electronics

MDPI

Article

A Code Reviewer Recommendation Approach Based on Attentive Neighbor Embedding Propagation

Jiahui Liu [1,*], Ansheng Deng [1,*], Qiuju Xie [2] and Guanli Yue [1]

[1] School of Information Science and Technology, Dalian Maritime University, Linghai Road, Dalian 116026, China
[2] College of Electrical and Information, Northeastern Agricultural University, Changjiang Road, Harbin 150030, China
* Correspondence: liu.jiahui123@foxmail.com (J.L.); ashdeng@dlmu.edu.cn (A.D.)

Abstract: Code review as an effective software quality assurance practice has been widely applied in many open-source software communities. However, finding a suitable reviewer for certain codes can be very challenging in open-source communities due to the difficulty of learning the characteristics of reviewers and the code-reviewer interaction sparsity in open-source software communities. To tackle this problem, most previous approaches focus on learning developers' capabilities and experiences and recommending suitable developers based on their historical interactions. However, such approaches usually suffer from data-sparsity and noise problems, which may reduce the recommendation accuracy. In this paper, we propose an attentive neighbor embedding propagation enhanced code reviewer recommendation framework (termed ANEP). In ANEP, we first construct the reviewer–code interaction graph and learn the semantic representations of the reviewer and code based on the transformer model. Then, we explicitly explore the attentive high-order embedding propagation of reviewers and code and refine the representations along their neighbors. Finally, to evaluate the effectiveness of ANEP, we conduct extensive experiments on four real-world datasets. The experimental results show that ANEP outperforms other state-of-the-art approaches significantly.

Keywords: attention mechanism; code reviewer; high-order embedding propagation; neighbor embedding; transformer

check for updates

Citation: Liu, J.; Deng, A.; Xie, Q.; Yue, G. A Code Reviewer Recommendation Approach Based on Attentive Neighbor Embedding Propagation. *Electronics* **2023**, *12*, 2113. https://doi.org/10.3390/electronics12092113

Academic Editor: Gwanggil Jeon

Received: 1 March 2023
Revised: 24 April 2023
Accepted: 1 May 2023
Published: 5 May 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In recent years, code review has been widely applied in the open-source software (OSS) development community. Code review can not only ensure the code quality, but also improve developers' learning and collaboration efficiency. Previous studies [1–5] also show that reviewed code has a lower bug rate and exhibits higher quality because reviewers can check the code and point out the shortcomings, mistakes, and convention violations during code review that may be overlooked during code development [6].

Although code reviews are necessary, choosing the appropriate reviewers for submitted code is a challenging problem. For example, in most OSS community projects, the project manager or integrator usually assigns code reviewers to check the submitted code. However, assigning code reviewers may cause two potential problems. First, it is difficult to find suitable reviewers for code review, which may lead to delays in code submission. Secondly, the professional skills and experience of reviewers do not match the code review, which may affect the quality of code review [7,8]. For example, Thongtanunam et al. [9] found that about 4–30% of code reviews cannot find suitable reviewers, and finding available code reviewers usually takes an average of 12 days.

To address the above problems, both academia and industry have proposed a lot of valuable code reviewer recommendation approaches [9–17]. However, most of the previous

approaches recommended reviewers mainly focus on the reviewers' experience characteristics, historical interactions, and code path similarity. For example, Rahman et al. [15] focused on modeling the expertise of the reviewers. Ying et al. [16] leveraged the text similarity of review requests and the social relations of reviewers to find the appropriate reviewers. Thongtanunam et al. [9] leveraged the similarity of file paths to recommend an appropriate code reviewer. Xia et al. [14] explored the text similarity of the review request and the similarity of changed file paths. Although most of those approaches have greatly improved the efficiency of code reviewer recommendations. They still suffer from two critical problems. The first is the data-sparsity problem. For example, according to the survey in [17], only a small percentage of developers can obtain the recommendation opportunity to review the code. Because the interactions between code and candidate reviewers are very sparse, which leads to poor recommendation accuracy. The second is the information limitation and noise problem. For example, according to our investigation, the information in the code title and label is very limited. Previous approaches did not perform deep semantic analysis on more useful information (e.g., code title, label, body). Therefore, how to capture more useful information from code to improve recommendation accuracy is a challenging problem.

In this paper, to alleviate the data-sparsity, information-limitation, and noise problems, we propose a novel attentive neighbor embedding propagating approach (ANEP). This approach is mainly motivated by the following observations. First, reviewers who have reviewed the same codes can be viewed as neighbors. We can establish a connection graph with these reviewers. In this connection graph, reviewers can be viewed as nodes, and connections between reviewers can be viewed as edges. The connection graph contains a collaborative filtering signal along the reviewer's neighbors. Second, reviewer representation can be enhanced by considering the neighbor reviewers in the graph. Intuitively, neighbor embeddings supply information about a reviewer as they have the similarities between a reviewer and their neighbors. Incorporating such similarities may further enrich target reviewer representations, and so does code representation. Third, previous approaches usually suffer from data-sparsity problems. To the best of our knowledge, this is the first attempt at exploiting the text semantic information of the code body. It can enrich the representations of reviewers and code and improve the performance of recommendations. The major contributions can be summarized as follows:

- First, we propose to leverage the transformer to learn the text semantics of the code body and then generate the code semantic representations. In particular, to distill more useful information, we utilize the multi-head attention mechanism to generate the weight of each word. Moreover, we also consider the file paths and label embeddings of code files, which contain important information such as business categories and technical categories.
- Second, to alleviate the data-sparsity problem and enhance the representations of the code and reviewers, we propose to explore the high-order embedding propagation and incorporate their neighbors' representations based on graph attentive network. Moreover, to alleviate the noise problem, we leverage the attention mechanism, which calculates the weights of different nodes and propagation layers in the code-reviewer neighbor network.
- Third, to evaluate the effectiveness of ANEP, we conduct extensive experiments on four real-world datasets. The experimental results show that ANEP outperforms other state-of-the-art approaches effectively. To the best of our knowledge, ANEP is the first attempt at incorporating the text semantics of code into graph embedding learning and propagation to address the code reviewer recommendations.

The remainder of this paper is organized as follows. Section 2 is the related work. Section 3 presents the proposed framework ANEP. Section 4 conducts experiments to evaluate the effectiveness of ANEP. Section 5 is the conclusion and future work.

