The Journal of Systems & Software 200 (2023) 111667

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Graph collaborative filtering-based bug triaging☆

Jie Dai, Qingshan Li *, Hui Xue, Zhao Luo, Yinglin Wang, Siyuan Zhan

Xidian University, School of Computer Science and Technology, Xi'an, Shaanxi, China

ARTICLE INFO

Article history:
Received 29 September 2022
Received in revised form 26 December 2022
Accepted 26 February 2023
Available online 1 March 2023

Keywords:
Software reliability engineering
Bug triaging
Deep graph learning
Graph collaborative filtering

ABSTRACT

Issue tracking systems are widely used for collecting bug reports. A target of intelligent software engineering is to automate assigning bugs to appropriate developers. Recently, the momentum of artificial intelligence has brought many successful studies that triage bugs by classifying their reports with NLP-based methods. Some studies also try to introduce context information to represent developers. Nevertheless, they take a fundamental assumption that developers and bugs, closely related entities in real-world scenarios, should be modeled independently.

To capture the bug-developer correlations in bug triaging activities, we propose a Graph Collaborative filtering-based Bug Triaging framework: (1) bug-developer correlations are modeled as a bipartite graph; (2) natural language processing-based pre-training is implemented on bug reports to initialize bug nodes; (3) spatial–temporal graph convolution strategy is designed to learn the representation of developer nodes; (4) information retrieval-based classifier is proposed to match bugs and developers. Extensive experiments across mainstream datasets show the competence of our GCBT. Moreover, We believe that GCBT could generally benefit the modeling of correlations in other software engineering scenarios.

1. Introduction

With the rapid development of modern software engineering, especially the widespread adoption of open-source software, software's increasing complexity and diversity have inevitably generated many bugs in issue tracking systems. These bug records, as shown in Fig. 1, constitute an essential artifact that connects developers and software. To improve the efficiency of software reliability engineering, researchers have been trying for a long time to automate the cumbersome process of manually triaging bugs to programmers, with the help of artificial intelligence toolkits.

Most existing studies treat bug triaging as a bug report classification problem. Thanks to the development of natural language processing (NLP), bug report texts are exploited to obtain semantic representations (Anvik and Murphy, 2011; Jonsson et al., 2016; Shokripour et al., 2013; Tamrawi et al., 2011; Wang et al., 2014; Xuan et al., 2014, 2012), which are then classified with developers as labels. Besides, a few works are trying to represent developers with context information, such as their activities in GitHub or StackOverflow (Badashian et al., 2015). Successful as they are, bugs and developers are studied under the assumption that they are independent of each other, which is far away from reality: the correlations of bug-developer should get deserved attention.

Correlations between two types of entities generally exist in many fields, such as the goods-consumer correlations in e-commerce. Therefore, we can get some insight from collaborative filtering (CF), a recommendation method widely applied in e-commerce (Zhu et al., 2014), streaming media (Davidson et al., 2010), and many other scenarios. By modeling user–item interactions, CF could capture implicit correlations and thus predict a user's future behaviors (Goldberg et al., 1992). With the blooming of graph learning techniques, Graph Collaborative Filtering (GCF) is proposed to capture high-order correlations in the interaction graph (He et al., 2020), leading to more comprehensive representations of user and item nodes, and more precise recommendation results. Relevant studies have theoretically and empirically proved that exploiting correlations could notably improve item–user matching. It is reasonable to assume that such convenience should also be enjoyed by bug-developer matching.

In recent years, attention has been gradually paid to capturing bug-developer correlations in bug triaging scenarios. Early studies try to borrow item-based CF (Karypis, 2001), and capture correlations via calculating the similarity between bug reports and developer records (i.e., their activities in social platforms like StackOverflow) (Badashian et al., 2015; Sajedi Badashian et al., 2016; Hu et al., 2014). Lately, researchers have turned to graph learning to capture bug-developer correlations and learn node representations. For example, SPAN (Mohsin et al., 2022) builds

☆ Editor: Dr. Burak Turhan.

* Corresponding author.
E-mail address: qshli@mail.xidian.edu.cn (Q. Li).

https://doi.org/10.1016/j.jss.2023.111667
0164-1212/

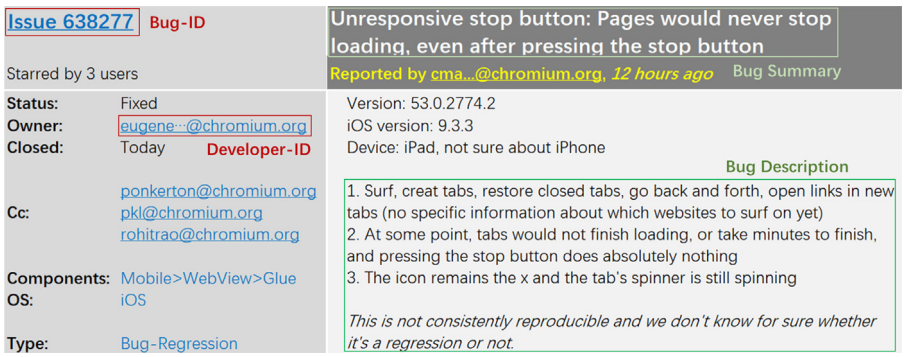

Fig. 1. Example of a bug (ID: 638277) report in the Google Chromium project, consisting of developer emails, a summary, and a detailed description.

a graph of bugs and developers and learns intermediate features with RNN to help triage bugs. Similarly, BTCSR (Yu et al., 2021) models the relationship among developers, bugs, and products as a heterogeneous graph and conducts random walks to capture sequential patterns among edges. These approaches simplify the non-Euclidean structure of graphs as a Euclidean sequential pattern, which makes modeling and calculation easier. However, it causes severe information loss on graph data (Hamilton, 2020), which limits the quality of representation learning and the performance of bug triaging.

To design a graph learning strategy for bug triaging, we propose a Graph Collaborative filtering-based Bug Triaging framework, GCBT: (1) bug-developer correlations are modeled as a bipartite graph; (2) natural language processing-based pre-training is implemented on bug reports to initialize bug nodes; (3) spatial–temporal graph convolution strategy is designed to represent the programming ability of developers; (4) an information retrieval-based classifier is applied to match bugs and developers.

The contribution of our work is three-folded:

- A novel framework GCBT is proposed to capture bug-developer correlations. Unlike most studies that model bug and developer separately, we emphasize the modeling of their correlations and construct a bug-developer bipartite graph.
- A graph learning strategy is designed to capture spatial–temporal features. Considering the unique characteristics of bug triaging tasks, we integrate spatial convolution with temporal aggregation to represent the programming ability of developers.
- Extensive experiments are conducted to show the outstanding performance of GCBT. On three benchmark datasets, GCBT generally outperforms other baselines. We further analyze the effects of the inner component and key hyper-parameters of GCBT.

2. Motivation

We motivate our work by revisiting two essential challenges in automated bug triaging:

1. **How should we construct a framework for bug triaging?** Many recent studies consider bug triaging as a classification problem for bug reports (Li and Zhong, 2021), based on which they utilize various techniques ranging from classic statistical learning (Murphy and Cubranic, 2004) to modern deep learning (Mani et al., 2019). However, such a paradigm limits the performance of developer ability representation and the whole bug triaging prediction.
2. **What is a good learning strategy for correlation modeling?** There are a few researchers who have come to realize the importance of modeling correlations. To take advantage of the bug-developer correlations, many existing studies conduct sequential embedding to represent developers (Hu et al., 2014). Unfortunately, they usually cause information loss and thus damage the performance of bug triaging models.

2.1. Bug triaging framework challenge

It is prevailing in recent studies to classify bug reports with available developers as labels. Most of their efforts are devoted to modeling the bug reports and learning implicit semantic embeddings from unstructured and noisy texts. Developers usually appear as classifying labels (i.e., one-hot vectors), which are far from adequate to represent the programming ability of developers.

We assume that there are three developers (Jack, Tim, and Sam) in a toy dataset of bug fixing. Jack and Tim are responsible for a C++ back-end component, and Jack has fixed more bugs. Sam is in charge of a Java back-end component. There are noticeable differences and similarities among these developers, which are hard to be revealed when Jack, Tim, and Sam are merely represented as "001, 010, 100". This challenge becomes more severe when their programming ability changes over time. For instance, Jack is called on to help Sam maintain the Java back-end, which happens to developers all the time.

