Heterogeneous Graph Neural Network-Based Software Developer Recommendation

Zhixiong Ye[1], Zhiyong Feng[1], Jianmao Xiao[2(✉)], Yuqing Gao[3], Guodong Fan[1], Huwei Zhang[1], and Shizhan Chen[1]

[1] Tianjin University, Tianjin, China
{lailai_zxy,zyfeng,guodongfan,zhuwe,shizhan}@tju.edu.cn
[2] Jiangxi Normal University, Jiangxi, China
jm_xiao@jxnu.edu.cn
[3] Fujian Normal University, Fujian, China
lilyeatcandy@icloud.com

Abstract. In software maintenance, it is critical for project managers to assign software issues to the appropriate developers. However, finding suitable developers is challenging due to the general sparsity and the long-tail of developer-issue interactions. In this paper, we propose a novel **H**eterogeneous **G**raph Neural Network-based method for **D**eveloper **R**ecommendation (called HGDR), in which text information embedding and self-supervised learning (SSL) are incorporated. Specifically, to alleviate the sparsity of developer-issue interactions, we unify developer-issue interactions, developer-source code file interactions and issue-source code file relations into a heterogeneous graph, and we embed text descriptions to graph nodes as information supplements. In addition, to mitigate the long-tail influence, e.g., recommendation bias, the proficiency weight suppression link supplementation is proposed to complement the tail developers by adjusting proficiency weights. Finally, to fully utilize rich structural information of heterogeneous graph, we use the joint learning of metapath-guided heterogeneous graph neural network and SSL to learn the embedding representation. Extensive comparison experiments on three real-world datasets show that HGDR outperforms the state-of-the-art methods by 6.02% to 44.27% on recommended metric. The experimental results also demonstrate the efficacy of HGDR in the sparse and long-tail scenario. Our code is available at https://github.com/1qweasdzxc/HGDR.

Keywords: Developer recommendation · Heterogeneous graph neural network · Information supplement · Self-supervised learning

1 Introduction

In the process of software maintenance and evolution, issues will continue to emerge and accumulate [14]. On the open source platform, users can request

H. Gao et al. (Eds.): CollaborateCom 2022, LNICST 460, pp. 433–452, 2022.
https://doi.org/10.1007/978-3-031-24383-7_24

issues, such as new feature requests and bugs encountered. However, due to the limited familiarity of project managers in assigning issues to developers, or the way developers go to the platform to solve problems on their own, issues are not handled timely or even remain unresolved for a long time [4]. Therefore, how to automatically recommend suitable developers to answer issues in a timely and accurately manner is an important problem with practical needs [16,28].

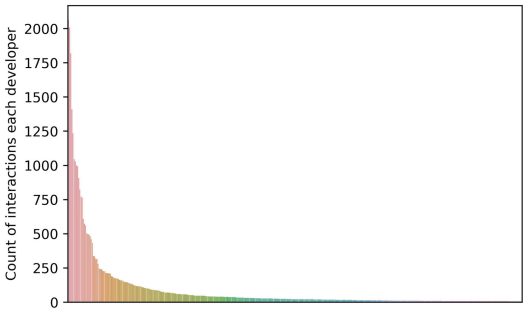


Fig. 1. Distribution of the number of issues solved by developers on the Tensorflow dataset

In recent years, to recommend suitable developers, a lot of efforts have been made in both research and practice. The matrix factorization (MF) [12] approach has been applied to recommend developers to solve issues [20,29,32], which is based on the hypothesis that developers have similar behavior [19,27]. Most of the current research methods focus only on developer-issue interactions and matching, while ignoring some useful information, such as the source code file as an intermediate bridge between the issue and the developer. This leads to low performance of recommendations. In addition, these approaches tend to recommend experienced developers. However, experienced developers are a minority, and it is difficult to solve the large number of issues raised by users timely. We argue that this is owning to two major limitations:

- **Scarcity of developer-issue interaction labels**: The solution to an issue usually involves only 1 to 3 developers, which leads to a sparse label of developer interactions. There may be other developers who are interested and capable of solving these problems, but these labels are difficult to collect.
- **Highly skewed data distribution**: Fig. 1 shows a long-tailed distribution of the number of issues solved by developers. The small number of experienced developers solve the majority of issues, which leads to a tendency to recommend experienced developers and a bias towards junior developers.

To solve the above problems, we find that the relationship between developers, issues, and the source code files can be well modeled by a heterogeneous graph. Recently, heterogeneous graph neural networks (HGNN) have been successful in various fields of processing heterogeneous information network (HIN) [17] data. The heterogeneous graph has a complex topology and rich relational information. Inspired by this, it can be applied to developer recommendation

scenarios with sparse labels. Meanwhile, self-supervised learning (SSL) is used as a method to improve deep representation learning through unlabeled data and has been widely used in the fields of computer vision and natural language processing [1,3,6]. There have been some studies on network schema and meta-paths of heterogeneous graphs for SSL data augmentation [25]. However, there is few studies on the heterogeneous graphs of the long-tail distribution of the data, which do not take into account the feature distribution of recommended items. This makes it easy to recommend head items.

In this paper, we propose a novel **H**eterogeneous **G**raph Neural Network-based method for **D**eveloper **R**ecommendation (called HGDR), in which text information embedding and SSL are incorporated. Specifically, we unify developer-issue interaction, developer-source code interaction and issue-source code relations into a heterogeneous graph. Meanwhile, we embed text descriptions of developer commits and issues to graph nodes, which can well alleviate the sparsity of developer-issue interaction. To compensate for the recommendation bias caused by the long-tail, we propose a proficiency weight suppression label link supplementation to complement the tail developers who may have the ability to solve issues. In order to fully utilize rich structural information, we learn structural feature representations of developers and issues with the joint learning of HGNN and SSL. A novel data augmentation method based on SSL is proposed to better learn the latent relations of developer features. The main contributions of this work are as follows:

- To the best of my knowledge, HGDR is the first proposal to use heterogeneous graph neural networks to model the complex relationship between source code files, developers, and issues to recommend developers.
- We present a novel Heterogeneous Graph Neural Network-based model, HGDR. To address the sparsity and long tail of the developer-issue interactions, text information embedding and the proficiency weight suppression link supplementation are proposed, respectively. HGDR can also be used in many similar recommendation scenarios where label sparsity and long tails. In addition, we construct three real-world datasets for developer recommendation.
- Extensive experiments on three real-world datasets show that our HGDR outperforms existing state-of-the-art. The results demonstrate the effectiveness of HGDR in addressing the long-tail distributions and sparse labels of developer recommendation.