2. Related Work

Finding appropriate reviewers is a critical and challenging problem in the OSS development community. To solve this problem, many previous approaches focused on explicitly learning the code reviewer's expertise, skill, abilities, contribution, and reputation [4,11,18–22]. They usually measured developers in different ways and aimed to recommend the "best" code reviewers. For example, Wang et al. [18] proposed to model developers' skills and recommend developers based on their skill improvement. Amreen et al. [22] proposed to measure reviewers' reputations and recommend suitable reviewers based on their reputations in the OSS development community. Gousios et al. [20] proposed to measure reviewers' expertise and contributions and integrated it into their recommendation model. Kim et al. [23] proposed to learn the reviewers' expertise based on the LDA model. They compute the reviewer scores based on the topic contribution level and reviewer expertise. Ying et al. [16] proposed to measure the reviewer's authority and expertise and generated the recommendations. However, the above approaches generally suffer from two problems. The first is that they usually ignored the relevance between reviewers and tasks, which may lead to poor recommendation accuracy. The second is that they always chose the "best" reviewers, which may result in only a small percentage of developers could be recommended. For example, according to the investigation in Google [13], it is usually difficult to obtain timely feedback when selecting the "best" reviewers, because the "best" reviewers may be very busy every day. Asthana et al. [24] found that using those approaches may result in only 20% of developers being recommended and most of them have no chance to be recommended.

Other approaches focused on exploring the interaction and similarity between code and reviewers. For example, Yu et al. [25] proposed to mine the social relationship and recommend code reviewers based on similar preferences. Xia et al. [14] proposed calculating the similarity of text and code file paths and recommend code reviewers based on this similarity. Zheng et al. [26] proposed to explore the similarity of reviewers' neighbors. Hirao et al. [27] proposed to construct the historical code-reviewer graphs and recommend suitable code reviewers based on these linkage relations. Thongtanunam et al. [9] proposed identifying the related code based on the file path similarity of the latest submitted code files. They considered that code files with similar file paths may be reviewed by similar code reviewers. Xia et al. [14] proposed to model the file paths and the changed code files to recommend suitable code reviewers. However, the limitation of those approaches is that when the explicit interaction and similarity between the code and reviewers are sparse, their performance may be poor. Another line of approaches [24,28–30] considered the ownership between code and reviewers. They usually focused on measuring the contribution of reviewers to the submitted code and preferred to select those reviewers who have made contributions to the code. For example, Asthana et al. [24] proposed to measure code ownership and integrate it into their recommendation model. Zanjani et al. [29] proposed to analyze the code ownership and generate the reviewers' characteristic model. However, those approaches may also suffer from two problems. The first is that code ownership is usually owned by a small percentage of developers. Therefore, it is difficult to find more available reviewers. The second is that reviewing code by code owners may not guarantee the diversity and credibility of the review results.

In general, the common limitation of those previous approaches is that they all ignore the text semantics hidden in the code body, which may lead to their poor recommendation accuracy. In this paper, we try to learn the text semantics of the code body and generate the code semantic representations and improve the recommendation accuracy.

3. Methodology

In this section, we first illustrate the overall framework of ANEP and its input and output in Section 3.1. Then, we illustrate the embedding learning of the code and reviewers in Section 3.2. In Section 3.3, we further explore the high-order embedding propagation

from the neighbors of code and reviewers. Finally, we generate the prediction model and recommend the Top-k reviewers in Section 3.4.

3.1. Framework

Figure 1 illustrates the overall framework of ANEP. Given a segment of code c and a reviewer r, ANEP outputs the possibility that the reviewer r can be selected to review the code c. As shown in Figure 1, the framework consists of three parts: ① learning the embeddings of code and reviewers; ② exploring the high-order embedding propagation and aggregation; and ③ calculating the prediction model and recommending suitable code reviewers. In the first part, we first construct the interaction graph from the historical interaction of reviewers and then learn the representations of the code and reviewer (as will be illustrated in Section 3.2). Here, a reviewer and one segment of code which they have reviewed can be regarded as two adjacent nodes in the interaction graph. The edge between them indicates that the reviewer has reviewed the code. We extract the neighbors of the code and reviewers from the interaction graph. In the second part (as shown in the red dot line in Figure 1), we explore the high-order embedding propagation and aggregation along the neighbors of the code and reviewers. For example, for a given reviewer r, we first treat his or her reviewed code as the initial seeds and then we propagate the high-order embedding of the seeds and stack the embeddings of their neighbors to generate the aggregated embeddings. We extended the seeds along the edges to form the reviewer embedding set L_r^k, and $k = 1, 2, \ldots, H$. The reviewer set L_r^k is the set of the reviewer embeddings that are k-hops away from the seed set L_r^0. Finally, in the third part, ANEP generates the final embeddings of the code and reviewers and calculates the predicted probability $\hat{y}$. The symbols utilized in this paper are shown in Table 1.