Although developers usually do not maintain a detailed resume about their programming ability, their fixed bugs can talk for them. The same challenge is faced by CF recommenders (Goldberg et al., 1992), and a user's preference can be inferred from interacted items when there are no explicit descriptions available. Similarly, we can excavate the bug-developer correlations to represent developers.

2.2. Bug-developer correlation modeling challenge

Correlations of bug-developers, usually modeled as a bipartite graph, are not like texts or images in the Euclidean space. Many recent studies try to transform the non-Euclidean graphs into one-dimension sequences (Grover and Leskovec, 2016) to represent nodes. However, such simplification takes little advantage of the rich connections in the graph, and limits the precision of bug triaging prediction.

In recent years, graph convolution networks have been proposed to preserve the information hidden in the spatial structure of graph data. By excavating neighboring correlations among nodes, it achieves outstanding performance in various tasks (Hamilton, 2020). Nevertheless, those methods can only statically analyze a graph, whereas representing a developer's programming ability calls for dynamic analysis. For example, Jack, who

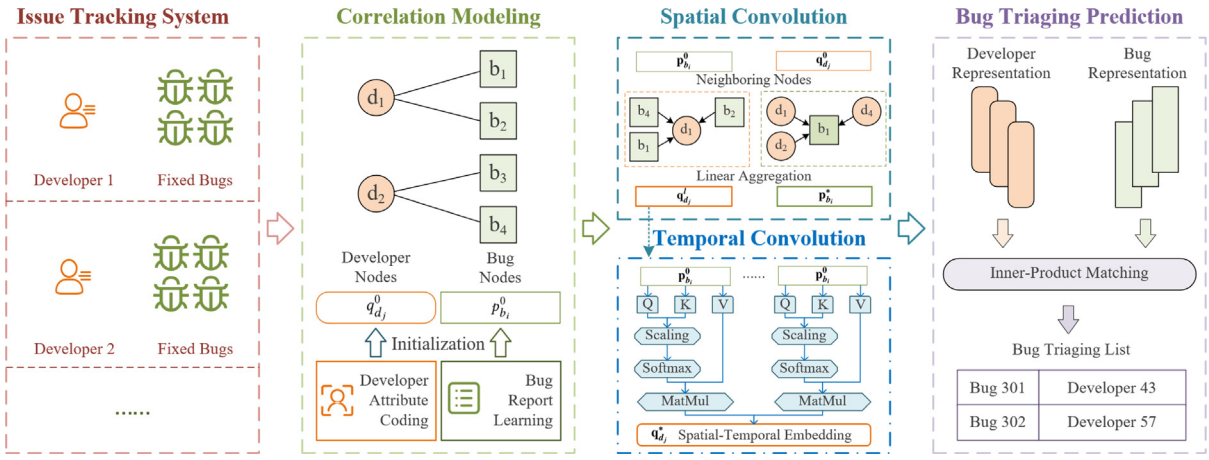

Fig. 2. The whole framework of our GCBT: bug-developer correlations are constructed as a bipartite graph. We conduct spatial–temporal convolutions to learn developer representation and predict the matching results of bug triaging.

used to work on the C++ back-end, now works on a Java component. Here temporal features are underlying his correlations with fixed bugs: the latest bugs are more representative of his programming ability than previous ones. The bug-developer correlations can be fully utilized only with a spatial–temporal graph convolution strategy.

3. Framework

Fig. 2 overviews the whole workflow of our proposed GCBT. We take as input the bug triaging records to build a bipartite graph. Bug nodes are initialized with their report text, whereas developer nodes are initialized with their attributes. We design a spatial–temporal convolution strategy for developer nodes embedding and represent their programming ability based on their connected bugs. An information retrieval (IR)-based classifier is then adopted to predict new bug-developer correlations.

3.1. Triaging graph construction

Given a bug triaging dataset, we define two sets of bugs and developers as follows:

$$Bug: B = \{b_i | i = 1, 2, \ldots, n_B\} \tag{1}$$

$$Developer: D = \{d_j | j = 1, 2, \ldots, n_D\} \tag{2}$$

where n_B and n_D are the total numbers of bugs and developers. Bugs and developers have their contextual attributes: a bug consists of a report title and description, whereas a developer has attributes like e-mail and sex:

$$b_i = \langle title_i, description_i \rangle \tag{3}$$

$$d_j = \langle email_j, sex_j, \ldots \rangle \tag{4}$$

We define the set of bug triaging records as:

$$T = \{t_{ij} | i = 1, 2, \ldots, n_B, j = 1, 2, \ldots, n_D\} \tag{5}$$

where the element t_{ij} represents the bug fixing records:

$$t_{ij} = \begin{cases} 1, & if\ b_i\ is\ assigned\ to\ d_j \\ 0, & if\ b_i\ not\ assigned\ to\ d_j \end{cases} \tag{6}$$

We construct a bipartite graph $H = (V, E)$ to represent the correlations of bug-developer. The node set V in H consists of two subsets, that is, $V = B \cup D$. And edge set E is corresponding to the bug-developer correlations:

$$E = \{e_{ij} = \langle b_i, t_{ij}, d_j \rangle | i = 1, \ldots, n_B, j = 1, \ldots, n_D\} \tag{7}$$

It is noteworthy that additional edges can also be added to H. For example, we can add developer–developer edges based on the similarity of their job duties, and bug–bug edges based on the software components they belong. They can help improve the node representation by heterogeneous graph learning.

3.2. Node initialization

This section introduces how to initialize bug nodes and developer nodes. The mainstream semantic embedding learning of bug reports is implemented here.

3.2.1. Bug embedding

Given a bug node and its report text, we need to summarize its title and description. Specifically, we extract word-level information via word2vec (Mikolov et al., 2013) technique and learn sentence-level embedding with a BiRNN (Mani et al., 2019) module.

Firstly, we preprocess the report texts. We remove all URLs, hex code, and stack trace logs and convert text to lowercase. Then we concatenate the title and description texts together and perform word tokenization with the help of NLTK tool (Cho et al., 2014). Furthermore, a preliminary vocabulary of all words is constructed based on the entire corpus, and we remove words whose occurring frequency is less than five times. Finally, a w-dimensional word2vec embedding is learned for each word.

Secondly, we obtain a semantic embedding for the cleaned bug report. The semantic information of a report text is the comprehensive summarization of its words. Therefore, we design a bidirectional GRU module to extract hidden information from the report, which is very common and effective in similar NLP studies (Cheng et al., 2020). As shown in Fig. 3, we take as input the word2vec embeddings of words and adopt two GRU units to scan them both forward and reversely. The output is a cumulative representation of the cleaned report, whose semantics are summarized as expected.

To sum up, the initialization of bug nodes is implemented as follows:

$$\mathbf{p}_{\mathbf{b_i}}^{\mathbf{0}} = BiGRU\left(F_{W2V}\left(F_{pre}\left(title_i, description_i\right)\right)\right) \tag{8}$$

where the F_{pre} is preprocessing operations, F_{W2V} is word embedding learning, and $BiGRU$ is the final step to get the initialization embedding $\mathbf{p}_{\mathbf{b_i}}^{\mathbf{0}} \in R^d$ for bug b_i.

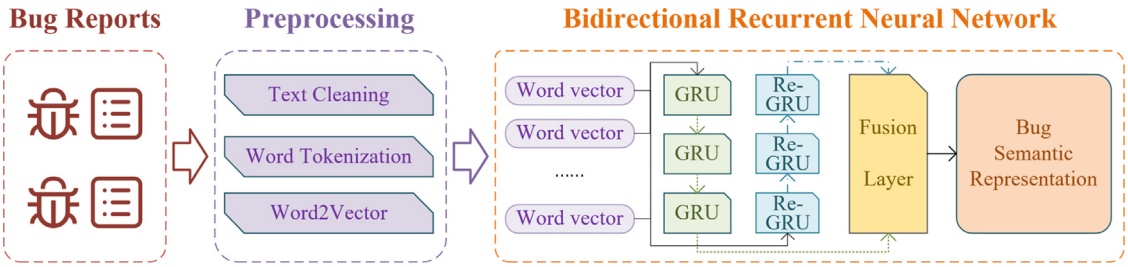

Fig. 3. The operational flow of our NLP-based initialization for bug nodes: a bug report is transformed into a sequence of word vectors, and its semantic representation is extracted via a bidirectional recurrent neural model.