2 Preliminary

In this section, we define some basic concepts related to developer recommendation.

Definition 1 Developer Recommendation. In a software development platform, for an issue $i_i \in I$ recommends a set of suitable developers $\{d_1, d_2, ..., d_m\} \subseteq D$, where I represents the set of all issues, D represents the

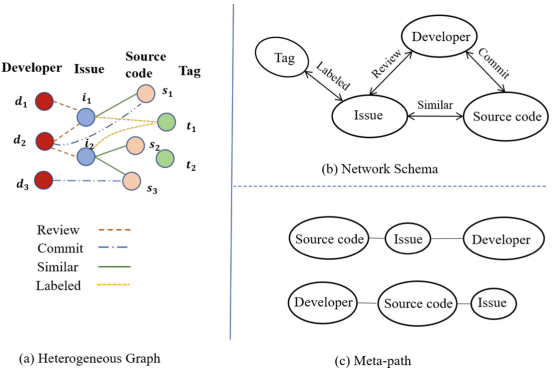


Fig. 2. Heterogeneous information network structure and network model.

set of all developers, k represents the number of recommended developers. We construct HIN to model our developer recommendation task. A HIN is defined as $G = (V, E)$, where V and E denote the set of nodes and edges that have more than one node type or edge link type. For example, Fig. 2 (a) illustrates an example of a HIN including four node types: developer, source code file, issue, and tag, and four relationships between these nodes: review, commit, similar, and label. Here V includes the set of developer nodes D, the set of issue nodes I, the set of source code files nodes S and the set of issue label nodes T. Edge E is composed of relational edges between nodes, e.g., the developer-issue interaction edge (d, i) is $x_{di} = 1$.

Definition 2 Network Schema and Meta-path. A network schema is defined as $S_G = (O, R)$, which is the schema of a heterogeneous information network containing a directed graph of relationships R between all object types O. For example, Fig. 2 (b) shows the network schema, where developers are able to interact with issues by comments, with source code files through commits, and issues are related to the associated source code files. The meta-path P is defined as the path $O_1O_2...O_{(l+1)}$, describing the complex relation $R = R_1 \circ R_2... \circ R_l$ between objects O_1 and $O_{(l+1)}$, and $\circ$ denotes the combinatorial operator of the relation. For example, Fig. 2 (c) shows two meta-paths in a network. SourceCode-Issue-Developer describes an issue that has been solved by the developer through comment replies, and submits some relevant source code files. Developer-SourceCode-Issue describes an issue that has been solved with some source code files. These source code files are also committed by some developers. Since meta-paths are combinations of multiple relations and contain complex semantics, they are considered as higher-order structures. The notations in this paper are shown in Table 1.

Table 1. Notations utilized in the paper.

Notations	Descriptions
I	$I = \{i_1, i_2, ..., i_n\}$ denotes a set of issues
D	$D = \{d_1, d_2, ..., d_m\}$ denotes a set of developers
S	$S = \{s_1, s_2, ..., s_p\}$ denotes a set of source code files
T	$T = \{t_1, t_2, ..., t_q\}$ denotes a set of tags
I-D-I	denotes issues solved by the same developer
D-I-D	denotes developers who have solved the same issue
I-T-I	denotes the same label issues
S-D-I	denotes developers who have solved this issue have also submitted some source code files
I-S-D	denotes some of the source code files submitted by the developer is related to some issue solving
D-S-I	denotes the issue is related to the source code files that has been modified by some developers' commits
S-I-D	denotes some issues solved by the developers are also related to some source code files

3 Approach

Figure 3 shows our model HGDR framework process for developer recommendation. First, we introduce data collection and heterogeneous graph construction. Second, We use text information embedding and link supplement methods to the heterogeneous graph. Third, joint learning based on meta-path and self-supervised is used to learn the embedding representation. Finally, with the embeddings of issues and developers, we predict the similarity score of developer-issue pair.

3.1 Data Collection and Heterogeneous Graph Construction

We collect the developer history commit records, issue related information and source code repository related information of open source projects from the popular open source platform Github. Then we preprocess the data to remove noisy data such as robot administrator data (some large projects set up robots to automatically submit simple actions), noisy source code file data (not in the set of source code files that have been submitted by developers), empty data, and duplicate data. To construct heterogeneous graph, we extract information about each issue's interaction with developer comments, the developer's commit records, the issue's tagging information, and the source code files associated with the issue solution. In particular, the source code files related to the issue solution are edge-linked using the issue description text after natural language processing preprocessing steps (including removing converting lowercase forms, removing punctuation, removing stop words, and stemming reduction lexical reduction word types). We use the NLTK [18] package to process the sentences, and the preprocessed words are compared with the words in the related source

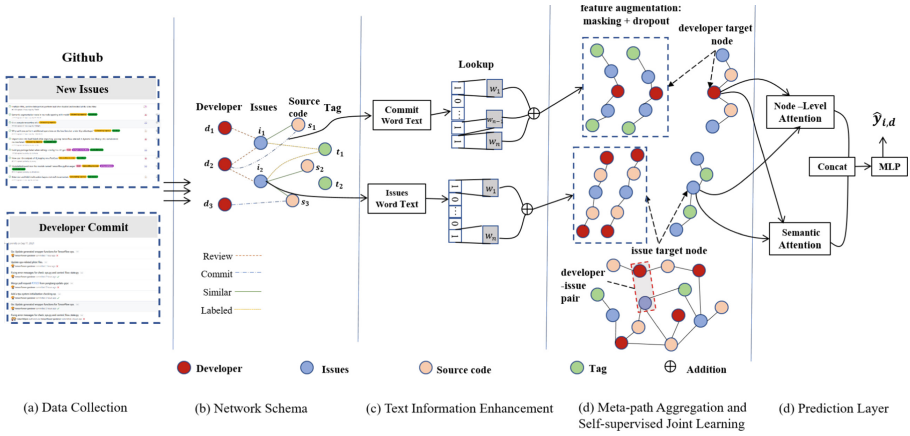


Fig. 3. The overall framework of HGDR.