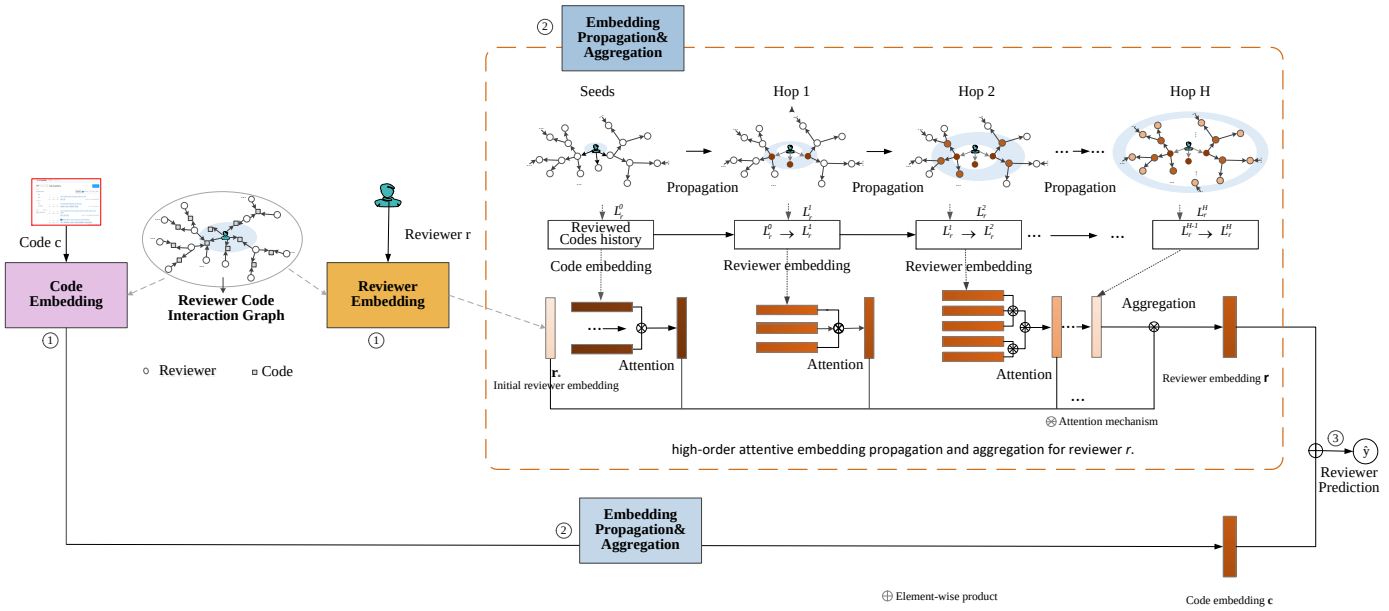


Figure 1. Illustration of the overall framework of ANEP. It takes one code segment and one candidate reviewer as input and outputs the possibility that the code will be assigned to the reviewer. The part inside the dotted line illustrates the high-order embedding propagation for the reviewer. The set L_r^0, $L_r^1, \ldots$, and L_r^H are activated by the reviewer's historical review.

Table 1. The symbols utilized in this paper.

Notations	Description
r	reviewer
r_1	reviewer r's 1-hop neighbor
L_r^0, L_r^k	reviewer embedding propagation set
e_r, e_{r_1}	reviewer embedding
N_r	1-hop neighbors of reviewer r
N_{r_1}	1-hop neighbors of reviewer r_1
$f(\cdot)$	information encoding function
$m_{r \leftarrow r_1}$	embedding propagated from r_1 to r
$m_{r \leftarrow r_1}^{(H)}$	embedding propagated from r's H-hop neighbors
${e_r}^{(1)}$	refined representation of reviewer r from r's 1-hop neighborhood
${e_r}^{(H)}$	refined representation of reviewer r from r's H-hop neighborhood
W_1, W_2	the trainable weight matrices
${W_1}^{(H)}$, ${W_2}^{(H)}$	the trainable transformation matrices
${e_r}^{(*)}$	the representations of r after H layer propagation
${e_c}^{(*)}$	the representations of the code c after H layer propagation
${e_r}^{(0)}$	initial representation of r
${e_c}^{(0)}$	initial representation of code c
${e_c}^{(H)}$	refined representation of code c from c's H-hop neighbors
$\hat{y}(r, c)$	reviewer's preference towards the target code
h_i^k	the representation of the i-th word with the k-th attention head
$\alpha_{i,j}^k$	the importance of the i-th and j-th words
W_s^k, W_v^k	projection matrices
β_i^w	attention weight of the i-th code body word
c_b, c_p, c_l	the representation of code body, path and label
γ_i^n	attention weight of the i-th
r_c	reviewer representation from code review history
r_l	reviewer label representation
q_n, S_n, s_n	the trainable parameters of attention weight

3.2. Representations of Code and Reviewer

3.2.1. Code Embedding

Inspired by Ge et al. [31], we use the transformer to generate a code representation from the code body content, titles, labels, and code path. The code body usually clearly describes the code content. Here, we leverage the simplified single-layer multi-head attention and transformer to generate the embedding vectors. Figure 2 shows the structure of the transformer, where the bottom embedding layer of the code body is the word embedding. It converts the code body words into a sequence of low-dimensional embedding vectors. In particular, given a piece of code body with M words, we convert the code body to a sequence of embedding vectors ($[e_1, e_2, \ldots, e_M]$) based on the transformer model.

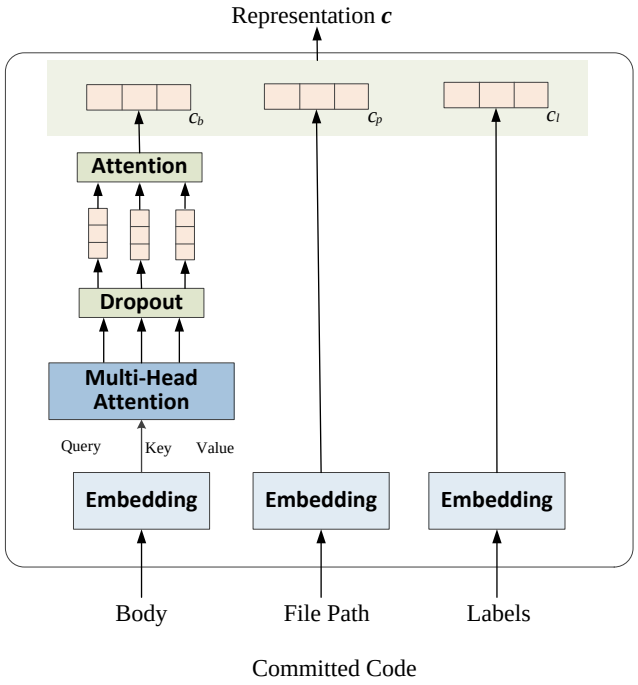


Figure 2. The code semantic representation of ANEP. It takes the code body, code file path, and code labels as input and outputs the representation of the code.

The next part is a word-level multi-head self-attention module. Enriching code representations is important to improve the accuracy of recommendations. We enrich code representations by collecting interaction information between words in the text content. We adopt a multi-head self-attention mechanism to generate code body contextual word representations. In this work, we compute the representation of the i-th word with the k-th attention head as:

$$h_i^k = W_v^k(\Sigma_{j=1}^M \alpha_{i,j}^k e_j), \tag{1}$$

$$\alpha_{i,j}^k = \frac{exp(e_i^T W_s^k e_j)}{\sum_{m=1}^M exp(e_i^T W_s^k e_m)}, \tag{2}$$

where W_s^K and W_v^k are the projection matrices. $\alpha_{i,j}^k$ is the relative importance of the relatedness between the ith and jth words in the code body content. h_i is a multi-head representation of the i-th word. It is concatenated by N separate attention heads. We set $h_i = [h_i^1; h_i^2; \ldots; h_i^N]$. We also add the dropout [32] module after the multi-head attention module to relieve overfitting.

