-attention network to learn pull request features, resulting in separate vectors representing the title and description textual information.

3.6 Self-attention mechanism used to learn contextual features

Upon acquiring the embedding vectors from the file path, contextual, and textual networks, we use a model equipped with a self-attention mechanism to capture their intrinsic features. This approach allows us to find connections between the networks, enhancing our capability to recommend reviewers based on the pull request. Ashish et al. [7] proposed a Transformer model based on a self-attention mechanism that effectively captures semantic information. The model identifies global dependencies through a weighted average operation, without the need to consider word distances.

Given a sequence $S_n = s_1, \ldots, s_I$ of length I, we first adjust its length to a fixed size, performing padding or truncation as necessary. The lengths we set for different types of input tokens, such as file path, context information, modified information, pull request title, and description are 15,200,200,100, and 100, respectively. Next, we apply a word embedding layer to each word in the sequence, converting it into a representation dimension d, resulting in $S'_n = s'_1, \ldots, s'_I$ where each $s'_i \in \mathbf{R}^d$. Finally, we transform S'_n into the query vector $Q_n \in \mathbf{R}^{I\times d}$, the key vector $K_n \in \mathbf{R}^{I\times d}$, and the value vector $V_n \in \mathbf{R}^{I\times d}$ by multiplying it with three distinct weight matrices W_Q, W_K, and W_V respectively. This way, each original word in the sequence is converted into three different representations, serving different roles in the self-attention mechanism.

$$Q_n = S_n \cdot W_Q^{\mathrm{T}}, \tag{4}$$

$$K_n = S_n \cdot W_K^{\mathrm{T}}, \tag{5}$$

$$V_n = S_n \cdot W_V^{\mathrm{T}}. \tag{6}$$

We then utilize the scaled dot-product attention model within the self-attention block to capture semantic information from different sequences. This is computed as follows:

$$\mathrm{Attention}(Q_n, K_n, V_n) = \mathrm{softmax}\left(\frac{Q_n \cdot K_n^{\mathrm{T}}}{\sqrt{d_k}}\right) \cdot V_n. \tag{7}$$

In this computation, the query vector interacts with the key vector at each position, resulting in a score reflecting the correlation between the current position and other positions. This score signifies the "Attention Level" towards other positions. The normalization factor $\sqrt{d}$, the square root of the key vector's dimension, is used to prevent the result from becoming excessively large. Subsequently, we perform a weighted summation of each value vector based on the $softmax$ score. The summed vector is the output of the self-attention module. Lastly, the final output for each feature is obtained by passing $V \in \mathbf{R}^{I\times d}$, the output of the self-attention network, through a simple position-wise fully connected feed-forward network. This is given by:

$$V = \mathrm{ReLU}(\mathrm{Attention}(Q_n, K_n, V_n) \cdot W_1 + b_1) \cdot W_2 + b_2. \tag{8}$$

Here, $W_1 \in \mathbf{R}^{d\times 4d}$, $W_2 \in \mathbf{R}^{d\times 4d}$, $b_1 \in \mathbf{R}^{d\times 4d}$, and $b_2 \in \mathbf{R}^{d\times 4d}$ are the learnable parameters in the model. The activation function is ReLU(·). The self-attention mechanism surpasses LSTM and CNN networks in learning temporal and local features, making it an excellent choice for capturing the intrinsic characteristics of different networks.

3.7 Code reviewer recommendation via fully connected neural network

A Fully Connected Neural Network (FNN) is a type of artificial neural network characterized by connections between all neurons in adjacent layers. Such architecture permits learning complex patterns and relationships from data, making it a popular choice in applications such as image classification and speech recognition. FNN consists of multiple layers: the input layer, which intakes data, hidden layers that process the input data, and the output layer, responsible for final predictions or classifications. The hidden layers are so-called because they are not directly accessible, unlike the input and

output layers.

We construct a joint vector that encapsulates the knowledge from the current pull request by merging vectors associated with file paths, context information, and modified information of the altered files, along with the textual information of the pull request. This joint vector is subsequently processed by the FNN to make reviewer recommendations. The FNN learns the joint vector, decreases its dimensionality, and outputs a lower-dimensional vector. The lower-dimensional vector signifies the potential reviewers to whom the pull request should be allocated.

Training our multi-classifier to determine reviewer assignments involves applying supervised learning techniques. The output vector embodies the probabilities associated with each potential reviewer. Conventionally, Code Change-Based Reviewer Recommendation (CCB-RR) selects the top K reviewers based on these probabilities.

The employment of FNNs and a softmax classifier is justified by several reasons. Primarily, FNNs excel in modeling complex non-linear relationships, making them suitable for large-scale, data-intensive codebases. The softmax classifier, typically used as the output layer in a neural network, can represent the probability distribution over multiple classes, aiding in multi-class classification problems. Moreover, both FNNs and softmax classifiers are chosen for their interpretability and adaptability in preventing overfitting by regularization.