code files (identifiers in the code and words in the comments) calculate the cosine similarity. The definition is as follows:

$$Similarity = \frac{|IssueRequest \cap SourceCode|}{|SourceCode|}, \tag{1}$$

where $IssueRequest$ represents all words in the preprocessed issue request description, $SourceCode$ represents all words of the preprocessed source code in the source code file. Finally, a list of source code files is ranked by their similarity values to the issue, and we extract the relevant source code file which have the similarity value over a threshold (is set to 0.75 and the manual test issue correlates better with the source code file) to connect to the issue with similar edges. Using the data extracted above, we can construct a heterogeneous graph G. We describe the developer $e_d \in \mathbb{R}^N$, issue $e_i \in \mathbb{R}^N$, tag $e_t \in \mathbb{R}^N$, and source code file $e_s \in \mathbb{R}^N$ with embedding vectors, where the upper right N of $\mathbb{R}$ denotes the size of the vector dimension, and then construct an embedding matrix lookup table that can map the id to the embedding vector as follows,

$$E = [e_{i_1}, ..., e_{i_n}, e_{d_1}, ..., e_{d_m}, e_{t_1}, ..., e_{t_p}, e_{s_1}, ..., e_{s_q}]. \tag{2}$$

3.2 Text Information Enhancement and Link Supplement

In order to address the sparsity of newly arrived issue requests, we introduce text data to enhance the initial node embedding of heterogeneous graphs. First, we construct the corpus with the text description information committed by the developers and the text description information of the issue. It includes the text information of developer commit activity and issue content, respectively. Secondly, we extract the text information using Word2Vec [13] pre-training, and construct the dictionary $W = \{w_1, w_2, ..., w_{n-1}, w_n\}$ to get the word embedding.

Finally, we encode the text descriptive segment of the issue, and the text contains the sum of all word embeddings averaged as the initial embedding vector of the issue node. In particular, since the developer may have more than one history of submitting text descriptions, we also obtain the embedding of the text description segment after summing and averaging each word embedding. And we average these text description segments summations as the initial embedding vector of the developer node. In this way, when a newly arrived issue node has less information about other edges, we can also make good recommendation based on the embedding information constructed from the text descriptions. We apply the method equation as follows,

$$\begin{aligned} e_i &= g(\{e_{w_1}, \ldots, e_{w_{n-1}}, e_{w_n}\}), \\ e_d &= g(\{\{e_{w_1}, \ldots, e_{w_{n-1}}, e_{w_n}\}, \ldots, \{e_{w_1}, , e_{w_{n-1}}, e_{w_n}\}\}), \end{aligned} \tag{3}$$

where e_{w_i} is the embedding of word w_i, e_i is the embedding of issue, e_d is the embedding of developer and the $g(\cdot)$ means the operation function applied to the words. In our experiments, we adopt the average function.

Since the developers' labels are in the long-tail distribution, this leads to a tendency to recommend experienced developers and a bias against developers with low proficiency. We propose a proficiency weight suppression link supplementation method for the proficiency bias of labels to mitigate the recommendation high proficiency tendency bias, which enables the recommendation to focus on finding the appropriate developer to solve the issue. The ranking indicator function DCG [9] is referenced to introduce an adaptive weighting mechanism that adaptively assigns weights based on the number of issues solved by the developer, while ensuring that the supplemented labels have solved similar issues before. Here the cosine similarity using the embedding information constructed from the text descriptions obtained above. We apply the method equation as follows,

$$\begin{aligned} S_{i,d} &= w_d \cdot Rel_{d,i}, \\ Rel_{d,i} &= \sum_{i' \in I_d} Sim(i, i'), \\ w_d &= 1 + weight/(1 + \log p_d), \end{aligned} \tag{4}$$

where $Rel_{d,i}$ denotes the similarity of all issues solved by the developer's history with this issue, $Sim(,)$ denotes the cosine score similar to Eq. 1 and $S_{i,d}$ denotes the similarity score between the developer and this issue. The w_d denotes the weight assigned adaptively by the developer, where p_d denotes the number of issues solved by the developer. The *weight* is used as hyperparameters to control the degree of suppression. In order to weight the similarity between the developer and the issue in the supplemented link while appropriately reducing the proficiency bias, we present experiments to manually adjust the *weight* in Sect. 4.6.

3.3 Meta-path and Self-supervised Joint Learning

Supervised learning based on meta-paths as the main task of joint learning. Meta-path aggregation is the sequential semantics encoded by meta-paths that reveal different aspects of connected objects. For node information propagation, the information of all neighbors is step-by-step aggregated, which can save memory and computation time compared to classical graph neural networks (GNN), such as graph convolutional networks (GCN) [11] and graph attention networks (GAT) [21]. Since the goal of learning is developer and issue representation, it is intuitive to choose to end the meta-path with a developer or issue node, which ensures that the information aggregation on the meta-path is the end of the node we care about.

We extracted rich meta-path higher-order features and design eight meta-paths I-T-I, S-D-I, I-D-I, D-S-I, I-S-D, T-I-D, S-I-D, and D-I-D. The notations meanings are shown in Table 1. For example, D-S-I indicates that the source code files related to the issue and the solution has been submitted for modification by some developers. The impact of different meta-paths on our recommendation performance is studied in Sect. 5. The meta-path information aggregation is shown as follows,

$$X_{p,1} = \sigma(GNN_{p,1}(E, A_{p,1})), X_{p,2} = \sigma(GNN_{p,2}(X_{p,1}, A_{p,2})), \tag{5}$$

$$X_{p,k} = \sigma(GNN_{p,k}\ (X_{p,k-1}, A_{p,k})), \tag{6}$$

where E denotes the initial node embedding, $\sigma()$ is the activation function. $X_{p,k}$ is the node representation on meta-path p after the kth propagation. $A_{p,k}$ denotes the adjacency matrix of meta-path p at the step k. The $GNN_{p,k}$ denotes GNN layer of the meta-path p at the step k and propagation method is GAT, which aggregates neighboring nodes considering the importance of attention at the node level. By stacking multiple GNN layers, HGDR can be used not only to explicitly explore the multi-hop connectivity in a meta-path, but also to efficiently capture collaboration signals. SSL as an auxiliary task of joint learning in Fig. 4. We consider a developers d_i, given the same developer node i input, after augmented by transformation function $f^{'}$ and $f^{''}$, i.e., masking nodes or edges with two perspectives. Finally, we use metapath-guided heterogeneous graph neural network g' and g'' encoding to get two perspectives of representation $z_i^{'}$ and $z_i^{''}$, that is

$$z_i^{'} \leftarrow g^{'}(f^{'}(e_i)),\ z_i^{''} \leftarrow g''(f''(e_i)). \tag{7}$$