In the next step, we model the relative importance of different words utilizing a word attention module to generate code body representation. We set β_i^w as the attention weight of the i-th word:

$$\beta_i^w = \frac{exp(q_w^T tanh(S_w \times h_i + s_w))}{\sum_{j=1}^M exp(q_w^T tanh(S_w \times h_j + s_w))}, \tag{3}$$

Here, q_w, S_w, and s_w are trainable parameters. The code body representation c_b is computed as $c_b = \sum_{i=1}^M \beta_i^w h_i$.

Since code path and label are also important. They reveal code information, such as relevant expertise and business area, and so on. For example, if the source code of a program waiting for review is about a kind of exchange rate conversion method in the accounting field, then its file path includes the word "exchange" and the source code may also include the label "exchange". We model code path and label via an embedding matrix.

Denote the output of the path representation and the label representation as c_p and c_l, respectively, and the final code representation is concatenated of the body representation, path representation, and topic representation, i.e., $c = [c_b; c_p; c_l]$.

3.2.2. Reviewer Embedding

The characteristics and expertise of a reviewer can be revealed from the code review history of the reviewer. Therefore, we model reviewer representations from the body content of the code that has been reviewed by the reviewer. On the other hand, the importance of each viewed code is different for modeling reviewer representation. For example, a section of code content "This is a java interface that obtains user basic data." is less important than the code content "C++ function: a method of forecast stock trend" in modeling reviewer representation. Thus, we utilize the attention mechanism to integrate reviewed code representations for reviewer representations. For a target reviewer r and K pieces of reviewed code, we first obtain transformer encoded code representation outputs [c_1,c_2, ...,c_K], then compute the attention weight γ_i^n of the i-th reviewed code as:

$$\gamma_i^n = \frac{exp(q_n^T tanh(S_n \times c_i + s_n))}{\sum_{j=1}^{K} exp(q_n^T tanh(S_n \times c_j + s_n))}, \tag{4}$$

where q_n, S_n, and s_n are the trainable parameters of the attention weight module. The code reviewer review history representation r_c, which is structured from the viewed code, is then computed as $r_c = \sum_{i=1}^{K} \gamma_i^n c_i$. Reviewer labels are also very important as they not only can reveal reviewer expertise information such as Java, Python, and so on, but also reveal business areas the reviewers are good at. In fact, the business areas in the label where the reviewer is good are also extremely important. A good code reviewer or software developer should not only have the necessary technical capabilities, but also have sufficient relevant business capabilities. For example, to realize the exchange rate conversion function in the accounting area, the reviewer who understands the accounting-related functions is more likely to find potential logic problems in the program, thereby improving the review quality. Denote the reviewer label representation as r_l, and the final reviewer representation is concatenated of the review history representation and label representation, i.e., $r = [r_c; r_l]$.

3.3. High-Order Embedding Propagation

We refine a reviewer embedding by injecting its high-order connectivity relations of multiple embedding propagation layers in ANEP. This structure can capture a neighbor-embedding signal along the reviewer-neighbor-embedding graph. Intuitively, neighbor embeddings supply information about a reviewer as it has similarities between a reviewer and his neighbors. Based on this intuition, we perform embedding propagation between the connected reviewer and his neighbors. For connected reviewer r and his 1-hop neighbor r_1 pair (r,r_1), we define the information from r_1 to r as:

$$m_{r \leftarrow r_1} = f(e_{r_1}, e_r, p_{rr_1}), \tag{5}$$

Here, $m_{r \leftarrow r_1}$ is the signal embedding to be propagated. $f(\cdot)$ is the information encoding function, e_r and e_{r_1} are input reviewer embeddings, p_{rr_1} is the coefficient, which is a decay factor to control the propagation of each propagation $edge(r, r_1)$. We define function $f(\cdot)$ as below:

$$m_{r \leftarrow r_1} = \frac{1}{\sqrt{|N_r||N_{r_1}|}}(W_1 e_{r_1} + W_2(e_{r_1} \odot e_r)), \tag{6}$$

In the equation above, W_1, $W_2 \in \mathbb{R}^{d \times d}$, W_1 and W_2 denote the trainable weight matrices. They obtain useful propagation messages. We add element-wise product interaction $\odot$ between e_r and e_{r_1} to propagate more information through the same embedding. To represent the discount factor of information being propagated with the path length, we set p_{rr_1} as graph Laplacian norm $1/\sqrt{|N_r||N_{r_1}|}$, where N_r and N_{r_1}, respectively, define the

first-hop neighbors of reviewer r and r_1. This method can increase the representation effect of embedding and the performance of the recommendation.

In ANEP, to refine the representation of r from r's neighborhood, we set the aggregation function to aggregate the information from r's neighborhood. Take a 1-hop neighborhood for example:

$$e_r^{(1)} = LeakyReLU(m_{r\leftarrow r} + \sum_{r_1 \in N_r} m_{r\leftarrow r_1}), \tag{7}$$

where $e_r^{(1)}$ is the embedding of r. The representation information of r is aggregated from the first-order propagation of its neighbors. We choose the LeakyReLU function as an activation function to encode both positive and small negative information. In order to obtain the original features of r, we set $m_{r\leftarrow r} = W_1 e_r$, and W_1 is the weight matrix as denoted before in Equation (2). $m_{r\leftarrow r}$ represents the self-connection information of r.