4 Experiments

In this section, we describe the details of the dataset, experimental design, evaluation metrics, and analysis of experimental results.

4.1 Dataset description

To evaluate our model, we employed a publicly accessible dataset, which users can download from the Android Review public dataset repository. This dataset comprises four projects often used in code reviewer recommendation tasks [1,3,4,33,34]. Each pull request in this dataset is classified as either "Merged" or "Abandoned", and it contains at least one modified file path. However, some important details like request descriptions and modified source code file information were missing from the original dataset. To rectify this, we re-collected the pull requests, eliminating unnecessary details such as segmented sentences, symbols, and numerals. The data were then chronologically arranged to aid model validation and comparison. The historical dataset was split into three parts: 80% for training, 10% for validation, and 10% for testing.

Table 1 presents the statistics of these projects. The rows specify project names, the number of reviews, the total count of reviewers, the count of modified code files, the average number of candidate reviewers per review, and the average count of modified files per review.

Table 1 Statistics of project reviews, reviewers, and modified files

Statistical type	Android	OpenStack	Qt	LibreOffice
Reviews	5126	6586	23810	6523
Code-reviewers	94	82	202	63
Files	26840	16953	78401	35273
Average Reviewers	1.06	1.44	1.07	1.01
Average Files	8.26	6.04	10.64	11.14

4.2 Evaluation metrics

We employ two widely adopted metrics: $Top@k$ and Mean Reciprocal Rank (MRR), which are commonly used to evaluate the performance of ranking models. Our method assesses the ranking quality of the proposed model utilizing both training and testing datasets.

4.2.1 Top-k accuracy

This measure signifies the fraction of pull requests in which the actual code reviewers are situated within the Top-k positions in the returned reviewer ranking list. Given a pull request r and Top-k, the function $isRecomm(r, Top-k)$ provides a binary output indicating the correctness of the reviewer recommendation. We denote this correctness by assigning a value of 1 or 0 to $isRecomm(r, Top-k)$. If at least one of the top-k recommended reviewers actually reviewed the pull request r, the value is set to 1. This indicates a correct recommendation. If not, it is set to 0, denoting an incorrect recommendation. Therefore, for a given set of pull requests PR, Top-k accuracy is calculated as:

$$Top@k = \frac{\sum_{r \in PR} isRecomm(r, Top\text{-}k)}{|PR|}. \tag{9}$$

The equation defines $Top@k$, a metric for recommendation systems. It measures the ratio of actual recommendations found in the Top-k suggested items. *isRecomm* checks if a recommendation is in the Top-k, and $|PR|$ denotes the total number of real recommendations.

4.2.2 Files at risk (FaR)

FaR is a key measure in code reviewer systems. It estimates the risk of losing knowledge in a project. If only a few developers know a file well, the file is *at risk*. If these developers leave, crucial knowledge may be lost. We calculate FaR like this: Let's say you have a project P with a file f. D_f is all the contributors to f. The risk for f, or R_f, is:

$$R_f = \frac{1}{|D_f|}. \tag{10}$$

$|D_f|$ is the number of developers in D_f.

To get the total risk, R_P, for the project P, you take the average risk for all files f:

$$R_P = \frac{1}{|F|} \sum_{f \in F} R_f. \tag{11}$$

F is all files in the project.

Using FaR in code reviewer recommendations helps spread out knowledge of the codebase. This can lower the risk of losing knowledge.

4.2.3 Mean reciprocal rank (MRR)

For a specific pull request r, its reciprocal rank is the position of the first correct reviewer in the ranking list generated by the code reviewer recommendation model. The function $rank(r)$ denotes the position of the first correctly recommended

reviewer in the returned ranking list, and then the MRR is computed as the average of the reciprocal ranks for the pull requests in a set of pull requests PR:

$$MRR = \frac{1}{|PR|}\sum_{r\in PR}\frac{1}{rank(r)}. \quad (12)$$

The larger the values of $Top@k$ and MRR, the better the performance of the code reviewer recommendation model. In this study, we set k to 1, 3, 5, and 10.

4.3 Research questions

4.3.1 Experiment for RQ 1

Motivation The goal of RQ1 is to evaluate the effectiveness of the CCB-RR model against other baseline methods including RS, REVFINDER, and WRC. A positive outcome would support the use of deep learning in enhancing the code reviewer recommendation by leveraging the contextual information of source files.

Approach The performance of CCB-RR is compared with established code reviewers' recommendation methods. RS [3] employs a recommender-based approach and analyzes reviewer behavior.

REVFINDER [4] incorporates file location in its recommendations, and WRC is a path-similarity-based approach [24]. The effectiveness is determined by Top-k accuracies (k = 1, 3, 5, and 10) and MRR.