Different from existing heterogeneous graphs with random masking mechanisms for edges or nodes. For example, a developer may loses the source code files submitted about the front-end sub-interface of the configuration software, but retains the source code file information of the configuration home page, which is not conducive to comparative learning to learn meaningful representation information. We could randomly divide the node neighbors into two mutually exclusive sets of neighbors for data augmentation. We call this method Random Dropout (RMD), and will use it as one of our baselines. We now introduce Similarity Meta-path Neighbor Masking (SMP) where we further explore

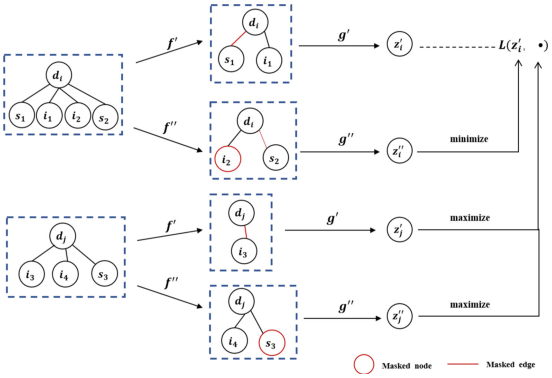


Fig. 4. Self-supervised learning framework illustration.

the feature correlations when creating masking patterns. Specifically, we can learn more meaningful comparative features by dividing the meta-path into two subsets of neighbors with large differences in similarity based on their embedded Euclidean Distance [2].

Similarity Meta-path Neighbor Masking. We randomly select the seed meta-paths Mp_{seed} for each batch developer, and use the pre-computed similarity. Here we split it into two equal number of mutually exclusive neighbor sets $N_d = \{Mp_{seed}, Mp_1, ...Mp_k\}$ by first order neighbor similarity. k is half the number of available neighbor nodes. We change the seed meta-paths for each batch so that the SSL task can learn various types of mask patterns.

After different data augmentation f' and f'', ensure that the model recognizes the representation of the same input node i. In other words, minimize the learning loss represented by the same nodes $z_i^{'}$ and $z_i{''}$ and maximize the learning loss of different nodes $z_i^{'}$ and $z_j^{''}$ after data enhancement. We treat $(z_i^{'}, z_i^{''})$ as the positive pair and $(z_i^{'}, z_j^{''})$ as the negative pair for $i \neq j$. We define the SSL loss for a batch of developer $\{d_i\}$ as:

$$L_{ssl}\{d_i\} = -\frac{1}{n}\sum_{i \in D}^{n} \log \frac{exp(s(z_i^{'}, z_i^{''})/\tau)}{\sum_{j \in D}^{n} exp(s(z_i^{'}, z_j^{''})/\tau)}, \tag{8}$$

where $s(\cdot)$ measures the similarity of two embeddings, we still using the cosine similarity function. τ is a softmax hyperparameter. The above loss function $L_{ssl}\{d_i\}$ learns a robust embedding space, similar developers are close to each other after data augmentation, and random developers are pushed farther away. SSL can be performed for unlabeled scenarios.

3.4 Prediction Layer

After joint learning of the meta-path-aware subgraph, the semantic information revealed by all meta-paths is fused. Suppose that for a developer node d, its set of

node representations $X_d = \{X^d_{p_1}, X^d_{p_2}, ..., X^d_{p_\lambda}\}$ is aggregated from λ meta-paths. The importance of meta-path semantics and node representation varies, and the contribution of each meta-path should be adjusted accordingly. Therefore, we use attention mechanism to learn the importance of each meta-path. We measure each meta-path weight as follows:

$$W^d_{mp} = \alpha^T \cdot X_d, \tag{9}$$

where W_{mp} measures the vector of meta-path importance and α is the matrix of learnable parameters. We then normalize the meta-path importance scores using the softmax function and obtain the attention factor for each meta-path as follows:

$$\beta^d_{mp_i} = \frac{exp(W^d_{mp_i})}{\sum_{j=1}^{\lambda} exp(W^d_{mp_j})}, \tag{10}$$

where $\beta^d_{mp_i}$ denotes the normalized attention factor of the meta-path mp_i on node d. Using the learned attention factors, we can fuse all meta-path aggregation node representations into the final meta-path-aware node representation z_d as follows:

$$z_d = \sum_{i=1}^{\lambda} \beta^d_{mp_i} X^d_{mp_i}, \tag{11}$$

z_i is obtained in the same way as z_d. We use a different pooling strategy which connects the embedding representation z_i of the target issue node with the embedding representation z_d aggregated representation of the developer node. i.e.,

$$z_g = concat(z_i, z_d). \tag{12}$$

Then, we use the 2-layer Multilayer Perceptron (MLP) to compute the matching scores of the issue-developer pairs. Let us denote by $\hat{y}_{i,d}$ the prediction function for the interaction score of issue i and developer d can be expressed as follows.

$$\hat{y}_{i,d} = w_2^T \sigma(w_1^T z_g + b_1) + b_2, \tag{13}$$

where w_1, w_2, b_1 and b_2 are trainable parameters of the MLP and σ is the nonlinear activation function.

3.5 Multi-task Training

In order to make the SSL representation contribute to improved learning of the supervised task, we utilize a multi-task training strategy in which the primary supervised task and the secondary SSL task are jointly optimized. Precisely, let $\{(i, d)\}$ be a batch of issue-developer pairs sampled from the training set, and let d be sampled from a batch of developer distribution D. The loss of joint learning is:

$$L = L_{main}(\{(i, d)\}) + \lambda_1 L_{ssl}(\{d\}) + \lambda_2 ||\theta||_2^2, \tag{14}$$

where L_{main} is the loss function of the main task HGNN recommendation. θ is the set of model parameters, λ_1 and λ_2 are the hyperparameters controlling the regularization strength of SSL and L_2, respectively.