Analogously, we can iterate the operation of embedding propagation to acquire reviewer r's high-order propagation information. The operation can be performed iteratively on reviewer r's H-hop neighbors as Equation (4) shows below:

$$e_r{}^{(H)} = LeakyReLU(m_{r\leftarrow r}^{(H)} + \sum_{r_1 \in N_r} m_{r\leftarrow r_1}^{(H)}), \tag{8}$$

where the information being propagated is calculated as:

$$m_{r\leftarrow r_1}^{(H)} = \frac{1}{\sqrt{|N_r||N_{r_1}|}}(W_1^{(H)} e_{r_1}^{(H-1)} + W_2^{(H)}(e_{r_1}^{(H-1)} \odot e_r^{(H-1)})), \tag{9}$$

$$m_{r\leftarrow r}^{(H)} = W_1^{(H)} e_r^{(H-1)}, \tag{10}$$

In the above equations, W_1 and W_2 refer to the trainable transformation matrices with useful propagation information, and $e_r^{(H-1)}$ is the r's representation, which is obtained from the previous information propagation phase. It contains the information from its $(H-1)$-hop neighbors. Then, $e_r^{(H-1)}$ propagates the information to the representation of reviewer r at layer H. We can also perform the computation of $e_r^{(H)}$ at layer H the same as $e_r^{(H-1)}$.

3.4. Code Reviewer Recommendation

After H layers of propagation, we obtain a multiple-representations set for reviewer r, which is $\{e_r^{(0)}, \ldots, e_r^{(H)}\}$. Representations obtained in different layers contribute different signals to the reviewer's characteristics. Therefore, we combine them to obtain the final embedding for a reviewer. The propagation procedure can also be performed on code, and concatenating the code representations $\{e_c^{(0)}, \ldots, e_c^{(H)}\}$ obtained from different layers to obtain the final code embedding:

$$e_r{}^* = e_r{}^{(0)}|| \ldots ||e_r^{(H)}, e_c{}^* = e_c{}^{(0)}|| \ldots ||e_c{}^{(H)}, \tag{11}$$

where $||$ denotes the concatenation operation to combine the responses of all layers. By concatenation, we enrich the initial reviewer and code embedding with information from other embedding propagation layers and be able to control the scope of propagation by tuning layer number H simultaneously. In fact, besides concatenation, there are several other aggregators, such as LSTM, weighted average, etc. They all can be applied to conduct different assumptions in combining embeddings in different layers. We choose the concatenation operation as its simplicity, which involves no additional parameters to learn. In addition, concatenation operation is extremely effective, which is proved in the work

of graph neural networks [33]. At last, we employ the inner product to estimate the code developer's preference toward the target code viewer:

$$\hat{y}(r, c) = {e_r}^{*\mathrm{T}} {e_c}^*, \tag{12}$$

Here, we only utilize the simple inner product as an interaction function, as we just emphasize the embedding function learning in this work. With regard to other more complicated interaction functions, such as neural-network-based interaction functions and so on, we will study them in the future.

4. Experiments

To verify the performance of ANEP, we conduct a set of experiments. First, we build the code-neighbor graph based on the datasets, then compare the performance of ANEP with other baseline approaches. Here, we aim to answer the following two questions:

- RQ1: How does the proposed ANEP perform compared with the existing reviewer recommendation approaches?
- RQ2: How does the embedding propagation layer effect ANEP?

4.1. Datasets

In this work, we take large open-source software projects as the datasets, i.e., Android (https://android-review.googlesource.com accessed on 1 September 2022) , Qt (https://codereview.qt-project.org accessed on 7 September 2022), OpenStack (https://review.openstack.org accessed on 10 September 2022), and LibreOfce https://gerrit.libreoffice.org accessed on 12 September 2022). They are also often used by other modern code review approaches. These projects contain a lot of code review history data. For Qt, Android, and OpenStack, the same as prior studies [9,34,35], we use the code review datasets of Hamasaki et al. [36]. For LibreOfce, we use the review dataset of Yang et al. [37]. The datasets include code information, review comment content, and reviewer information. These datasets have a large number of projects recorded in the code review tool. The information in the datasets includes code text description, code path, subject title, reviewer information, and interaction information between the reviewer and code. We collected information after filtering out the code without a reviewer and split the datasets into training datasets and testing datasets. The first 80% are used for training and the remaining 20% are used for testing. We constructed the code neighbor embedding graph and the reviewer neighbor embedding graph from the interaction information of the reviewer and code.

4.2. Baseline Methods

To evaluate the performance of our ANEP, we choose three existing approaches below to compare the performance:

- RevFinder [9]: This approach proposes to compute the similarity between code file paths that have been reviewed by the target code reviewer and the latest submitted code file paths. They consider that if a code reviewer has reviewed some code in a file path, the code located in similar paths is likely to be reviewed by the same reviewer.
- EARec [16]: This approach not only considers developer expertise, but also developer authority. They first construct a graph of the latest submitted code file and possible reviewers, and then calculate the text similarity of the latest submitted code file and social relations of reviewers to find suitable reviewers.
- MulTO [38]: This multi-objective approach proposes to not only evaluate reviewer expertise, but also consider estimating reviewer availability based on workload. They consider the review workload is also an influence important factor for a reviewer to decide if they will accept a review task.

4.3. Parameter Settings

In our experiment, we set the embedding size to 64. The initial model parameters of ANEP are created based on Xavier, which is a widely applied neural network initializer.

We use the grid search method [39] to set hyper-parameters. Set learning rate η as $\{0.01, 0.001, 0.0001\}$, $L2$ normalization is set as $\{10^{-4}, 10^{-3}, \ldots, 10^{1}, 10^{2}\}$. For the overfitting problem, we utilize the node dropout method [40] to solve it. Here, the dropout ratio is tuned in $\{0.1,0.2,0.3,0.4,0.5,0.6,0.7\}$. As for the number of layers, we set the default layer number L to 3 as the experiment effect was the best when layer number L was 3 (evidence in Section 4.7). Because the experimental results may be random, in order to obtain accurate experimental results, following the previous approach [41], we repeat each independent experiment more than 30 times.