3.2.2. Developer initialization

The initialization of developer nodes is to encode their attributes, which is relatively simple compared to the NLP-based representation of bug nodes. We define the general process as follows:

$$\mathbf{q}_{d_j}^{\mathbf{0}} = Concat\left(Encode\left(attribute_j\right)\right) \tag{9}$$

where $\mathbf{q}_{d_j}^{\mathbf{0}} \in R^d$ is the concatenation of attribute encodings. In this paper, we adopt the developer's email as the only attribute, which is common in issue tracking systems. For example, the emails of three developers (Jack, Tim, and Sam) are Jack@bt.com, Tim@bt.com, and Sam@bt.com, and we can encode them as 001, 010, and 100, just like classification labels. If other attributes (i.e., sex and position) are available, they can be processed similarly and concatenated together. Simple as it is, the one-hot email vector is sufficient for initializing developer nodes. The complicated representation of their programming ability will be collaboratively learned by the following spatial–temporal graph convolution.

3.3. Spatial–temporal graph convolution

This section proposes a novel spatial–temporal convolution strategy for representing developer nodes. therefore representing their programming ability based on their bug-fixing records.

Inspired by state-of-the-art studies on graph convolutional network and recommendation (He et al., 2020), we first design an effective spatial convolution strategy to extract latent programming semantics from neighboring bug nodes. Secondly, we propose an attention-based temporal convolution strategy to extract temporal features as complementary, leading to more comprehensive developer representation.

3.3.1. Spatial convolution

Spatial convolution on a graph has once been implemented as random walk (Nikolentzos and Vazirgiannis, 2020), a sequential learning-based embedding technique. Since it may cause severe information loss, recent studies prefer to conduct neighboring aggregation (Wu et al., 2021), which is a special strategy for the graph data structure. Therefore, we choose neighboring aggregation to fully extract spatial features from connected bug nodes, hoping to improve model performance further.

With developer nodes initialized as $\mathbf{q}_{d_j}^{\mathbf{0}}$, the spatial convolution is defined as:

$$\mathbf{q}_{d_j}^{\boldsymbol{l}} = \mathbf{W}^{\mathbf{S}}\left(F_{aggr}^{S}\left(\mathbf{p}_{b_i}^{\mathbf{0}} \middle| b_i \in Neigh_{d_j}(B)\right) + \mathbf{q}_{d_j}^{\mathbf{0}}\right) \tag{10}$$

where $\mathbf{q}_{d_j}^{\boldsymbol{l}} \in R^d$ is the developer embedding after neighboring aggregation is conducted by l times. $Neigh_{d_j}(B)$ is those bug nodes neighboring to the current developer node, d_j. The initialized embeddings of those bug nodes, $\mathbf{p}_{b_i}^{\mathbf{0}}$, are taken as input to conduct the spatial aggregating operation in F_{aggr}^{S}, usually summing them up or concatenating them together. Then, aggregated bug embeddings are integrated with the developer's initialization embedding, $\mathbf{q}_{d_j}^{\mathbf{0}}$. After being linearly transferred by a matrix $\mathbf{W}^{\mathbf{S}} \in R^{d\times d}$, the spatial convolution-based embedding of the developer node is acquired.

3.3.2. Temporal convolution

As mentioned in Section 2, the programming ability of developers is so complicated that learning spatial features is not enough. For example, a developer, Jack, has previously fixed many C++ bugs, and recently he has been in charge of a few Java bugs. Although Jack's duty has already shifted, which is very common in the industry, spatial convolution on his bug-fixing records probably infers that he is still responsible for C++ bugs. The programming ability of developers can be precisely represented only with the help of a temporal features-capturing algorithm.

In the field of CF, researchers have faced the same problem: the dynamically varied preference of a user is difficult to extract from the sequence of interaction records (Xu et al., 2019). To solve this problem, many recent studies integrate the self-attention mechanism into CF backbone models (Wang et al., 2022; Yu et al., 2022; Wei et al., 2022), and successfully capture temporal features of users' preferences. The essential advantage of self-attention is its competence in capturing the impact of previous data on the prediction of current data, as well as its efficiency compared to LSTM and GRU (Vaswani et al., 2017). It has widely improved many other studies in text understanding (Zhang et al., 2020), traffic prediction (Shi et al., 2021), and industrial sensor modeling (Yuan et al., 2021). Therefore, we would adopt the self-attention mechanism to capture temporal features from a developer's bug-fixing records.

From the perspective of graph learning, the temporal signals lie in the edges that connect a developer and fixed bugs. Enlightened by similar studies that conduct temporal convolution on graphs (Thekumparampil et al., 2018; Wu et al., 2022), we propose a self-attention-based temporal convolution strategy to amend the spatial convolution above:

$$\mathbf{q}_{d_j}^{*} = \mathbf{W}^{\mathbf{T}}\left(F_{aggr}^{T}\left(\mathbf{p}_{b_i}^{\mathbf{0}} \middle| b_i \in Neigh_{d_j}(B)\right) + \mathbf{q}_{d_j}^{\boldsymbol{l}}\right) \tag{11}$$

where $\mathbf{q}_{d_j}^{*} \in R^d$ integrates the temporal feature with the spatial feature in $\mathbf{q}_{d_j}^{\boldsymbol{l}}$. $\mathbf{W}^{\mathbf{T}} \in R^{d\times d}$ is another matrix for linear transformation. F_{aggr}^{T} is a self-attention-based function to extract temporal features from neighboring nodes, which is defined as:

$$F_{aggr}^{temp} = F_{sum}\left(F_{Att}\left(\mathbf{p}_{b_i}^{\mathbf{0}}\mathbf{W}^{\mathbf{Q}}, \mathbf{p}_{b_i}^{\mathbf{0}}\mathbf{W}^{\mathbf{K}}, \mathbf{p}_{b_i}^{\mathbf{0}}\mathbf{W}^{\mathbf{V}}\right)\right) \tag{12}$$

where F_{sum} is the summing up operation, F_{Att} is the self-attention weighing function, within which we have $\mathbf{W}^{\mathbf{Q}} \in R^{d\times h}$, $\mathbf{W}^{\mathbf{K}} \in R^{d\times h}$ and $\mathbf{W}^{\mathbf{V}} \in R^{d\times d}$ as linear transformation matrix, h as the multi-head attention number.

To be more specific, the self-attention assigns temporal weights for each bug embedding, just like this:

$$F_{Att}\left(\mathbf{q}, \mathbf{k}, \mathbf{v}\right) = softmax\frac{\mathbf{q}\mathbf{k}^{\top}}{\sqrt{h}}\mathbf{v} \tag{13}$$

where *softmax* is the activating function to get implicit weights, $\mathbf{q} = \mathbf{p}^0_{b_i}\mathbf{W}^{\mathbf{Q}}$ and $\mathbf{k} = \mathbf{p}^0_{b_i}\mathbf{W}^{\mathbf{K}}$ are h-dimensional vectors, and $\mathbf{v} = \mathbf{p}^0_{b_i}\mathbf{W}^{\mathbf{V}}$ is d-dimensional.

In brief, given a developer d_1 and fixed bugs b_1, b_2, b_3 on the bug-developer bipartite graph, we firstly conduct spatial convolution on $\mathbf{p}^0_{b_1}, \mathbf{p}^2_{b_1}, \mathbf{p}^3_{b_1}$ to get $\mathbf{q}^l_{d_j}$, then we concatenate it with the temporal embedding that comes from the self-attention-based convolution on $\mathbf{p}^0_{b_1}, \mathbf{p}^2_{b_1}, \mathbf{p}^3_{b_1}$. Consequently, the programming ability of d_1 is comprehensively represented by $\mathbf{q}^*_{d_1}$, which extracts both spatial and temporal features from fixed bugs b_1, b_2, b_3.