Results Table 2 shows the Top-k prediction accuracies (Top 1, 3, 5, 10) and Mean Reciprocal Rank (MRR) results of seven different methods across four projects: Android, OpenStack, QT, and LibreOffice. The "CCB" method generally outperforms the other methods in most categories across all projects, signifying its robustness. However, its effectiveness varies, and in some cases, like with the LibreOffice project, other methods like "TIE" can exhibit higher performance in certain categories. These findings highlight that the best method can depend on the specific project and prediction category. The lower performance of the Code Context Based Reviewer Recommendation (CCB) method on the LibreOffice project, as shown in the table, is due to insufficient context information and text content in this project. To enhance CCB's performance, it is necessary to enrich the project with more context and descriptive text data that align with the KeyBERT model's requirements.

The values in Table 3 presents a comparison of seven models (RS, REV, WRC, TIE, TNE, RevRec, and CCB) across four projects using the FaR metric. In most cases, the Code Context Based Reviewer Recommendation (CCB) model performs the best in terms of the FaR score, indicating its superior effectiveness in minimizing faults. The average performance across all projects also shows CCB as the most efficient model.

4.3.2 Experiment for RQ 2

Motivation The core model of our research, CCB-RR, is an amalgamation of three distinct components: the File Path Network, the Context-Aware Network, and the Textual Network. These networks process varied data facets including the file path, contextual code modifications, and the title and description of pull requests. The purpose of the second research question (RQ2) is to examine the individual and collective efficacy of these networks, and ascertain whether the combination of these networks provides superior performance to their individual counterparts.

Approach To explore RQ2, we employ ablation experiments to delve into the individual influence of each sub-network on the overall performance of our model. The comparison is drawn against the combined result of the CCB-RR model and the individual outputs of its three sub-networks. The performance of these components is then assessed through Top-k prediction accuracies (where k = 1, 3, 5, and 10) and Mean Reciprocal Rank (MRR).

Table 2 Top-k Prediction accuracies (Top 1, 3, 5, 10) and MRR results achieved by each method for each system. These methods contain Recommendation System (RS), REVFINDER (REV), Weighted Reviewer Classification (WRC), Text Information Extraction [1] (TIE), Topic-based Navigation Engine [27] (TNE), Reviewer Recommendation [33] (RevRec), and Code Context Based Reviewer Recommendation (CCB) represent different models

Project	Method	Top 1	Top 3	Top 5	Top 10	MRR
Android	RS	0.37	0.55	0.59	0.66	0.46
	REV	0.40	0.64	0.68	0.81	0.55
	WRC	0.43	0.70	0.68	0.77	0.55
	TIE	0.52	0.76	0.82	0.87	0.65
	TNE	0.40	0.65	0.70	0.75	0.60
	RevRec	0.53	0.42	0.34	0.29	0.64
	CCB	**0.60**	**0.80**	**0.85**	**0.83**	**0.71**
OpenStack	RS	0.28	0.61	0.73	0.77	0.40
	REV	0.32	0.61	0.73	0.82	0.50
	WRC	0.36	0.63	0.70	0.86	0.44
	TIE	0.38	0.68	0.78	0.86	0.55
	TNE	0.30	0.60	0.70	0.75	0.50
	RevRec	0.54	0.46	0.38	0.31	0.58
	CCB	**0.55**	**0.72**	**0.81**	**0.87**	**0.62**
QT	RS	0.33	0.57	0.67	0.63	0.45
	REV	0.26	0.33	0.39	0.64	0.26
	WRC	0.33	0.57	0.64	0.62	0.36
	TIE	0.25	0.40	0.47	0.57	0.36
	TNE	0.27	0.55	0.65	0.70	0.50
	RevRec	0.44	0.40	0.36	0.29	0.49
	CCB	**0.51**	**0.61**	**0.73**	**0.77**	**0.52**
LibreOffice	RS	0.15	0.28	0.37	0.66	0.45
	REV	0.19	0.36	0.46	0.69	0.35
	WRC	0.27	0.43	0.52	0.72	0.50
	TIE	**0.71**	**0.86**	**0.88**	**0.91**	**0.79**
	TNE	0.25	0.45	0.55	0.65	0.47
	RevRec	–	–	–	–	–
	CCB	0.45	0.59	0.72	0.80	0.68

Table 3 FaR metric comparison for various models. Recommendation System (RS), REVFINDER (REV), Weighted Reviewer Classification (WRC), Text Information Extraction [1] (TIE), Topic-based Navigation Engine [27] (TNE), Reviewer Recommendation [33] (RevRec), and Code Context Based Reviewer Recommendation (CCB) represent different models