Our main task uses pairwise Bayesian personalized ranking (BPR) loss [15], which is common for recommender systems. The BPR loss can be expressed as follows,

$$L_{main}(\{(i,d)\}) = \sum_{(i,d_+,d_-)\in B} -\log(\hat{y}(i,d_+) - \hat{y}(i,d_-)), \tag{15}$$

where $B = \{(i,d_+,i_-)|y_{(i,d_+)} \in Y_+, y_{(i,d_-)} \in Y_-\}$ is the training set, Y_+ is the observed issue-developer interaction (positive sample), while Y_- is the unobserved issue-developer interaction (negative sample).

4 Experiment Setup

Table 2. Statistics of three utilized real-world datasets.

Project	Commit	Total issue	Open issue	Developer	Source file	Time period
Tensorflow	123264	33683	2137	3046	24426	2015-11-12- 2021-01-16
Flutter	27032	63251	10649	957	6053	2015-04-30- 2022-01-06
Vscode	91053	125690	6427	1559	5593	2015-11-20- 2022-01-15

We provide empirical results to demonstrate the effectiveness of our proposed HGDR in real open source software projects. The experiments aim to answer the following research questions.

- **Q1**: Can our construction of a heterogeneous graph neural network model effectively improve the accuracy of recommendation developers compared to other baseline methods?
- **Q2**: Does the SSL, text information embedding module improve the model performance? Can it effectively address the recommendation bias caused by the long-tail distribution of developers, while alleviating the sparsity problem of labels?
- **Q3**: How does the meta-path relationship of the graph structure affect HGDR?
- **Q4**: How do the weight suppression parameters, SSL parameters data augmentation parameters and loss function parameters affect the model effect?

4.1 Datasets and Metrics

In this experiment, we collect 3 popular open source projects from GitHub based on the GitHub REST API[1], such as tensorflow, flutter, and vscode. The features

[1] https://docs.github.com/en/rest.

of each project are shown in Table 2. For Tensorflow projects, there are a total of 3046 active developers, 33683 total issues and 2137 open unsolved issues. Our datasets are targeted at open source projects with more issues, while there are a large number of issues in the open state not processed in a timely manner.

Metrics: To evaluate the performance of the recommended developers, we use the popular standard metrics Recall@K and Normalized Discounted Cumulative Gain (NDCG@K) to evaluate the recommended performance for each configuration of the experimental results. Recall@K measures the percentage of test datasets that have been included in the top-K ranking list, and NDCG @K complements recall by assigning higher scores to hits higher on the developer list.

4.2 Baselines and Hyper-parameters

To demonstrate the validity of our HGDR model, we compared it with the following five state-of-the-art methods in different aspects.

MFBPR [15]: This is a matrix decomposition method in the Bayesian personalized ranking pairwise learning framework, which is widely used in developer recommendations as an important baseline for our comparisons.

NeuMF [8]: Using neural networks to enhance matrix decomposition algorithms with nonlinearities, this is used to compare the advantages of our approach with the same addition of nonlinear neural networks.

NGCF [23]: This is a model of bipartite graphs, which uses graph neural networks to extract higher-order connectivity of recommendations. Only issues and developers are used here to construct the interaction bipartite graph, which is used to compare the advantages of our approach over the bipartite graph.

LightGCN [7]: This is a light bipartite graph state-of-the-art graph convolution model based on NGCF, that propagates linearly over the user-item interaction graph to learn user and item embeddings.

KGAT [22]: This is the advanced knowledge-based model for merging higher-order information by learning entity attention performing attentional embedding propagation over the knowledge graph, which compares our approach to knowledge graph approaches with the same kinds of relationships and nodes.

We train and test divide using Leave-One-Out Cross Validation [10], and set the number of negative sampling candidates to {10,20,50,99}. We use Adam optimizer and mini-batch of size 1024 or 2048 with embedding size 64. Learning rate is searched at {1e−5, 1e−4, 1e−3,1e−2}, L2 regularization term weight

decay $\lambda 2$ at {1e−7, 1e−6, 1e−5, 1e−4, 1e−3, 1e−2}. For MFBPR, NeuMF, NGCF, LightGCN and KGAT, we tested and reported the best performance for them. For other parameters, we follow the settings in their original papers. For the text embedding parameters, the parameter vector size is 64, the window is set to 8, and the number of training rounds is 8. For contrast learning, softmax temperature is set to {0.01,0.05,0.1,0.5,1}, λ = {0.1, 0.3, 1.0, 3.0}, dropout rate = {0.1, 0.2, ... , 0.9}. Finally, the best results are reported.

4.3 Effectiveness of HGDR Compared to Other Baseline Methods (to Q1)

HGDR is designed to automatically recommend the appropriate developers to solve issues, which helps to reduce the latency of issues and improve collaboration efficiency. Therefore, we need to know how it performs in terms of developer recommendation compared to existing developer recommendation methods. To answer this research question, we used our datasets in existing developer recommendation baselines as well as some classical recommendation baseline methods for the comparison. We try to use the method parameters as set in their original paper, and some common parameters are set the same as our method to ensure the fairness of the comparison. Table 3 shows the performance of HGDR against the baseline performance of the other five recommended methods.

We can observe that our model HGDR significantly outperforms all baselines in terms of Recall and NDCG metrics. Specifically, HGDR outperforms the

Table 3. Performance comparisons on three real-world datasets with six baselines.