3.4. Bug embedding augmentation

Although initialized in an NLP way, bug nodes also need graph convolution as augmentation. Unlike the developer's programming ability which may change over time, the semantic information of a bug report is static. Thus we only conduct spatial convolution on bug nodes, which is defined as follows:

$$\mathbf{p}^*_{b_i} = \mathbf{W}^{\mathbf{S}}\left(F^S_{aggr}\left(\mathbf{q}^0_{d_j} | d_j \in Neigh_{b_i}(D)\right) + \mathbf{p}^0_{b_i}\right) \tag{14}$$

where $\mathbf{p}^*_{b_i} \in R^d$ is the final embedding for bug nodes, $\mathbf{W}^{\mathbf{S}} \in R^{d\times d}$ is the same matrix for spatial convolution, and those aggregated $\mathbf{q}^0_{d_j}$s come from neighboring developer nodes. F^S_{aggr} is the spatial aggregating operation that adds all elements up.

3.5. IR-based classifier

Bug triaging prediction means finding potential edges on the bug-developer correlation graph, which corresponds to the graph learning task of link prediction. Since we have learned embeddings for bug and developer nodes via graph convolution operations, the link prediction can be implemented as an inner product, a typical IR-based classifier:

$$\hat{y} = \mathbf{p}^*_{b_i} \cdot \mathbf{q}^*_{d_j} \tag{15}$$

where $\hat{y}$ is the final prediction result. For each pair of bug-developer nodes, it is a binary classification problem to predict whether an edge exists, equivalent to whether that bug should be assigned to the developer. Therefore, an unresolved bug will be assigned to the candidate developer with the highest $\hat{y}$. The ground truth label comes from the dataset, 1 for assigned, and 0 for not assigned.

3.6. Time and space complexity

To calculate the time complexity, we focus on the spatial–temporal graph convolution of developer representation, since it takes up the main part of the whole GCBT. Suppose the number of nodes and edges in the bug-developer correlation graph is $|V|$ and $|E|$, respectively. Let s denote the number of epochs, d denote the embedding size, l denote the number of graph convolution layers, h denote the attention head number. The complexity comes from the following two parts:

- Spatial convolution. Our spatial convolution is based on the linear aggregation of neighboring nodes, just similar to LightGCN (He et al., 2020), and its time complexity is $O(|E|^2 lds)$;
- Temporal convolution. Our temporal convolution is implemented by integrating self-attention with linear aggregation of neighboring nodes, just similar to GAT (Song et al., 2019), and its time complexity is $O(|E|h + |V|dh)$;

Table 1
Summary of bug triaging datasets in the experiments.

Property	Google Chromium	Mozilla Core	Mozilla Firefox
Number of Bugs	351366	295458	149733
Number of Developers	3393	2310	840
Bugs for Learning	109862	110651	22986

The overall complexity of GCBT is $O(|E|^2 lds + |E|h + |V|dh)$, which is basically in the same magnitude as that of LightGCN (He et al., 2020), one of the most efficient graph learning algorithms. Although the self-attention mechanism is introduced, it appears very deep in our GCBT model, bringing little computation cost. The time cost of our GCBT will be further discussed in Section 5.2.

As to the space complexity, the main trainable parameters of GCBT are the embeddings of bug and developer nodes, i.e., $\mathbf{p}^*_{b_i}$ and $\mathbf{q}^*_{d_j}$, whose complexity is $O((|V| + |E|)d)$. Other parameters like linear transformation matrices $\mathbf{W}^{\mathbf{S}}$ and $\mathbf{W}^{\mathbf{Q}}$ only take up a small and fixed amount of space. In brief, the space complexity of GCBT is relatively low.

4. Experiment settings

This section details how we conduct extensive experiments to evaluate our proposed GCBT for bug triaging. Specifically, we introduce our dataset and benchmark, implementation details, and our research questions.

4.1. Dataset and benchmark

Like many studies (Alazzam et al., 2020), we choose three public datasets that are obtained from popular open-source systems: Google Chromium (GC) (Badashian et al., 2015), Mozilla Core (MC), and Mozilla Firefox (MF) (Bettenburg et al., 2008). The summary of the datasets is provided in Table 1. Here is the introduction of their details:

1. **Google Chromium.** We adopt bug reports in GC that range from August 2008 to July 2015. Their attributes include "bug title", "description", "owner", and "reported time". The value in the "owner" field is an email address, indicating the developer in charge (Bhattacharya and Neamtiu, 2010). We choose bugs as the benchmark whose status is "Verified" or "Fixed".
2. **Mozilla Core.** We adopt bug reports in MC that range from April 1998 to March 2014. These bugs have attributes like "status", "title", "description", and "assigned to". The developer email in the "assigned to" field is the ground truth result for bug triaging. Bug reports with status as "verified fixed", "resolved fixed", and "closed fixed" are chosen for learning.
3. **Mozilla Firefox.** We adopt bug reports in MF that range from July 1999 to June 2015. Similar to MC, bug reports also have four attributes, and we choose bug reports whose status is "fixed" and other attributes are all valid.

To split the bug-fixing records, the training, validating, and testing data account for 80%, 10%, and 10%, respectively. To get a $\langle positive, negative\rangle$ sample pair for a developer in the training set, we set one fixed bug as the positive sample, and randomly sample one bug that has not been interacted as the negative sample. To reduce noises from inactive developers, we implement a k-sample strategy on datasets to ensure that every selected developer has enough training bugs, which is generally accepted by relevant studies (Anvik and Murphy, 2011; Jonsson et al., 2016; Zaidi et al., 2022). In this paper, we set the threshold parameter to 10.

4.2. Evaluation metric

We choose the top-N hit ratio as our evaluation metric for bug triaging precision. For a given bug, the trained model will calculate the probability for every developer, and those top-N most probable developers are output as the prediction results. Therefore, the top-N hit ratio $R_{hit} \in R$ is acquired like this:

$$R_{hit} = \frac{NUM_{hit}}{NUM_{total}} \tag{16}$$

where $NUM_{hit} \in R$ is the number of bugs whose ground-truth developer lies in the top-N prediction list, and $NUM_{total} \in R$ is the total number of bugs. In the following experiment, we will separately calculate the hit ratio of top-1 to top 10.

4.3. Implementation details

The Adam optimizer is chosen for model training, the training batch size is set as 1000, the largest training epoch number is set as 300, the learning rate is set as 0.002, the parameters of the network are initialized by the default Xavier distribution, the embedding size is 64, a 10-step early stopping is adopted to prevent overfitting.

As to the hyper-parameters of GCBT, the number of convolution layers in spatial convolution is set as 1, the developers with top-10 probability are chosen as the output for each bug. The implementation code of GCBT, based on PyTorch, is available at: https://gitee.com/papercodefromasexd/GCBT.git. We run GCBT on a computer with Windows 10 and 16G RAM, an Intel Core i7-7700 as CPU, and PyCharm as IDE.

4.4. Research questions

To make a complete assessment of the feasibility and performance of GCBT, we focus on three following research questions:

1. **RQ1: How is the performance of our GCBT in bug triaging?** Particularly, we compare several representative studies with our GCBT, and investigate differences between the bug classification-based paradigm and the correlation-based paradigm.
2. **RQ2: What is the superior strategy for bug-developer graph convolution?** Among bug triaging frameworks that adopt graph learning techniques, we compare several typical strategies for graph convolution. Moreover, we analyze the effectiveness of GCBT in capturing temporal features.
3. **RQ3: How does the hyper-parameter in GCBT affect the performance of bug triaging?** We conduct additional experiments for key hyper-parameters in our GCBT to explore how they affect the performance of bug triaging.

5. Evaluation

This section shows the results of our investigations for the three research questions.

5.1. RQ1: performance of bug triaging

We select seven baselines to compare with our GCBT, and list the comparison results of performance and time cost in Table 2. These baseline methods can be categorized as bug classification-based methods, including CBR, DBRNN-A, DABT, PMI, and correlation-based methods, including CosTriage, GRCNN:

- CBR (Anvik et al., 2006) converts bug titles and descriptions to numeric vectors and classifies them under an SVM classifier. This is one of the most classical bug report classification methods.
- DBRNN-A (Mani et al., 2019) is a deep learning-based framework that proposes an effective bidirectional recurrent neural network to learn the syntactic and semantic features from bug reports.
- DABT (Jahanshahi et al., 2021) builds a homogeneous graph of bug–bug dependencies and utilizes graph statistic learning to excavate them, as a complementary to bug reports semantics.
- PMI (Zaidi et al., 2022) builds a heterogeneous graph upon word–word and word-bug co-occurrences and adopts pointwise mutual information for weighting word–word edges, based on which the bug report is represented and classified.
- CosTriage (Park et al., 2011) combines NLP-based bug representation with a developer profiling module. It extracts semantic features from bug reports for labeling developers. It can be seen as an elementary version of GCF.
- GRCNN (Wu et al., 2022) builds a homogeneous graph of developer collaborative relationships to represent developers, alongside the NLP representation module for bug reports.