4.4. Evaluation Metrics

We choose precision ratio precision@k, recall@k, and normalized discounted cumulative gain ndcg@k [42] to construct evaluation metrics to evaluate ANEP. This is widely used in the recommendation approach. They are defined as:

$$precision@k = \frac{| \ ActualRev \cap RecRev \ |}{| \ RecRev \ |}, \tag{13}$$

$$recall@k = \frac{| \ ActualRev \cap RecRev \ |}{| \ ActualRev \ |}, \tag{14}$$

$$ndcg@k = \frac{DCG@k}{IDCG@k}, \left(DCG@k = \sum_{i=1}^{k} \frac{2^{f_i} - 1}{\log_2(i+1)} \right) \tag{15}$$

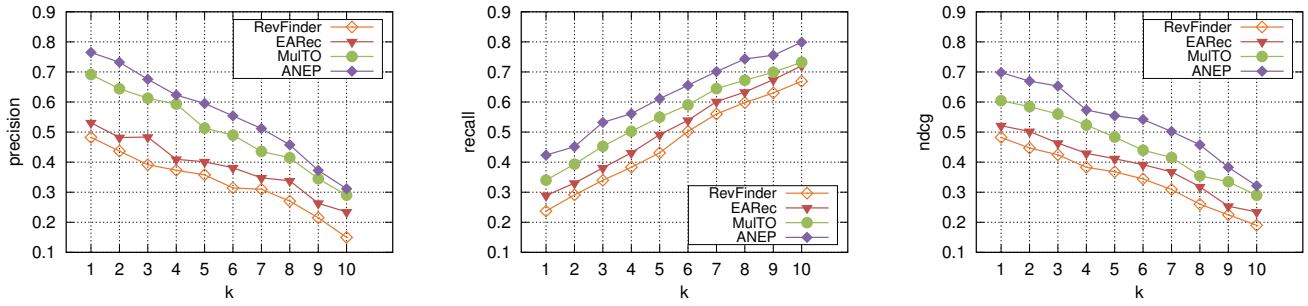
where *RecRev* refers to the recommended reviewers, and $k = |RecRev|$. *ActualRev* denotes the actual reviewers of the code in the datasets. Normalized discounted cumulative gain ndcg is used to evaluate sorting results. It will assign greater weight to the top elements in the list and discount the items at the bottom of the list. In (15), following the previous approach [42], f_i is the relevance of the reviewer r at position i. When r exists in the test dataset, $f_i = 1$, otherwise $f_i = 0$. IDCG represents the list of the ideal recommendation results, that is, DCG under perfect results. The related reviewers will be placed in front of the recommendation list, e.g., when we recommend $k = 6$ reviewers, and the 2-st, 4-st, and 6-st reviewers exist in the test dataset. DCG@6 can be computed based on $\{0, 1, 0, 1, 0, 1\}$, then the ideal result can be computed based on $\{1, 1, 1, 0, 0, 0\}$. Besides, in our datasets, over 98% of code has 1 to 10 reviewers, so k is set from 1 to 10.

4.5. Statistical Test Method

We use the Wilcoxon signed-rank test method [41] to verify the significant difference between ANEP and other reviewer recommendation approaches. Wilcoxon signed-rank test is a non-parametric test method. It has an advantage over the parametric test method. The non-parametric test method does not require the normal distribution of the data. Because the overall distribution of the dataset is usually unknown, and the sample size is not large enough to use the central limit theorem to perform parameter testing, it can not infer the overall central tendency and the parameters dispersion degree. The reason why we choose the Wilcoxon signed rank test method as the statistical test method is that the overall distribution of the dataset is usually unknown, and the data are continuous and in a pair-wise fashion. We can draw a conclusion that our approach is significantly different from the others if the p-value < 0.05 (https://en.wikipedia.org/wiki/P-value accessed on 16 April 2023). It not only uses the positive and negative difference between the observed value and the original hypothesis center position, but also uses information about the size of the difference value. In addition, we utilize Cliff's delta (https://rdrr.io/cran/rcompanion/man/cliffDelta.html accessed on 16 April 2023) to compute the effect size d. The effect size is considered to be large enough (https://en.wikipedia.org/wiki/Effect_size accessed on 16 April 2023) when $|d| \geq 0.474$.

4.6. Performance Comparison (RQ1)

To demonstrate the effectiveness of ANEP, we compare the overall performance of ANEP with other state-of-the-art code reviewer recommendation approaches. The experimental results are shown in Figure 3, Tables 2 and 3, and we have the following observations.



(a) precision. (b) recall. (c) ndcg.

Figure 3. Comparison of precision, recall, and ndcg of top-*k* recommendation (*k* is from 1 to 10).

Table 2. Average performance comparison in precision, recall, and ndcg (*k* is from 1 to 10). Where "Imprv.↑" denotes the average performance improvement of ANEP compared to other approaches.

Evaluation Metrics	RevFinder	EARec	MulTO	ANEP	Imprv.↑
precision	0.3300	0.3872	0.5029	0.5598	10.17%
recall	0.4642	0.5091	0.5576	0.62336	10.55%
ndcg	0.3432	0.3892	0.4589	0.53539	14.27%

Table 3. Statistical test result between ANEP and other approaches (*k* is from 1 to 10), L is $|d| \geq 0.474$).

ANEP	Precision		Recall		ndcg	
vs.	Mediam	*p*-Value(d)	Mediam	*p*-Value(d)	Mediam	*p*-Value(d)
RevFinder	0.3359	<0.05(L)	0.4668	<0.05(L)	0.3659	<0.05(L)
EARec	0.3914	<0.05(L)	0.5153	<0.05(L)	0.4014	<0.05(L)
MulTO	0.5014	<0.05(L)	0.5699	<0.05(L)	0.4614	<0.05(L)
ANEP	0.5746	-	0.6333	-	0.5482	-

First, Figure 3 presents the top-*k* performance (*k* is from 1 to 10) including recision@k, recall@k, and ndcg@k. The result demonstrates that ANEP consistently performs best in code reviewer recommendation compared with the other three existing approaches. We present the average performance of the four approaches in Table 2. We can see from the average precision, recall, and ndcg that ANEP outperforms all the other approaches with respect to precision by at least 10.17%, recall by 10.55%, and ndcg by 14.27%, respectively. Table 3 presents the statistical test results of ANEP, RevFinder, EARec, and MulTO. According to the statistical test method (described in Section 4.5), we can see that ANEP has significant differences compared with RevFinder, EARec, and MulTO (i.e., p-value < 0.05 and $|d| \geq 0.474$). In particular, take the p-value for example. For precision, the p-value between ANEP and RevFinder, EARec, and MulTO are 0.0048, 0.0049, and 0.0051, respectively. For recall, the p-value between ANEP and RevFinder, EARec, and MulTO are 0.0050, 0.0053, and 0.0058, respectively. For ndcg, the p-value between ANEP and RevFinder, EARec, and MulTO are 0.0051, 0.0054, and 0.0057, respectively. The above experimental results demonstrate the effectiveness of our ANEP approach. By using the attentive neighbors embedding propagation model, ANEP can extract more useful reviewer and code information

effectively. So, data sparsity and noise issues will be greatly alleviated. Meanwhile, it can optimize the recommendation accuracy. Moreover, ANET can incorporate the reviewer's business and expertise information from the body and label of code. It can improve the accuracy and quality of code reviewer recommendations effectively.