Project	RS	REV	WRC	TIE	TNE	RevRec	CCB
Android	0.35	0.37	0.36	0.38	0.33	0.31	**0.42**
OpenStack	0.32	0.33	0.35	0.40	0.30	0.29	**0.43**
QT	0.34	0.35	0.40	0.39	0.32	0.30	**0.44**
LibreOffice	0.39	0.37	0.36	0.35	0.34	0.32	**0.41**
Average.	0.35	0.36	0.37	0.38	0.33	0.31	**0.42**

Results Each sub-network holds a pivotal role in recommending appropriate code reviewers. When combined, these sub-networks demonstrate greater efficiency than when operating alone. As evident from Table 4, the combined performance of CCB-RR outshines the individual capabilities of the File Path Network, Context-Aware Network, and Textual Network, both in terms of $Top@k$ prediction accuracies and MRR values.

Upon comparing the results, we observe that the CCB-RR model outperforms the individual networks in the majority of cases. Across the four tested projects, the most noteworthy enhancement over the baseline is observed in the Top-1 prediction by CCB-RR, regardless of the sub-network in question. CCB-RR's superior performance can be attributed to its holistic approach to data, with different sub-networks complementing each other and fostering a more comprehensive understanding of pull requests.

Interestingly, our findings suggest that for Android and OpenStack, the Context-Aware Network trumps the composite CCB-RR model in terms of Top-3 accuracy. However, for Top-1 accuracy, the entire network takes precedence over the individual sub-networks. While the Context-Aware Network tends to rank suitable reviewers within the top three, the combined model excels in identifying the most appropriate reviewer as the top candidate. Since developers often pre-choose candidate reviewers from recommendation lists, a model's ability to prioritize the most suitable reviewer holds significant real-world value.

Upon comparing the individual performances of each sub-network, it is evident that the combined approach leads to superior results. The File Path and Context-Aware Networks outperform the Textual Network, primarily due to their data sources. The file paths, reflecting the file location of the pull request, and the context-aware source files, containing the change history, hold significant relations to potential code reviewers. Conversely, the developer-generated textual information might be subjective and less linked to potential reviewers. On average, textual analysis of source files yields insufficient information about pull requests, which explains why the Textual Network falls short of the

4.3.3 Experiment for RQ 3

Motivation Our methodology heavily relies on KeyBERT for extracting keywords that encapsulate the most pertinent information from a pull request. This research question (RQ) is targeted at understanding KeyBERT's effectiveness in obtaining knowledge from source files' contextual and modified data, as well as assessing KeyBERT's performance as a primary information extractor. Specifically, we investigate if the use of KeyBERT can enhance the efficiency of information extraction in the Context-Aware Network for code reviewer recommendation.

Approach To ascertain the efficacy of KeyBERT in keyword extraction within the entire network, we initiate a comparative analysis of KeyBERT with two other keyword extractors used in the Context-Aware Network: YAKE [35] and RAKE [5]. YAKE is an unsupervised learning methodology for automatic keyword extraction from documents by examining statistical features of the text from five perspectives, such as case, word positional, word frequency, word relatedness to context, and word different sentence. Conversely, RAKE extracts keywords based on the co-occurrence relationships of words within a document. Following these evaluations, we measure the Top-k accuracies (k = 1, 3, 5, and 10) and Mean Reciprocal Rank (MRR) across the keyword extraction methodologies.

Results The results indicate that the KeyBERT algorithm effectively summarizes relevant information in code files and efficiently recommends suitable code reviewers in software engineering tasks. Table 5 presents the Top-k accuracies (k = 1, 3, 5, and 10) and MRR values of the Context-Aware Network using KeyBERT, YAKE, and RAKE for keyword extraction. Upon analyzing the experimental results, we observed that the Context-Aware Network, when integrated

Table 4 Performance comparison of different submodules and full model

Project	Top-1				Top-3				Top-5				Top-10				MRR			
	I	**II**	**III**	**F**	**I**	**II**	**III**	**F**	**I**	**II**	**III**	**F**	**I**	**II**	**III**	**F**	**I**	**II**	**III**	**F**
Android	0.44	0.45	0.27	**0.60**	0.72	0.76	0.67	**0.80**	0.79	0.80	0.75	**0.85**	0.84	0.86	0.86	0.83	0.59	0.61	0.47	**0.71**
OpenStack	0.38	0.34	0.33	**0.55**	0.70	0.76	0.60	**0.72**	0.81	0.80	0.75	**0.81**	0.91	0.88	0.88	0.87	0.67	0.61	0.57	**0.62**
Qt	0.34	0.28	0.30	**0.51**	0.71	0.53	0.56	**0.61**	0.78	0.64	0.64	**0.73**	0.86	0.74	0.74	0.77	0.51	0.45	0.47	**0.52**
LibreOffice	0.43	0.40	0.36	**0.45**	0.60	0.59	0.54	**0.59**	0.67	0.68	0.67	**0.72**	0.77	0.85	0.79	0.80	0.54	0.53	0.49	**0.68**
Average	0.40	0.37	0.32	**0.53**	0.68	0.66	0.59	**0.68**	0.76	0.73	0.70	**0.78**	0.85	0.83	0.82	0.82	0.58	0.55	0.50	**0.63**