Firstly, we review methods that belong to the classification-based paradigm. Compared to CBR that utilizes early NLP techniques such as TF-IDF, the deep learning-driven DBRNN-A shows better performance on hit ratio. Such improvement can be attributed to the bidirectional recurrent learning method that mines more syntactic information from complicated text sentences. NLP-aided classification gets improvement when graph learning is introduced, which explains the better performance of DABT and PMI. It is worth noting that PMI outperforms DABT. DABT utilizes classical statistic learning to extract features from external bug–bug correlations. In contrast, PMI implements modern graph convolution to capture semantic features from internal word-bug correlations, which makes it outstanding in all classification-based methods.

Secondly, the three correlation-based methods significantly outperform the classification-based methods. CosTriage represents developers by simple collaborative learning and yet achieves considerable improvement. GRCNN and GCBT introduce deep graph learning into correlation modeling, bringing more improvement. Specifically, GRCNN introduces a developer representation module that learns cooperation willingness from the bug-tossing graph. It is a pity that GRCNN ignores the bug-developer correlations, which limits its performance. Contrarily, our GCBT takes full advantage of bug-developer correlations to learn node representations, especially those of developers' programming ability, thus yielding more precise bug triaging results.

Thirdly, we compare the training time cost per epoch of all methods, except CBR, DABT, and CosTriage that are untrainable with the conventional technique SVM. The overall time costs among the three datasets are consistent with their data scale. That is, a larger dataset needs more time to train. Furthermore, DBRNN-A is the slowest model, which is attributed to the heavy bidirectional RNN structure it adopts. The fastest model is GRCNN because it only processes a developer–developer correlation graph, which is relatively smaller than the word-bug graph of PMI and the bug-developer graph of GCBT. Although our GCBT adopts self-attention that may bring high computation cost, it appears in the very deep position of our model, which makes the time cost well tolerated, slightly higher than GRCNN. Such efficiency can also be attributed to the lightweight design of our spatial convolution algorithm.

5.2. RQ2: graph learning strategy

Among several graph learning-based methods, different graph convolution strategies are implemented to handle bug-developer

Table 2
The performance and time cost of two types of methods on three datasets.

Datasets	Methods	Top-N hit ratio											Time per epoch (s)
		top-1	top-2	top-3	top-4	top-5	top-6	top-7	top-8	top-9	top-10	*Average*	
GC	CBR	0.1407	0.2031	0.2200	0.2699	0.2832	0.2945	0.3173	0.3223	0.3469	0.3413	*0.2739*	–
	DBRNN-A	0.1707	0.2124	0.2309	0.2766	0.3279	0.3309	0.3456	0.3560	0.3792	0.3905	*0.3021*	1.2131
	DABT	0.1691	0.2188	0.2250	0.2758	0.3249	0.3308	0.3394	0.3574	0.3861	0.4166	*0.3044*	–
	PMI	0.1833	0.2141	0.2363	0.2783	0.3305	0.3397	0.3488	0.3580	0.4093	0.4209	*0.3119*	0.7220
	CosTriage	0.2139	0.2227	0.2499	0.2895	0.3379	0.3519	0.3635	0.3800	0.4119	0.4257	*0.3247*	–
	GRCNN	0.2425	0.2588	0.2974	0.3120	0.3438	0.3689	0.3720	0.3854	0.4144	0.4554	*0.3451*	0.7179
	GCBT	**0.2519**	**0.2724**	**0.3166**	**0.3472**	**0.3701**	**0.3924**	**0.4046**	**0.4311**	**0.4634**	**0.5251**	***0.3775***	0.7310
MC	CBR	0.1395	0.1525	0.2308	0.2521	0.2692	0.3269	0.3793	0.4024	0.4650	0.4812	*0.3099*	–
	DBRNN-A	0.1503	0.1715	0.2449	0.2770	0.3092	0.3397	0.3810	0.4203	0.4871	0.5063	*0.3287*	1.2046
	DABT	0.1435	0.1751	0.2898	0.2733	0.2921	0.3428	0.3948	0.4151	0.4862	0.5110	*0.3324*	–
	PMI	0.1675	0.1808	0.2611	0.2858	0.3277	0.3528	0.4042	0.4293	0.5093	0.5253	*0.3444*	0.7211
	CosTriage	0.1836	0.2051	0.2968	0.3061	0.3411	0.3647	0.3964	0.4527	0.5238	0.5501	*0.3621*	–
	GRCNN	0.2051	0.2101	0.3214	0.3451	0.3692	0.4066	0.4329	0.4682	0.5500	0.5782	*0.3887*	0.7141
	GCBT	**0.2149**	**0.2215**	**0.3462**	**0.3930**	**0.4121**	**0.4364**	**0.4670**	**0.4912**	**0.5741**	**0.6024**	***0.4159***	0.7268
MF	CBR	0.2179	0.2316	0.2503	0.3325	0.4020	0.4510	0.4755	0.4970	0.5366	0.5607	*0.3955*	–
	DBRNN-A	0.2236	0.2476	0.2627	0.3457	0.4197	0.4539	0.4769	0.5203	0.5500	0.5741	*0.4074*	1.0982
	DABT	0.2336	0.2377	0.2631	0.3449	0.4173	0.4575	0.4899	0.5181	0.5416	0.5799	*0.4084*	–
	PMI	0.2476	0.2534	0.2844	0.3574	0.4229	0.4675	0.4993	0.5323	0.5647	0.5884	*0.4218*	0.6112
	CosTriage	0.2521	0.2689	0.2982	0.3720	0.4383	0.4705	0.5056	0.5485	0.5927	0.5954	*0.4342*	–
	GRCNN	0.2636	0.2739	0.3025	0.3802	0.4442	0.4739	0.5210	0.5500	0.6044	0.6325	*0.4446*	0.6039
	GCBT	**0.2696**	**0.2954**	**0.3149**	**0.3887**	**0.4693**	**0.5190**	**0.5579**	**0.5892**	**0.6279**	**0.6891**	***0.4721***	0.6255

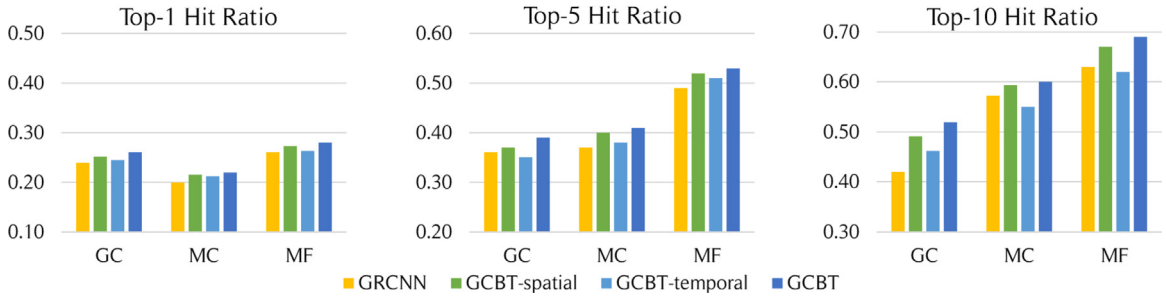

Fig. 4. The impact of four different graph convolution strategies on the top-5 and top-10 hit ratio.

correlations. For example, GRCNN conducts random-walk-based convolution for developer node representation, which belongs to sequential pattern recognition (Nikolentzos and Vazirgiannis, 2020). Our GCBT, conducting neighboring convolution, is based on spatial pattern recognition. Here we investigate which one is generally superior via comparative experiments. Particularly, we add two variants of GCBT, GCBT-spatial, and GCBT-temporal, as an ablation study for revealing the individual contribution of GCBT's spatial–temporal convolution strategy. Here GCBT-spatial implements node convolution as the linear aggregation of neighboring nodes, and GCBT-temporal implements convolution as self-attention-based aggregation.