Subject	Top k of Recall and NDCG	MFBPR	NeuMF	NGCF	LightGCN	KGAT	HGDR	%Impro
Tensorflow	5	0.5764	0.5731	0.5471	0.4939	0.6763	**0.8931**	32.06%
	10	0.6924	0.6844	0.6727	0.6256	0.8263	**0.9501**	14.98%
	20	0.7943	0.7805	0.781	0.7446	0.9105	**0.9879**	10.94%
	5	0.449	0.450	0.4196	0.3705	0.5030	**0.7257**	44.27%
	10	0.4866	0.4855	0.4603	0.4134	0.5516	**0.7504**	36.04%
	20	0.5125	0.5098	0.4878	0.4433	0.5780	**0.7581**	31.16%
Flutter	5	0.8288	0.8276	0.7194	0.8148	0.8843	**0.9495**	7.37%
	10	0.8729	0.8608	0.8066	0.8799	0.9143	**0.9799**	7.17%
	20	0.8438	0.858	0.8784	0.9248	0.9378	**0.9943**	6.02%
	5	0.7153	0.7225	0.5913	0.7054	0.7351	**0.8587**	16.81%
	10	0.7365	0.7434	0.6196	0.7266	0.7596	**0.8687**	14.36%
	20	0.7418	0.7503	0.6378	0.738	0.7631	**0.8724**	14.32%
Vscode	5	0.5292	0.5243	0.5087	0.5048	0.5321	**0.6058**	13.85%
	10	0.6587	0.645	0.6405	0.6327	0.7112	**0.8211**	15.45%
	20	0.7806	0.766	0.7674	0.7608	0.8572	**0.9591**	11.89%
	5	0.407	0.4068	0.3861	0.3835	0.4296	**0.4753**	10.64%
	10	0.4486	0.4459	0.4287	0.4248	0.5087	**0.6373**	25.28%
	20	0.4796	0.4765	0.4609	0.4573	0.5821	**0.6814**	17.06%

best baseline by 10.94%–44.27%, 6.02%–16.81%, and 10.64%–25.28% on Tensorflow, Flutter, and Vscode full datasets, respectively. The better performance of KGAT compared to the models (MFBPR, NeuMF, NGCF and LightGCN) using only developer-issue interaction information proves that multiple graph structure relationships are necessary.

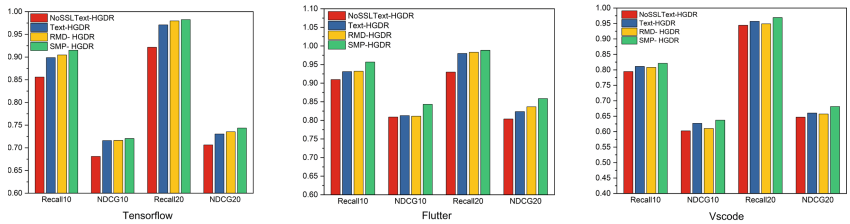
Among all the methods of comparison, HGDR achieves the best performance in most cases for all these datasets. There are three main reasons for this: 1) HGDR models the complex relationships between source code, developer, and issue display encoded for developer and issue representation learning. From the developer perspective, unlike traditional matrix decomposition models (MFBPR, NeuMF) and the bipartite graph methods (NGCF and LightGCN), which simply use sparse developer-issue history interactions as developer representation. HGDR exploits to more precise information source code file code changes and the complex graph relationships between them, allowing more useful information to be extracted. 2) HGDR uses text information embedding supplement and encoded into the developer and issue representation learning, which helps to alleviate the interaction sparsity issue and improve the accuracy of recommendations(as will be demonstrated in Sect. 4.4)). 3) SSL of HGDR data augmentation that enables contrast learning to learn more meaningful contrast features for sparsity scenarios (as will be demonstrated in Sect. 4.4)).

4.4 Ablation Studies (to Q2)

By analyzing the maintenance process of open source projects on Github, we found that many developers have very few historical developer-issue interactions. In particular, the interaction sparsity problem is a real and common phenomenon for some newcomers developers. It usually degrades the performance of recommendations because the limited developer interactions make it difficult for previous methods to generate high-quality representations of developers and issues. Meanwhile, previous methods tend to recommend head developers with high proficiency. Here, we aim to evaluate the efficacy of different modules of HGDR in solving the interaction sparsity problem and improving the accuracy of tail developers.

To answer Q2, we evaluate the impact of SSL, text information embedding on model quality separately, we focus on using similarity meta-path neighbor masking (SMP) and random masking and dropout (RMD) as SSL data augmentation techniques and Word2vec text embedding as text data enhancement techniques.

HGDR without SSL as well as text information embedding module (NoSSLText-HGDR) can be considered as an ablation study to isolate text embedding (Text-HGDR) and SSL to observe the improvement effect separately. Finally, comparing our proposed contrast learning data augmentation method (SMP-HGDR) with the widely used baselines data augmentation method (RMD-HGDR).



(a) (b) (c)

Fig. 5. Ablation study of key designs with different datasets.

As shown in Fig. 5, we observe that for the full dataset, SMP consistently performs best compared to no-SSL regularization techniques. On the Tensorflow dataset, the performance of SMP and the Text module is 6.62% and 5.36% higher respectively, relative to the pre-Ablation module. This helps to answer that RQ2, the proposed SSL framework, and text enhancement do improve the model performance of the recommender. By comparing SMP with RMD, it can be seen that SMP has better performance for SSL regularization.

Table 4. Experiment results trained on the sparse (30% down-sampled) datasets.

Model	30% Tensorflow		30% Flutter		30% Vscode	
	Recall@10	NDCG@10	Recall@10	NDCG@10	Recall@10	NDCG@10
MFBPR	0.4102	0.2548	0.7838	0.469	0.281	0.1659
NoTextSSL-HGDR	0.7810	0.6048	0.9183	0.8270	0.5581	0.2547
Text-HGDR	**0.9244**	**0.7145**	**0.9656**	**0.8374**	0.6802	0.4577
RMD-HGDR	0.7844	0.5855	0.9288	0.8100	0.6132	0.3179
SMP-HGDR	0.8745	0.6515	0.9488	0.8270	**0.6923**	**0.4894**

Table 5. Head and tail developer recommended performance results on the sparse datasets.