Key: I - File Path Network, II - Context-Aware Network, III - Textual Network, F - Full Model (CCB-RR)

Table 5 Comparison of performance in different feature extraction technologies

Project	Top-1			Top-3			Top-5			Top-10			MRR		
	Y^+	R^+	*KeyBERT*$^+$	Y^+	R^+	*KeyBERT*$^+$	Y^+	R^+	*KeyBERT*$^+$	Y^+	R^+	*KeyBERT*$^+$	Y^+	R^+	*KeyBERT*$^+$
Android	0.32	0.27	**0.60**	0.61	0.68	**0.80**	0.70	0.75	**0.85**	0.79	0.87	0.83	0.48	0.47	**0.71**
OpenStack	0.28	0.29	**0.55**	0.54	0.52	**0.72**	0.68	0.65	**0.81**	0.84	0.81	**0.87**	0.53	0.53	**0.62**
Qt	0.19	0.15	**0.51**	0.37	0.30	**0.61**	0.49	0.41	**0.73**	0.64	0.57	**0.77**	0.34	0.28	**0.52**
LibreOffice	0.28	0.29	**0.45**	0.56	0.58	**0.59**	0.64	0.66	**0.72**	0.76	0.76	**0.80**	0.46	0.45	**0.68**
Average	0.27	0.25	**0.53**	0.52	0.52	**0.68**	0.63	0.62	**0.78**	0.76	0.75	**0.88**	0.45	0.43	**0.63**

KeyBERT$^+$: Using KeyBERT as the feature extractor for extracting context information in Context-Aware Network; Y^+: Using YAKE as the feature extractor for extracting context information in Context-Aware Network; R^+: Using RAKE as the feature extractor for extracting context information in Context-Aware Network.

with KeyBERT, generally outperforms the versions with YAKE and RAKE in terms of accuracy and MRR.

While YAKE tends to prioritize capitalized words and words positioned at the document's beginning, it does not entirely conform to our keyword selection criteria. Furthermore, RAKE's effectiveness is contingent on a comprehensive list of stop words; otherwise, it could result in disproportionately long phrases. In contrast, KeyBERT leverages BERT embeddings to generate semantically relevant keywords from documents, focusing more on the text's semantic content rather than just word frequency. By using KeyBERT, we address the issue of keyword selection bias towards word frequency in traditional keyword algorithms. KeyBERT facilitates the identification of keywords within the source file that can accurately represent the overall source file context and modified information via semantic similarity comparisons.

5 Related work

This section offers an overview of existing research on code reviewer recommendation. We categorize this discussion into machine learning-based, heuristic-based, and multi-objective methods, as well as keyword extraction techniques, for a more organized understanding. This field has gained significant attention in software engineering research due to its inherent challenges and solutions.

5.1 Machine learning-based approaches

Machine learning techniques have been extensively applied for recommending code reviewers. Studies such as Nasrabadi and King [36], de Lima Junior et al. [37], and Mirsaeedi and Rigby [38] have used various data inputs and algorithms for this purpose. Recently, Ye et al. [26] employed deep learning in this context. Our proposed approach, CCB-RR, incorporates the context of the target modified file and the code change to enhance the precision of recommendations.

5.2 Heuristic-based approaches

Heuristic strategies have been widely used for code reviewer recommendation. These strategies often utilize mathematical expressions and score calculations [39]. For example, Balachandran et al. [40] and Asthana et al. [41] have integrated automatic static analysis and developer experience, respectively. The path-similarity-based approach, WRC [24], also falls under this category. Our approach, CCB-RR, enhances these strategies by considering the code information from source files.

5.3 Multi-objective approaches

Multi-objective methods such as FaR [38] have been used for code reviewer recommendations, considering factors like reviewer workload and expertise. Our approach adds a unique perspective by examining the relationships between pull requests and reviewers.

5.4 Domain-specific keyword extraction technologies

Keyword extraction is a critical research domain in machine learning and artificial intelligence. Various methods like TF-IDF by Salton et al. [42], YAKE [35], and graph-based ranking methods like TextRank [43] and Lexrank [44] have been proposed. Deep learning-based approaches like the sentence embedding method by Bennani-Smires et al. [45] and the KeyBERT model [5] mark a shift in the domain. The deep learning-based TIE, Text, and Sim methods are examples of this shift. These methods effectively address issues related to word frequency and semantic relevance, prioritizing the overall relevance of the text.