The comparative result between sequential and graph pattern recognition is displayed in Fig. 4. Based on the bug-developer correlation graph, we implement node convolution strategies of different patterns and compare their top-1, top-5, and top-10 hit ratios. The three neighboring convolution-based methods outperform the sequential learning-based ones in general. The GCBT-spatial convolution is competent enough, and GCBT-temporal convolution provides an additional improvement on the final hit ratio of triaging.

Furthermore, we conduct an additional experiment to explore whether GCBT is competent in capturing temporal features. Enlightened by GRCNN (Wu et al., 2022), we resample the original datasets to obtain data groups for long-term category (L-G1, L-G2, L-G3), medium-term (M-G1, M-G2, M-G3), and short-term category (S-G1, S-G2, S-G3), making temporal features in bug-developer correlations more distinct. A long-term group contains data of continuous five years for training and data of the following year for testing. Medium-term and short-term groups are similarly measured in months and weeks. Based on these data groups, we compare the top-10 hit ratio of GCBT and the two representative baselines, PMI and GRCNN, hoping to reveal how well the temporal features can be extracted. As shown in Fig. 5, when the data scale shrinks from years to weeks, the performance of all methods decreases, which is mainly because of the insufficient training data. Nevertheless, our GCBT generally shows better resistance than PMI and GRCNN. On medium-term data groups, GCBT is notably as good as on long-term groups, which can be owed to the temporal strategy of our GCBT that effectively captures the dynamic variation of developers' programming ability. Even on short-term groups with the sparsest data, GCBT generates prediction results that are as good as those of PMI and GRCNN on medium-term data.

5.3. RQ3: impact of hyper-parameters

Based on RQ2, we further study how the performance of our proposed GCBT is affected by two key hyper-parameters: neighboring convolution layers and fixed bug numbers in sampled datasets.

5.3.1. Neighboring convolution layers

In our GCBT, we conduct developer node convolution on its direct neighboring bug nodes, or the first-layer nodes. Theoretically, such convolution can also be conducted on second and third-layer

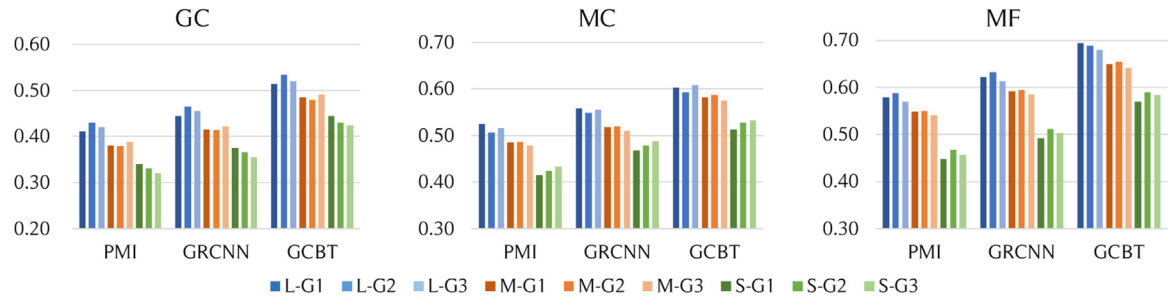

Fig. 5. The effectiveness of capturing temporal features in resampled long-term, medium-term, and short-term data groups.

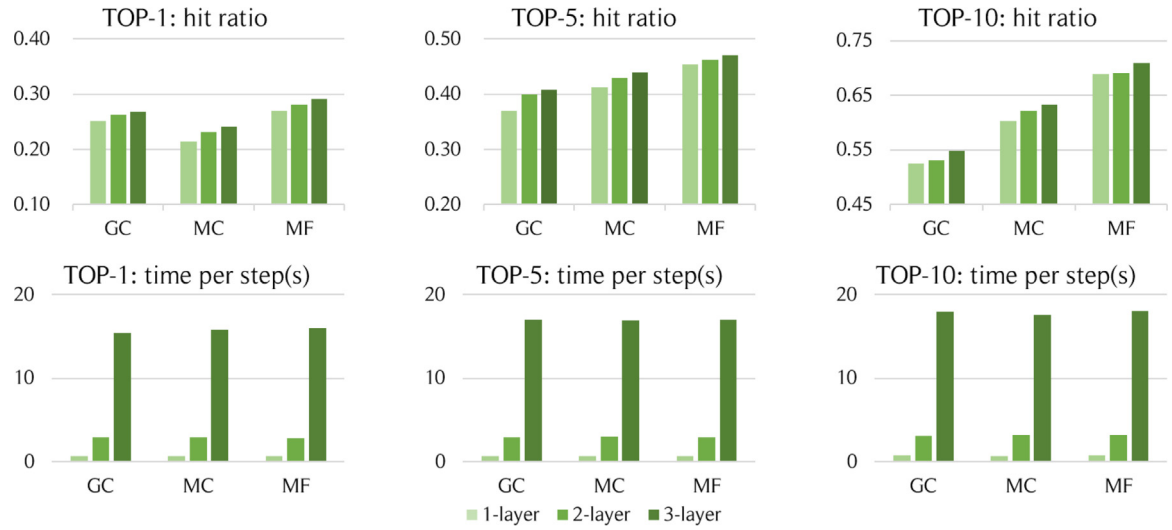

Fig. 6. The impact of different convolution layers on GCBT's performance and time cost.

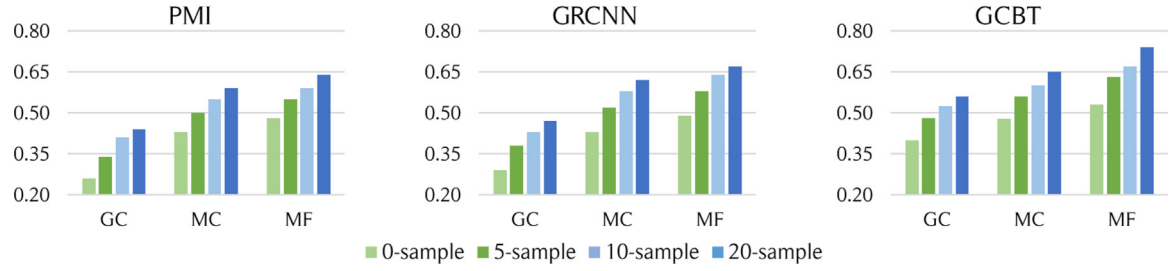

Fig. 7. The impact of dataset resampling scale on the hit ratio of two representative bug triaging methods.

nodes. More convolution layers may obtain more comprehensive representations, yet also bring embedding oversmoothing and computation cost spiking. We conduct several comparative experiments to investigate how the number of neighboring convolution layers affects performance. As shown in Fig. 6, we compare the top-1, top-5 and top-10 hit ratio and training time of GCBT under 1-layer, 2-layer and 3-layer convolution. The benchmark datasets are 10-sample-based GC, MC, and MF. When the number of convolution layers increases, the hit ratio of bug triaging prediction improves, which is too tiny compared to the extra time cost. Thus, we choose only one convolution layer to balance the trade-off between performance and cost.

5.3.2. Fixed bug numbers of developers

The k-sample strategy determines the number of fixed bugs for each developer in the benchmark dataset, which leads to different training data sparsity and potential performance damage (Natarajan et al., 2020). Additional to the 10-sample strategy adopted in previous experiments, we try three more sampling scales, 0-sample, 5-sample, and 20-sample. PMI and GRCNN are selected as two comparison methods with GCBT, as is shown in Fig. 7. The prediction performance suffers the most when benchmarks are acquired by 0-sample, since the fixed bugs of every developer are pretty insufficient. Nevertheless, our GCBT shows strong resistance to such data sparsity because it captures temporal features from bug-developer correlations as a complementary. When the sampling scale gets larger, for example, 20-sample, we can see that all the methods perform much better. To make the experiments differentiated, we decided to use 10-sample sub-datasets.

6. Discussion

In this section, we discuss the threats to the validity and applications of our study.

6.1. Threats to validity

The threats to validity include internal threats and external threats. We will analyze them as follows.