Model	30% Tensorflow				30% Flutter			
	Head		Tail		Head		Tail	
	Recall@10	NDCG@10	Recall@10	NDCG@10	Recall@10	NDCG@10	Recall@10	NDCG@10
MFBPR	0.6924	0.4866	0.479	0.3697	0.7838	0.469	0.5461	0.3659
NoTextSSL-HGDR	0.8831	0.5531	0.6135	0.3618	0.9234	0.8343	0.7107	0.5260
RMD-HGDR	0.9040	0.7032	0.6844	0.4855	0.9234	0.8343	0.7107	0.5260
HGDR	**0.9480**	**0.7491**	**0.8840**	**0.6787**	**0.9844**	**0.8999**	**0.8842**	**0.6468**

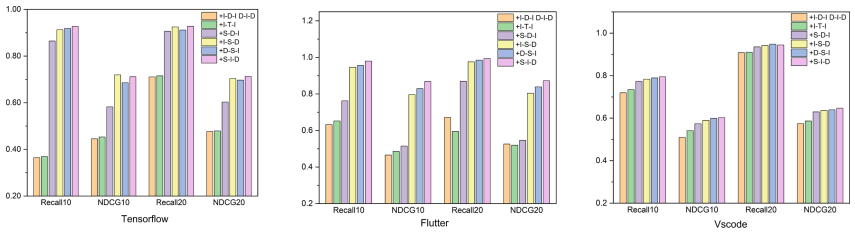
Label Sparsity Analysis. We study the effectiveness of HGDR in presence of sparse data to address Q2. We uniformly down-sampled 30% (too low can seriously affect the recommended performance) of training data and evaluate on the same (full) test dataset. The experimental results are shown in Table 4. Increasing data sparsity, HGDR provides a greater improvement. In addition,

for some new incoming issues with very few labels, the text embedding information is better for such issue recommendations. For example, in Tensorflow, Text-HGDR improves Recall@10 125.35% on 30% sparse dataset compared to MFBPR method.

Head-Tail Analysis. To understand the gain from each module for mitigating long-tail developers, we further decompose the overall performance by looking at different developer slices by developer proficiency. For the Tensorflow test dataset, the header dataset contains examples where groundtruth developers are in the top 10% of the most frequent developers and the rest of the test developers are considered tails. Our hypothesis is that SSL usually helps to improve the performance of slices (e.g. tail developers) without much supervision. The results evaluated on the tail and head test sets are reported in Table 5. We observe that the proposed SSL approach improves the performance recommended by both head and tail developers, with greater gains for tail developers. For example, in Tensorflow, SMP improves Recall@10 by more than 51.5% on tail items and by 8.57% on heads.

4.5 Effects of Different Meta-paths (to Q3)

Unlike traditional models [20,30,31], which only focus on a single relationship between the interaction between developers and issues. HGDR extracts a rich set of meta-path higher-order feature relationships, which can effectively improve the recommendation performance. Here, we aim to evaluate the impact of integrating these meta-paths on recommendation results.

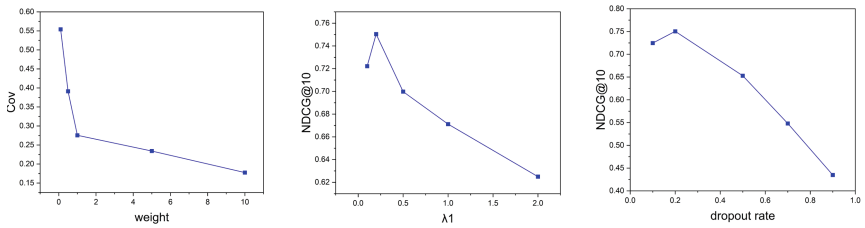


(a) (b) (c)

Fig. 6. Performances of HGDR with different meta-paths.

We choose to add meta-paths accordingly, and comparing the results as shown in Fig. 6. We have the following observations. First, the meta-paths I-D-I, D-I-D that only exploit the issue-developer synergy relationship have poor performance. For example, on the Tensorflow dataset, the Recall@10 and NDCG@10 for +I-D-I, D-I-D are 36.44%, 44.55%, respectively. Second, the improvement was not significant after adding the issue tagging relationship meta-path I-T-I. Until adding source code, developers, issue higher-order meta-path S-D-I relations, the recommendation performance has a large improvement. Here we consider only

the collaborative relationship of developer-issue interaction, because the issue can only be solved by a small number of developers and then in a closed state, resulting in this collaborative relationship to disseminate limited information. We add source code information as a bridge between developers and the issue higher-order connection, so that the information disseminated between developers and the issue is richer and more accurate. Finally, adding I-S-D, D-S-I, and S-I-D meta-paths, the recommendation performance was also improved, proving that the higher-order graph structure relationship is helpful for recommendation effect improvement.



(a) (b) (c)

Fig. 7. Effect of different parameters on the model in the Tensorflow dataset.

4.6 Effects of Parameters (to Q4)

In order to analyze the impact of different weight suppression parameters on the model, we observe the change in the coverage of recommended developers to all developers. Coverage measures the ratio of distinct developers in all lists to all developers in an experiment instance. Cov value is calculated for each experiment instance as follows:

$$Cov = \frac{\sum_{u=1}^{m} distinct(pl(d))}{|L|}, \tag{16}$$

where $distinct(L)$ is the number of distinct developers in list L, $pl(d)$ is developers prediction list for each issue. We can see that the experimental results in Fig. 7 (a) show that when the weight is smaller, the coverage of recommended developers is higher, the more we suppress the developers with more interactions, and the recommended results are more diverse.

Figure 7 (b) summarizes the regularization strength evaluated on the Tensorflow dataset. We observe that as the regularization parameter increases, the model performance increases first and gets worse after a certain threshold. This is expected since the large SSL weights lead to a multitasking loss L dominating the loss function in Eq. (13). Figure 7 (c) shows the increase in model performance for different dropout rates as parameters, peaking at 0.2. It then deteriorates when we further increase the dropout rate. This observation is consistent with our expectation that when the dropout rate is too large, the input information becomes too small to learn a meaningful representation via SSL.

5 Related Work

5.1 Developer Recommendation

To find the appropriate developer, [20,31] focus on developers-issues interactions and recommend developers based on matrix factorization (MF) [12,30] or collaborative filtering (CF) [20,31]. CF-based approaches typically suffer from severe sparsity and cold-start problems when the explicit interactions between developers and issues are sparse. For example, by analyzing the GitHub dataset, the data sparsity of the developer-issue explicit interaction matrix is as low as 0.1%, which greatly limits the effectiveness of recommendations. To address these limitations, previous work incorporates various side information into MF or CF. Recently, Sun et al. [20] proposed EDR_SI to recommend developers by exploring commit repositories using collaborative topic modeling (CTM) techniques. Xie et al. [30] proposed a SoftRec approach that incorporates developer collaboration relations and inter-task collaboration relations multi-relations into matrix decomposition to recommend developers. These methods alleviate sparsity and improve the performance of recommendation to some extent. Most of these methods are based on MF or CF traditional machine learning methods, while ignoring the fact that the relational information that may help improve recommendation performance, and cannot learn developer and issue representation due to sparsity and long-tail well.