6 Threats to validity

This section acknowledges the inherent limitations due to our study's methodological choices. We classify these limitations into four categories: construct validity, internal validity, external validity, and reliability. We have adapted this classification from best practices, as recommended in the literature [46].

6.1 Construct validity

Construct validity is concerned with the appropriateness of the metrics used to represent the phenomenon under study. Following the suggestions proposed by H. A. Çetin et al. [47], we used well-recognized metrics such as TopK, MRR, and FaR. These metrics have been rigorously evaluated in our experiments, lending credibility to our findings. Plans are in place to introduce more comprehensive metrics in future studies to enhance the construct validity of our approach.

6.2 External validity

As pointed out by Kong et al. [48], generalizability is a common issue in reviewer recommendation systems. To address this, we conducted tests on four different projects —Android, OpenStack, QT, and LibreOffice —to provide a wide spectrum of use-cases. However, the specific datasets we used limit the broad applicability of our findings. Therefore, future work should involve testing on more diverse project types, such as AI-driven systems, web applications, and healthcare software, to further affirm the external validity.

6.3 Reliability

Reliability measures the consistency of research findings over time. Echoing the concerns raised by Li et al. [49], potential threats to reliability could arise from various aspects of the study's experimental settings, such as the sampling methods and the choice of metrics. To address such problem, we conducted time-series evaluations that extended our analysis over time. This approach allowed us to gauge the model's performance during various stages of project development. By comparing our method with established baseline approaches, we further bolstered the reliability of our study.

7 Discussion

7.1 Deep learning and model selection

Our research demonstrates the considerable benefits of deep learning for handling complex tasks and identifying subtle details, particularly when dealing with large datasets. This becomes evident in our code review experiment. However, the choice of model, in this case KeyBERT for keyword extraction, draws attention to the importance of understanding the nature of data. We see the potential of using CodeBERT, a

model specifically built for source code analysis, in future research. While we need to be cautious not to over-interpret results from deep learning, future studies should focus on developing methods to make these models more interpretable.

7.2 Enhancement strategies

Our current Automatic Reviewer Recommendation system may not fully capture the tree-like structure of file paths, potentially overlooking important data. To address this, we plan to implement methods better suited for handling hierarchical data, such as tree-based models or path embedding algorithms. This may ensure that all critical information is taken into account, enhancing our system's performance and utility for real-world applications.

7.3 Probabilistic reviewer ranking

Our model ranks reviewers based on the predicted probability of their suitability for a PR, not in any preset order like alphabetical. The highest likely reviewer is recommended first, then the next, and so on. We clarify this mechanism in our paper to highlight its importance in shaping our reviewer recommendations.

7.4 Limitations and future directions

Despite the successes, it is important to acknowledge potential risks and downsides of deep learning for code reviewer recommendations. One such limitation is related to the adaptability of our current model to changes in the reviewer pool, a common scenario in open-source projects due to high turnover rates. Addressing this may require modifications to our model's architecture, particularly the Fully Connected (FC) layer, to avoid frequent and impractical retraining. Our future work aims to provide a solution to these challenges, to make the model more practical, efficient, and suitable for real-world applications.

8 Conclusions

A wealth of knowledge from pull requests has been harnessed for code reviewer recommendation tasks. In this study, we encapsulated the valuable information embedded within the modified files and utilized keyword extraction to glean insights from the code files to aid code reviewer recommendations. We introduced a novel code reviewer recommendation approach, CCB-RR, which uses file path information of modified files, context information, changes in source files, and the title and description of pull requests as the basis for input. CCB-RR employs deep learning networks and self-attention neural networks to extract features critical for code reviewer recommendation. Empirical evaluations show that our approach effectively recommends code reviewers across four large-scale open-source projects.

Furthermore, we plan to experiment with additional contextual characterization methods to capture the nuanced information within source files. Additionally, we intend to examine more open-source projects or features to uncover more beneficial information, aiming to further improve our code reviewer recommendation system.

Acknowledgement This work was supported in part by the Science and Technology Development Fund (FDCT), Macau SAR, China (Nos. 0047/2020/A1 and 0014/2022/A).

Competing interests The authors declare that they have no competing interests or financial conflicts to disclose.