6.1.1. Internal threats

The dataset benchmarks for the experiment are created by resampling from issue tracking datasets available online, which may lead to statistically biased learning. With original datasets, Google Chromium, Mozilla Core, and Mozilla Firefox, their data distribution is already not uniform, and our k-sample strategy could increase the difficulty levels for learning. We alleviate it by training and testing triaging models on sub-datasets with different degrees of k-sample strategies. The results reveal that the prediction generally performs well on all sub-datasets.

There may also exist cognitive biases in the ground truth data of the three datasets (Tecimer et al., 2022). For example, a few fixed bugs should have been assigned to other appropriate developers, or some "closed" bug reports should be reopened because of system architecture reconstruction. These cognitively biased data labels could damage the generality and precision of trained models. To clean the ground truth data, we check the three datasets and remove bug-fixing records that have been reopened or reassigned, which can effectively exclude cognitive bias (Tüzün et al., 2021). Moreover, we leave out the last year's data in the three datasets, in case some bugs may be reopened or reassigned more than a year later.

Besides, we only use bug fixing reports for developer representation, not considering personal attributes like area, age, gender, or expertise. As mentioned above, context information is unavailable for most developers, not only in datasets but also in real-world issue tracking systems. Enlightened by CF recommenders, it is intuitive and feasible to represent developers' programming ability based on fixed bugs. Adding unnecessary attributes may bring inadvisable complexity and thus lead to conflicts.

6.1.2. External threats

There are more effective bug triaging frameworks than we investigated in this paper. However, some studies have not provided open-source code, or their code cannot function well. To solve this problem, we collect existing bug triaging studies and examine their implementation codes. Concerning frameworks that are representative but unavailable, we contact the authors to get the latest version, and rewrite to reconstruct when we receive no response.

There are more datasets, such as Eclipse, than the three ones we pick, which may cause a problem with generalizability for our GCBT. Nevertheless, the three datasets, GC, MC, and MF, are collected from famous large software projects, which makes them representative enough. Moreover, we want to demonstrate how effective the correlation-based paradigm is for developer representation. Even if our GCBT may need to catch up in other untested datasets, we still believe that introducing developer representation into intelligent software engineering is a promising direction for future research.

6.2. Applications

In this subsection, we discuss the implications of GCBT to the industry and academia.

6.2.1. An aid for bug triaging

Considering the implication of our work to the industry, GCBT shows the potential to aid bug triaging. Although its top-1 hit ratio needs to be higher for recommending an appropriate developer, its top-10 hit ratio rises to around 0.6. The top-N accuracy will be higher with a larger N, which makes it a possible aid to support manual bug triaging. Specifically, GCBT could be adopted as a component in the back-end of an issue tracking system to generate a list of the top-20 (or even more) developers as candidates. This could help the project manager efficiently decide to whom a bug should be assigned. With the progress of future studies on bug triaging, such a semi-automatic process can be upgraded to entirely automatic, and improve the efficiency of software maintenance.

6.2.2. New research possibilities

To the best of our knowledge, GCBT is the first model that explicitly exploits CF for bug triaging. Unlike most works that emphasize bug report classification, we focus on the essence of bug triaging, the matching of bug-developer correlations. In future work, we would like to make GCF more entrenched with bug triaging scenarios and open up new research possibilities. Going beyond self-attention-based aggregation, we plan to integrate other state-of-the-art dynamic graph learning techniques into developer representation. Moreover, we will focus on pre-training and fine-tuning in bug triaging, that is, to pre-train a model which captures universal and transferable patterns across multiple datasets, and fine-tune it on other upcoming datasets. Another promising direction is exploring the self-supervised learning paradigm to help capture more latent features as a powerful complementary.

7. Related work

In recent years, various methods for bug triaging have been proposed, ranging from classical machine learning to modern deep learning, and can be categorized as bug report classification and bug-developer correlation matching. In this section, we present a summary of typical approaches.

7.1. Classification-based triaging

Classification-based paradigm has been adopted for bug triaging ever since 2004 (Murphy and Cubranic, 2004). In the early work, classic text classification algorithms, such as Naive Bayes, are utilized to learn semantic features from bug reports. Since then, more and more researchers have paid attention to bug report classification. For example, the expectation–maximization algorithm is applied to improve Naive Bayes and get a more precise triaging prediction (Xuan et al., 2017). With the development of machine learning, diverse classification techniques are successively introduced, and SVM is one of the most effective ones applied to further improve the prediction accuracy (Anvik and Murphy, 2011). Recently, NLP has achieved remarkable achievement and helped bug report classification raise its accuracy to a higher level. For example, bidirectional RNN models (Xia et al., 2016; Mani et al., 2019) and graph representing models (Zaidi and Lee, 2021; Zaidi et al., 2022) can extract semantic and structural features from long sentences in bug reports.

Bug triaging is a correlated process between bugs and developers, other than merely classifying bug reports. Aware of that, a few studies try to represent developers as complementary. Unlike bugs, developers in the issue tracking system seldom have context information, and early works utilize crowdsourcing platforms as contextual information: based on a developer's e-mail, they crawl open-source platforms (Al-batlaa et al., 2018)

and programming forums (Sajedi Badashian et al., 2016) to acquire activity records. In addition, bug tossing activities among developers are also extracted from the tracking system as context data (Wu et al., 2022; Guo et al., 2020). However, such techniques only work for a few developers and need more generalizability.

In summary, the bug-developer correlation is the rich information deposit to be mined, which could benefit developer representation and bug triaging. It deserves more attention from academia.

7.2. Correlation-based triaging

Inspired by CF recommenders (Linden et al., 2003), a few researchers try to take advantage of bug-developer correlation data and represent developers (Hu et al., 2014; Wang et al., 2013). Among modern machine learning techniques, graph learning is the most competent in representing entities based on their correlations. Several studies try to reconstruct the correlations of bug-developer as a graph and conduct node embedding (Alazzam et al., 2020) by capturing spatial signals. Moreover, the attention mechanism is adopted to extract temporal signals in graphs (Wu et al., 2022). Generally speaking, it has become a consensus that graph learning is very promising for bug triaging models (Hu et al., 2014; Yu et al., 2021; Zaidi and Lee, 2021).

Among existing correlation-based studies, GRCNN (Wu et al., 2022) is the most similar to ours. As a typical graph learning-based framework, it constructs a graph with bugs and developers as two node sets. Bug nodes are conventionally initialized by processing their bug reports, and developers are initialized via mining the bug-tossing records. Their representations are separately learned and used for bug triaging prediction. It is a pity that bug-developer correlations still need to be explored for GRCNN. In contrast, our GCBT fully utilizes bug-developer correlations on a GCF-based framework. With the spatial–temporal convolution strategy, we can deeply reveal implicit features from bug-developer correlations and achieve better prediction performance.

8. Conclusion

This paper proposes a GCF-based framework for the bug triaging task. Unlike many studies that represent bugs or developers separately, we creatively propose GCBT to model the correlations of bug-developer. With triaging activities constructed as a bipartite graph, we design a spatial–temporal convolution strategy to learn the representations of developer nodes, comprehensively capturing the implicit features of programming ability. Bug nodes are initialized based on their report texts and then assigned to proper developers by matching bug-developer representation pairs. Extensive experiments across three datasets show the competence of GCBT in bug triaging. This work represents an initial attempt to exploit correlation modeling in bug triaging scenarios and could open up new research possibilities.

CRediT authorship contribution statement

Jie Dai: Methodology, Project administration, Writing – original draft, Writing – review & editing. **Qingshan Li:** Conceptualization, Funding acquisition, Supervision. **Hui Xue:** Data curation, Methodology. **Zhao Luo:** Data curation, Investigation. **Yinglin Wang:** Validation, Reviewing. **Siyuan Zhan:** Validation, Reviewing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared a link to the data and code in the manuscript.

Acknowledgments

This work was supported by National Natural Science Foundation of China [grant numbers 61972300, U21B2015], and Young Talent Fund of Association for Science and Technology in Shaanxi [grant numbers 20220113].

References

Al-batlaa, A., Abdullah-Al-Wadud, M., Hossain, M.A., 2018. A review on recommending solutions for bugs using crowdsourcing. In: 21st Saudi Computer Society National Computer Conference. NCC, (2018), pp. 1–4.