5.2 Heterogeneous Graph Neural Networks

In recent years, graph neural networks (GNNs) have attracted considerable attention. Most of them have been proposed for homogeneous graphs [7,23] and a detailed survey can be found in [26]. Recently, some researchers have focused on heterogeneous graphs. For example, HAN [24] uses hierarchical attention to describe node-level and semantic-level structures. PEAGNN [5] specifies meta-path node embeddings by means of features in a contrastive manner. The above methods greatly enrich the graph structure to represent the learning information, but there is no way to directly apply the methods for developer recommendation.

6 Conclusion

In this paper, we propose HGDR, a heterogeneous graph neural network-based method for developer recommendation. The text information embedding and the proficiency weight suppression link supplement are proposed to address the sparsity and long-tail of the developer-issue interactions. To the best of my knowledge, this is the first work to propose the use of HGNN to model source codes files, developers, and issues complexity relationships to recommend developers. Meanwhile, a generality SMP is used for SSL joint learning. Moreover, Extensive comparison experiments are conducted on three real-world datasets, which show that the performance outperforms state-of-the-art by a maximum of 44.27% in terms of NDCG. The experimental results also prove the efficacy of HGDR in sparse and long-tail scenario.

In the future, we plan to further explore the usefulness of HGDR for practical production. We hope to provide usable tools for some open source communities or commercial software companies, and further investigate some features related to developer recommendation.

Acknowledgement. This work is supported by the National Natural Science Key Foundation of China grant No. 61832014 and No. 62032016, the National Natural Science Foundation of China grant No. 62102281, the Natural Science Foundation of Tianjin City grant No. 19JCQNJC00200, and the Foundation of Jiangxi Educational Committee (GJJ210338).

References

1. Chen, T., Kornblith, S., Norouzi, M., Hinton, G.: A simple framework for contrastive learning of visual representations. In: International Conference on Machine Learning, pp. 1597–1607. PMLR (2020)
2. Danielsson, P.E.: Euclidean distance mapping. Comput. Graphics Image Process. **14**(3), 227–248 (1980)
3. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
4. Gousios, G., Zaidman, A., Storey, M.A., Van Deursen, A.: Work practices and challenges in pull-based development: the integrator's perspective. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 358–368. IEEE (2015)
5. Han, Z., et al.: Metapath-and entity-aware graph neural network for recommendation. arXiv e-prints, arXiv-2010 (2020)
6. Hassani, K., Khasahmadi, A.H.: Contrastive multi-view representation learning on graphs. In: International Conference on Machine Learning, pp. 4116–4126. PMLR (2020)
7. He, X., Deng, K., Wang, X., Li, Y., Zhang, Y., Wang, M.: LightGCN: simplifying and powering graph convolution network for recommendation. In: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 639–648 (2020)
8. He, X., Liao, L., Zhang, H., Nie, L., Hu, X., Chua, T.S.: Neural collaborative filtering. In: Proceedings of the 26th International Conference on World Wide Web, pp. 173–182 (2017)
9. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inf. Syst. (TOIS) **20**(4), 422–446 (2002)
10. Kearns, M., Ron, D.: Algorithmic stability and sanity-check bounds for leave-one-out cross-validation. Neural Comput. **11**(6), 1427–1453 (1999)
11. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
12. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009)
13. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
14. Rajlich, V.: Software evolution and maintenance. In: Future of Software Engineering Proceedings, pp. 133–144 (2014)

15. Rendle, S., Freudenthaler, C., Gantner, Z., Schmidt-Thieme, L.: BPR: Bayesian personalized ranking from implicit feedback. arXiv preprint arXiv:1205.2618 (2012)
16. Servant, F., Jones, J.A.: WhoseFault: automatic developer-to-fault assignment through fault localization. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 36–46. IEEE (2012)
17. Shi, C., Li, Y., Zhang, J., Sun, Y., Philip, S.Y.: A survey of heterogeneous information network analysis. IEEE Trans. Knowl. Data Eng. **29**(1), 17–37 (2016)
18. Steven, B.: NLTK: the natural language toolkit in proceedings of the ACL 2004 on interactive poster and demonstration sessions. In: Association for Computational Linguistics, p. 31 (2004)
19. Sun, X., Yang, H., Leung, H., Li, B., Li, H.J., Liao, L.: Effectiveness of exploring historical commits for developer recommendation: an empirical study. Front. Comp. Sci. **12**(3), 528–544 (2018). https://doi.org/10.1007/s11704-016-6023-3
20. Sun, X., Yang, H., Xia, X., Li, B.: Enhancing developer recommendation with supplementary information via mining historical commits. J. Syst. Softw. **134**, 355–368 (2017)
21. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
22. 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, pp. 950–958 (2019)
23. Wang, X., He, X., Wang, M., Feng, F., Chua, T.S.: Neural graph collaborative filtering. In: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 165–174 (2019)
24. Wang, X., et al.: Heterogeneous graph attention network. In: The World Wide Web Conference, pp. 2022–2032 (2019)
25. Wang, X., Liu, N., Han, H., Shi, C.: Self-supervised heterogeneous graph neural network with co-contrastive learning. arXiv preprint arXiv:2105.09111 (2021)
26. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2020)
27. 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: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 261–270. IEEE (2015)
28. Xia, X., Lo, D., Wang, X., Zhou, B.: Dual analysis for recommending developers to resolve bugs. J. Softw. Evol. Process **27**(3), 195–220 (2015)
29. Xia, Z., Sun, H., Jiang, J., Wang, X., Liu, X.: A hybrid approach to code reviewer recommendation with collaborative filtering. In: 2017 6th International Workshop on Software Mining (SoftwareMining), pp. 24–31 (2017). https://doi.org/10.1109/SOFTWAREMINING.2017.8100850
30. Xie, X., Wang, B., Yang, X.: SoftRec: multi-relationship fused software developer recommendation. Appl. Sci. **10**(12), 4333 (2020)
31. Xin, X., He, X., Zhang, Y., Zhang, Y., Jose, J.: Relational collaborative filtering: modeling multiple item relations for recommendation. In: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 125–134 (2019)
32. Ye, L., Sun, H., Wang, X., Wang, J.: Personalized teammate recommendation for crowdsourced software developers. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 808–813 (2018)