References

1. Xia X, Lo D, Wang X, Yang X. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution. 2015, 261–270
2. Shull F, Seaman C. Inspecting the history of inspections: an example of evidence-based technology diffusion. IEEE Software, 2008, 25(1): 88–90
3. Chueshev A, Lawall J, Bendraou R, Ziadi T. Expanding the number of reviewers in open-source projects by recommending appropriate developers. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution. 2020, 499–510
4. Thongtanunam P, Tantithamthavorn C, Kula R G, Yoshida N, Iida H, Matsumoto K I. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering. 2015, 141–150
5. Goto I, Tanaka H. Detecting untranslated content for neural machine translation. In: Proceedings of the 1st Workshop on Neural Machine Translation. 2017, 47–55
6. Stahlberg F. Neural machine translation: a review. Journal of Artificial Intelligence Research, 2020, 69: 343–418
7. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017, 6000–6010
8. Yu Y, Wang H, Yin G, Wang T. Reviewer recommendation for pull-requests in GitHub: what can we learn from code review and bug assignment? Information and Software Technology, 2016, 74: 204−218
9. Zanjani M B, Kagdi H, Bird C. Automatically recommending peer reviewers in modern code review. Transactions on Software Engineering, 2016, 42(6): 530–543
10. Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 35th International Conference on Software Engineering. 2013, 712–721
11. Rigby P C, Storey M A. Understanding broadcast based peer review on open source software projects. In: Proceedings of the 33rd International Conference on Software Engineering. 2011, 541–550
12. Yu Y, Wang H, Filkov V, Devanbu P, Vasilescu B. Wait for it: determinants of pull request evaluation latency on GitHub. In: Proceedings of the 12th Working Conference on Mining Software Repositories. 2015, 367–371
13. Tymchuk Y, Mocci A, Lanza M. Code review: veni, ViDI, vici. In: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering. 2015, 151–160
14. Qiu X, Sun T, Xu Y, Shao Y, Dai N, Huang X. Pre-trained models for natural language processing: a survey. Science China Technological Sciences, 2020, 63(10): 1872–1897
15. Lavrenko V, Croft W B. Relevance-based language models. ACM SIGIR Forum, 2017, 51(2): 260–267
16. Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems. 2013, 3111–3119
17. Pennington J, Socher R, Manning C. GloVe: global vectors for word

representation. In: Proceedings of the 19th Conference on Empirical Methods in Natural Language Processing. 2014, 1532–1543