Second, we can see that the experimental results of MulTO are more accurate than RevFinder and EARec in most cases. This may be because MulTO not only simultaneously considers reviewer expertise and history of collaborations for reviewer recommendations, but also simultaneously considers reviewer availability. However, MulTO does not enhance reviewer or code representations by aggregating neighbor representations, which makes its accuracy not as good as ANEP. Unlike MulTO, ANEP can refine reviewer and code representations by integrating neighbor representations through a graph attention network. It leverages the high-order embedding propagation and generates information from different nodes and different network propagation layers according to the attentive weights.

Third, we can see that among all comparison approaches, RevFinder achieves relatively poor performance. For example, the average precision, recall, and ndcg of RevFinder are only 0.3300, 0.4642, and 0.3432, respectively. It may be because that RevFinder only takes the reviewer recommendation problem as a simple similarity problem between the previously reviewed file path and code path. It ignores the expertise, business, history of collaborations message extracted from the body, and the label of representation.

4.7. Effect of Embedding Propagation Layer Numbers (RQ2)

In this section, we investigate how embedding propagation layer numbers affect the recommendation performance of ANEP. We varied the model depth, increasing the layer numbers from 1 to 4. We summarize the experimental results in Table 4. Here, the ANEP-1 column presents the precision, recall, and ndcg results in the model with one embedding propagation layer. Notations ANEP-2 and ANEP-3 are similar to ANEP-1. Jointly analyzing Tables 2 and 4, we have the observations below.

Table 4. Effect of embedding propagation layer numbers.

Evaluation Method	ANEP-1	ANEP-2	ANEP-3	ANEP-4
precision	0.4381	0.4965	0.5842	0.5882
recall	0.4423	0.5329	0.6421	0.6485
ndcg	0.4047	0.4768	0.5545	0.5378

First, we can see that when we increase the number of the embedding propagation layers, it can significantly improve the recommendation performance. For example, the experimental results in Table 4 show that the performance of ANEP-2 and ANEP-3 is steadily better than ANEP-1. Because ANEP-1 only injects the first-order neighbor information. We attribute the improvement to effective aggregating neighbor representations via a graph attention network. In the network, the second and third-order nodes are similar to the original node and they propagate the signal.

Second, we can see that when increasing the propagation layer number to 4, the experimental results of ANEP-4 may not perform as well as we thought. This may be because applying deeper network structures may introduce more noise to the representation learning and cause overfitting problems. This demonstrates that applying three propagation layers is sufficient to draw neighbor information to do reviewer recommendations.

Third, we can see that even if the number of propagation layers is changed, the performance of ANEP consistently exceeds other recommended approaches across all datasets in most cases. It also verifies the effectiveness of ANEP. The experimental results show that applying a high-order attentive neighbor embedding propagating model to capture neighbor collaborative filtering signals can greatly improve the performance of code reviewer recommendations.

5. Conclusions

In this paper, we proposed ANEP, a novel attentive semantic representation learning and high-order embedding propagation framework for code reviewer recommendation. We explicitly learn the semantic representations of code body and reviewers based on the transformer model and multi-head attention mechanism. Moreover, to alleviate the interaction sparsity between code and reviewers, we leverage the high-order embedding propagation to enhance the representation learning of code and reviewers. To the best of our knowledge, ANEP is the first attempt at incorporating text semantics of code into graph embedding learning for code reviewer recommendation. It effectively addresses the interaction sparsity, information limitation, and noise problem.

To evaluate the performance of ANEP, we conduct extensive experiments on four large real-world datasets. The experimental results show that ANEP outperforms the state-of-the-art approaches up to 10.17%, 10.55%, and 14.27% on precision, recall, and ndcg, respectively. In future work, we plan to further explore the usefulness of ANEP and consider incorporating more factors such as the reviewer's workload, willingness, expertise, response time, and review quality into the reviewer's representation learning. We hope to incorporate these factors into ANEP to improve the overall collaboration efficiency of code review. Furthermore, to improve the recommendation accuracy, ANEP has to collect code information, reviewer information, and interaction information between code and reviewer. Then, construct the code neighbor embedding graph and the reviewer neighbor embedding graph from the interaction information of the reviewer and code. The process of data collection and graph construction is quite complex. This has a certain influence on the popularization effect of the method. So, we also plan to consider alleviating this efficiency problem in future work.

Author Contributions: Conceptualization, J.L. and A.D.; methodology, J.L. and A.D.; software, Q.X. and G.Y.; validation, J.L.; formal analysis, G.Y.; investigation, J.L. and Q.X.; writing—original draft preparation, J.L.; writing—review and editing, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: National Natural Science Foundation of China (61502068), Professor ANSHENG DENG.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Tufan, R.; Pascarella, L.; Tufanoy, M.; Poshyvanykz, D.; Bavota, G. Towards Automating Code Review Activities. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; pp. 163–174.
2. Morales, R.; McIntosh, S.; Khomh, F. Do code review practices impact design quality? A case study of the qt, vtk, and itk projects. In Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, 2–6 March 2015; pp. 171–180.
3. Bavota, G.; Russo, B. Four eyes are better than two: On the impact of code reviews on software quality. In Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 29 September–1 October 2015; pp. 81–90.
4. McIntosh, S.; Kamei, Y.; Adams, B.; Hassan, A.E. An empirical study of the impact of modern code review practices on software quality. *Empir. Softw. Eng.* **2016**, *21*, 2146–2189. [CrossRef]
5. Thongtanunam, P.; McIntosh, S.; Hassan, A.E.; Iida, H. Investigating code review practices in defective files: An empirical study of the qt system. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 168–179.
6. Alami, A.; Cohn, M.L.; Wasowski, A. Why does code review work for open source software communities? In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 1073–1083.
7. Çetin, H.A.; Doğan, E.; Tüzün, E. A review of code reviewer recommendation studies: Challenges and future directions. *Sci. Comput. Program.* **2021**, *208*, 102652. [CrossRef]
8. MacLeod, L.; Greiler, M.; Storey, M.A.; Bird, C.; Czerwonka, J. Code reviewing in the trenches: Challenges and best practices. *IEEE Softw.* **2017**, *35*, 34–42. [CrossRef]