18. Cherney L R, Kaye R C, Lee J B, van Vuuren S. Impact of personal relevance on acquisition and generalization of script training for aphasia: a preliminary analysis. American Journal of Speech-Language Pathology, 2015, 24(4): S913–S922
19. Lourie N, Le Bras R, Bhagavatula C, Choi Y. UNICORN on RAINBOW: a universal commonsense reasoning model on a new multitask benchmark. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence. 2021, 13480–13488
20. Yang Z, Dai Z, Yang Y, Carbonell J, Salakhutdinov R, Le Q V. XLNet: generalized autoregressive pretraining for language understanding. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. 2019, 517
21. Liu Z, Lin W, Shi Y, Zhao J. A robustly optimized BERT pre-training approach with post-training. In: Proceedings of the 20th China National Conference on Chinese Computational Linguistics. 2021, 471–484
22. Chi P H, Chung P H, Wu T H, Hsieh C C, Chen Y H, Li S W, Lee H Y. Audio albert: a lite bert for self-supervised learning of audio representation. In: Proceedings of 2021 IEEE Spoken Language Technology Workshop. 2021, 344−350
23. Brown T B, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler D M, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D. Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. 2020, 159
24. Hannebauer C, Patalas M, Stünkel S, Gruhn V. Automatically recommending code reviewers based on their expertise: an empirical comparison. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016, 99−110
25. Doğan E, Tüzün E, Tecimer K A, Güvenir H A. Investigating the validity of ground truth in code reviewer recommendation studies. In: Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement. 2019, 1−6
26. Ye X, Zheng Y, Aljedaani W, Mkaouer M W. Recommending pull request reviewers based on code changes. Soft Computing, 2021, 25(7): 5619–5632
27. Fejzer M, Przymus P, Stencel K. Profile based recommendation of code reviewers. Journal of Intelligent Information Systems, 2018, 50(3): 597–619
28. Ye X. Learning to rank reviewers for pull requests. IEEE Access, 2019, 7: 85382–85391
29. Firoozeh N, Nazarenko A, Alizon F, Daille B. Keyword extraction: issues and methods. Natural Language Engineering, 2020, 26(3): 259–291
30. Piskorski J, Stefanovitch N, Jacquet G, Podavini A. Exploring linguistically-lightweight keyword extraction techniques for indexing news articles in a multilingual set-up. In: Proceedings of the 16th EACL Hackashop on News Media Content Analysis and Automated Report Generation. 2021, 35−44
31. Weninger F, Geiger J, Wöllmer M, Schuller B, Rigoll G. Feature enhancement by deep LSTM networks for ASR in reverberant multisource environments. Computer Speech & Language, 2014, 28(4): 888–902
32. Jiang N, Lutellier T, Tan L. CURE: code-aware neural machine translation for automatic program repair. In: Proceedings of the 43rd International Conference on Software Engineering. 2021, 1161−1173
33. Ouni A, Kula R G, Inoue K. Search-based peer reviewers recommendation in modern code review. In: Proceedings of the 32nd International Conference on Software Maintenance and Evolution. 2016, 367−377
34. Chouchen M, Ouni A, Mkaouer M W, Kula R G, Inoue K. WhoReview: a multi-objective search-based approach for code reviewers recommendation in modern code review. Applied Soft Computing, 2021, 100: 106908
35. Campos R, Mangaravite V, Pasquali A, Jorge A, Nunes C, Jatowt A. YAKE! Keyword extraction from single documents using multiple local features. Information Sciences, 2020, 509: 257–289
36. Nasrabadi N M, King R A. Image coding using vector quantization: a review. IEEE Transactions on Communications, 1988, 36(8): 957–971
37. de Lima Junior M L, Soares D M, Plastino A, Murta L. Automatic assignment of integrators to pull requests: the importance of selecting appropriate attributes. Journal of Systems and Software, 2018, 144: 181–196
38. Mirsaeedi E, Rigby P C. Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution. In: Proceedings of the 42nd International Conference on Software Engineering. 2020, 1183−1195
39. Al-Zubaidi W H A, Thongtanunam P, Dam H K, Tantithamthavorn C, Ghose A. Workload-aware reviewer recommendation using a multi-objective search-based approach. In: Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering. 2020, 21−30
40. Balachandran V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 35th International Conference on Software Engineering. 2013, 931−940
41. Asthana S, Kumar R, Bhagwan R, Bird C, Bansal C, Maddila C, Mehta S, Ashok B. WhoDo: automating reviewer suggestions at scale. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019, 937−945
42. Salton G, Buckley C. Term-weighting approaches in automatic text retrieval. Information Processing & Management, 1988, 24(5): 513–523
43. Mihalcea R, Tarau P. TextRank: bringing order into text. In: Proceedings of the 2nd Conference on Empirical Methods in Natural Language Processing. 2004, 404−411
44. Erkan G, Radev D R. LexRank: graph-based lexical centrality as salience in text summarization. Journal of Artificial Intelligence Research, 2004, 22: 457–479
45. Nguyen T D, Luong M T. WINGNUS: keyphrase extraction utilizing document logical structure. In: Proceedings of the 5th International Workshop on Semantic Evaluation. 2010, 166−169
46. Rebai S, Amich A, Molaei S, Kessentini M, Kazman R. Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations. Automated Software Engineering, 2020, 27(3-4): 301–328
47. Çetin H A, Doğan E, Tüzün E. A review of code reviewer recommendation studies: challenges and future directions. Science of Computer Programming, 2021, 208: 102652
48. Kong D, Chen Q, Bao L, Sun C, Xia X, Li S. Recommending code reviewers for proprietary software projects: a large scale study. In: Proceedings of 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2022, 630−640
49. Li R, Liang P, Avgeriou P. Code reviewer recommendation for architecture violations: an exploratory study. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. 2023, 42−51

Dawei Yuan is a PhD student in the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), China. Before joining MUST, he worked as a software engineer at United Imaging Healthcare Corporation (UIH), China. He also got an MSc in Software Engineering from the University of Science and Technology of China and an BSc degree in Network Engineering from Nanjing University of Posts and Telecommunications, China. His research interests mainly focus on using artificial intelligence techniques to solve problems in software engineering, such as bug localization and code analysis.

Xiao Peng received the BSc in information and computing science from the University of Jinan, China in 2020, and MSc in applied mathematics and data science from Macau University of Science and Technology, China in 2022. She is currently working toward the PhD degree in science at Macau University of Science and Technology, China. Her research interests include software engineering and deep learning.

Zijie Chen received the MSc degrees in Applied Mathematics and Data Science from Macau University of Science and Technology, China, and the BSc degree in Software Engineering from Beijing Institute of Technology, China. He is currently pursuing the PhD degree in the School of Computer Science and Engineering, Macau University of Science and Technology, China. His research interests include deep learning, human activity recognition, and Internet of Things.

Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the PhD degree in computer science from the University of Seoul, Republic of Korea. After that, he spent one year with The Hong Kong Polytechnic University, China as a postdoctoral research fellow. Currently, he is an associate professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), China. Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 90 high-quality papers at renowned software engineering and security journals and concerences. He served as the General Chair of SANER 2023 and the PC members of several top-tier SE concerences such as FSE and ASE. He is a senior member of ACM and IEEE.

Ruijia Lei received the BSc degree in Software Engineering from Macau University of Science and Technology, China in 2021, the MSc degree in Computer Science from University of Amsterdam, the Netherlands in 2023, respectively. His research interests include software technology, machine learning, large-scale data processing, distributed systems, and principles and applications of LLMs.