9. 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 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, 2–6 March 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 141–150.
10. Bacchelli, A.; Bird, C. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2–6 March 2013; pp. 712–721.
11. Kovalenko, V.; Tintarev, N.; Pasynkov, E.; Bird, C.; Bacchelli, A. Does reviewer recommendation help developers? *IEEE Trans. Softw. Eng.* **2018**, *46*, 710–731. [CrossRef]
12. Rigby, P.C.; Bird, C. Convergent contemporary software peer review practices. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 202–212.
13. Sadowski, C.; Söderberg, E.; Church, L.; Sipko, M.; Bacchelli, A. Modern code review: A case study at google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, Gothenburg Sweden, 27 May–3 June 2018; pp. 181–190.
14. 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 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 29 September–1 October 2015; pp. 261–270.
15. Rahman, M.M.; Roy, C.K.; Collins, J.A. Correct: code reviewer recommendation in github based on cross-project and technology experience. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 222–231.
16. Ying, H.; Chen, L.; Liang, T.; Wu, J. Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In Proceedings of the 2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE), Austin, TX, USA, 16 May 2016; pp. 29–35.
17. Xie, X.; Yang, X.; Wang, B.; He, Q. DevRec: Multi-Relationship Embedded Software Developer Recommendation. *IEEE Trans. Softw. Eng. (TSE)* **2021**, *48*, 4357–4379. [CrossRef]
18. Wang, Z.; Sun, H.; Fu, Y.; Ye, L. Recommending crowdsourced software developers in consideration of skill improvement. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), Champaign, IL, USA, 30 October–3 November 2017; pp. 717–722.
19. Bosu, A.; Greiler, M.; Bird, C. Characteristics of useful code reviews: An empirical study at microsoft. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 146–156.
20. Gousios, G.; Pinzger, M.; Deursen, A.v. An exploratory study of the pull-based software development model. In Proceedings of the IEEE/ACM 36th International Conference on Software Engineering (ICSE), Hyderabad, India, 31 May–7 June 2014; pp. 345–355.
21. 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 (ASE), Singapore, 3–7 September 2016; pp. 99–110.
22. Amreen, S.; Karnauch, A.; Mockus, A. Developer Reputation Estimator (DRE). In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1082–1085.
23. Kim, J.; Lee, E. Understanding review expertise of developers: A reviewer recommendation approach based on latent Dirichlet allocation. *Symmetry* **2018**, *10*, 114. [CrossRef]
24. 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tallinn, Estonia, 26–30 August 2019; pp. 937–945.
25. 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? *Inf. Softw. Technol.* **2016**, *74*, 204–218. [CrossRef]
26. Xia, Z.; Sun, H.; Jiang, J.; Wang, X.; Liu, X. A hybrid approach to code reviewer recommendation with collaborative filtering. In Proceedings of the IEEE International Workshop on Software Mining (SoftwareMining), Champaign, IL, USA, 3 November 2017; pp. 24–31.
27. Hirao, T.; McIntosh, S.; Ihara, A.; Matsumoto, K. The review linkage graph for code review analytics: A recovery approach and empirical study. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tallinn, Estonia, 26–30 August 2019; pp. 578–589.
28. Mirsaeedi, E.; Rigby, P.C. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 5–11 October 2020; pp. 1183–1195.
29. Zanjani, M.B.; Kagdi, H.; Bird, C. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Trans. Softw. Eng. (TSE)* **2016**, *42*, 530–543. [CrossRef]
30. Lipcak, J.; Rossi, B. A large-scale study on source code reviewer recommendation. In Proceedings of the 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Prague, Czech Republic, 29–31 August 2018; pp. 378–387.

31. Ge, S.; Wu, C.; Wu, F.; Qi, T.; Huang, Y. Graph enhanced representation learning for news recommendation. In Proceedings of the Web Conference 2020, Taipei, Taiwan, 20–24 April 2020; pp. 2863–2869.
32. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
33. Wu, S.; Sun, F.; Zhang, W.; Xie, X.; Cui, B. Graph neural networks in recommender systems: A survey. *ACM Comput. Surv. (CSUR)* **2020**, *55*, 1–37. [CrossRef]
34. Ouni, A.; Kula, R.G.; Inoue, K. Search-based peer reviewers recommendation in modern code review. In Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, 2–7 October 2016; pp. 367–377.
35. Ruangwan, S.; Thongtanunam, P.; Ihara, A.; Matsumoto, K. The impact of human factors on the participation decision of reviewers in modern code review. *Empir. Softw. Eng.* **2019**, *24*, 973–1016. [CrossRef]
36. Hamasaki, K.; Kula, R.G.; Yoshida, N.; Cruz, A.C.; Fujiwara, K.; Iida, H. Who does what during a code review? Datasets of oss peer review repositories. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; pp. 49–52.
37. Yang, X.; Kula, R.G.; Yoshida, N.; Iida, H. Mining the modern code review repositories: A dataset of people, process and product. In Proceedings of the 13th International Conference on Mining Software Repositories, Austin, TX, USA, 14–22 May 2016; pp. 460–463.
38. Rebai, S.; Amich, A.; Molaei, S.; Kessentini, M.; Kazman, R. Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations. *Autom. Softw. Eng.* **2020**, *27*, 301–328. [CrossRef]
39. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. The impact of automated parameter optimization on defect prediction models. *IEEE Trans. Softw. Eng.* **2018**, *45*, 683–711. [CrossRef]
40. Wang, X.; He, X.; Cao, Y.; Liu, M.; Chua, T.S. Kgat: Knowledge graph attention network for recommendation. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Anchorage, AK, USA, 4–8 August 2019; pp. 950–958.
41. Arcuri, A.; Briand, L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 1–10.
42. He, X.; Chen, T.; Kan, M.Y.; Chen, X. Trirank: Review-aware explainable recommendation by modeling aspects. In Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, Melbourne, Australia, 18–23 October 2015; pp. 1661–1670.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.