Alazzam, I., Aleroud, A., Al Latifah, Z., Karabatis, G., 2020. Automatic bug triage in software systems using graph neighborhood relations for feature augmentation. IEEE Trans. Comput. Soc. Syst. 7 (5), 1288–1303.

Anvik, J., Hiew, L., Murphy, G.C., 2006. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering. pp. 361–370.

Anvik, J., Murphy, G.C., 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. 20 (3), 1–35.

Badashian, A.S., Hindle, A., Stroulia, E., 2015. Crowdsourced bug triaging. In: 2015 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 506–510.

Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S., 2008. Duplicate bug reports considered harmful…really? In: IEEE International Conference on Software Maintenance. vol. 2008, IEEE, pp. 337–345.

Bhattacharya, P., Neamtiu, I., 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.

Cheng, Y., Yao, L., Xiang, G., Zhang, G., Tang, T., Zhong, L., 2020. Text sentiment orientation analysis based on multi-channel cnn and bidirectional gru with attention mechanism. IEEE Access 8, 134964–134975.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using rnn encoder–decoder for statistical machine translation, arXiv preprint arXiv:1406.1078.

Davidson, J., Liebald, B., Liu, J., Nandy, P., Van Vleet, T., Gargi, U., Gupta, S., He, Y., Lambert, M., Livingston, B., et al., 2010. The youtube video recommendation system, In: Proceedings of the Fourth ACM Conference on Recommender Systems. pp. 293–296.

Goldberg, D., Nichols, D., Oki, B.M., Terry, D., 1992. Using collaborative filtering to weave an information tapestry. Commun. ACM 35 (12), 61–70.

Grover, A., Leskovec, J., 2016. node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 855–864.

Guo, S., Zhang, X., Yang, X., Chen, R., Guo, C., Li, H., Li, T., 2020. Developer activity motivated bug triaging: via convolutional neural network. Neural Process. Lett. 51 (3), 2589–2606.

Hamilton, W.L., 2020. Graph representation learning. Synth. Lect. Artif. Intell. Mach. Learn. 14 (3), 1–159.

He, X., Deng, K., Wang, X., Li, Y., Zhang, Y., Wang, M., 2020. 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.

Hu, H., Zhang, H., Xuan, J., Sun, W., 2014. Effective bug triage based on historical bug-fix information. In: IEEE 25th International Symposium on Software Reliability Engineering. IEEE, pp. 122–132.

Jahanshahi, H., Chhabra, K., Cevik, M., Baar, A., 2021. Dabt: A dependency-aware bug triaging method. In: Evaluation and Assessment in Software Engineering. pp. 221–230.

Jonsson, L., Borg, M., Broman, D., Sandahl, K., Eldh, S., Runeson, P., 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. Empir. Softw. Eng. 21 (4), 1533–1578.

Karypis, G., 2001. Evaluation of item-based top-n recommendation algorithms. In: Proceedings of the Tenth International Conference on Information and Knowledge Management. pp. 247–254.

Li, Z., Zhong, H., 2021. Revisiting textual feature of bug-triage approach. In: 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1183–1185.

Linden, G., Smith, B., York, J., 2003. Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Comput. 7 (1), 76–80.

Mani, S., Sankaran, A., Aralikatte, R., 2019. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In: Proceedings of the ACM India Joint International Conference on Data Science and Management of Data. pp. 171–179.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems. p. 26.

Mohsin, H., Shi, C., Hao, S., Jiang, H., 2022. Span: A self-paced association augmentation and node embedding-based model for software bug classification and assignment. Knowl.-Based Syst. 236, 107711.
Murphy, G., Cubranic, D., 2004. Automatic Bug Triage using Text Categorization, in: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering. Citeseer, pp. 1–6.
Natarajan, S., Vairavasundaram, S., Natarajan, S., Gandomi, A.H., 2020. Resolving data sparsity and cold start problem in collaborative filtering recommender system using linked open data. Expert Syst. Appl. 149, 113248.
Nikolentzos, G., Vazirgiannis, M., 2020. Random walk graph neural networks. Adv. Neural Inf. Process. Syst. 33, 16211–16222.
Park, J.W., Lee, M.W., Kim, J., Hwang, S.W., Kim, S., 2011. Costriage: A cost-aware triage algorithm for bug reporting systems. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 25. pp. 139–144.
Sajedi Badashian, A., Hindle, A., Stroulia, E., 2016. Crowdsourced bug triaging: Leveraging q & a platforms for bug assignment. In: International Conference on Fundamental Approaches To Software Engineering. Springer, pp. 231–248.
Shi, X., Qi, H., Shen, Y., Wu, G., Yin, B., 2021. A spatial–temporal attention approach for traffic prediction. IEEE Trans. Intell. Transp. Syst. 22 (8), 4909–4918.
Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S., 2013. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In: 10th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 2–11.
Song, W., Xiao, Z., Wang, Y., Charlin, L., 2019. Session-based social recommendation via dynamic graph attention networks. In: Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining. pp. 555–563.
Tamrawi, A., Nguyen, T.T., Al-Kofahi, J., Nguyen, T.N., 2011. Fuzzy set-based automatic bug triaging (nier track). In: Proceedings of the 33rd International Conference on Software Engineering. pp. 884–887.
Tecimer, K.A., Tüzün, E., Moran, C., Erdogmus, H., 2022. Cleaning ground truth data in software task assignment. Inf. Softw. Technol. 106956.
Thekumparampil, K.K., Wang, C., Oh, S., Li, L.J., 2018. Attention-based graph neural network for semi-supervised learning. arXiv preprint arXiv:1803.03735.
Tüzün, E., Erdogmus, H., Baldassarre, M.T., Felderer, M., Feldt, R., Turhan, B., 2021. Ground truth deficiencies in software engineering: when codifying the past can be counterproductive. IEEE Softw..
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention Is all you need, 2017. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. pp. 6000–6010.
Wang, R., Wu, Z., Lou, J., Jiang, Y., 2022. Attention-based dynamic user modeling and deep collaborative filtering recommendation. Expert Syst. Appl. 188, 116036.
Wang, S., Zhang, W., Wang, Q., 2014. Fixercache: Unsupervised caching active developers for diverse bug triage. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.
Wang, S., Zhang, W., Yang, Y., Wang, Q., 2013. Devnet: Exploring developer collaboration in heterogeneous networks of bug repositories. In: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 193–202.
Wei, W., Huang, C., Xia, L., Xu, Y., Zhao, J., Yin, D., 2022. Contrastive meta learning with behavior multiplicity for recommendation. In: Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining. pp. 1120–1128.
Wu, H., Ma, Y., Xiang, Z., Yang, C., He, K., 2022. A spatial–temporal graph neural network framework for automated software bug triaging. Knowl.-Based Syst. 241, 108308.
Wu, J., Wang, X., Feng, F., He, X., Chen, L., Lian, J., Xie, X., 2021. Self-supervised graph learning for recommendation. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 726–735.
Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N., Wang, X., 2016. Improving automated bug triaging with specialized topic model. IEEE Trans. Softw. Eng. 43 (3), 272–297.
Xu, C., Zhao, P., Liu, Y., Sheng, V.S., Xu, J., Zhuang, F., Zhou, X., 2019. Graph contextualized self-attention network for session-based recommendation. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. pp. 3940–3946.
Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., Wu, X., 2014. Towards effective bug triage with software data reduction techniques. IEEE Trans. Knowl. Data Eng. 27 (1), 264–280.
Xuan, J., Jiang, H., Ren, Z., Yan, J., Luo, Z., 2017. Automatic bug triage using semi-supervised text classification, arXiv preprint arXiv:1704.04769.
Xuan, J., Jiang, H., Ren, Z., Zou, W., 2012. Developer prioritization in bug repositories. In: 34th International Conference on Software Engineering. ICSE, IEEE, pp. 25–35.
Yu, X., Wan, F., Du, J., Jiang, F., Guo, L., Lin, J., 2021. Bug triage model considering cooperative and sequential relationship. In: International Conference on Wireless Algorithms, Systems, and Applications. Springer, pp. 160–172.
Yu, B., Zhang, R., Chen, W., Fang, J., 2022. Graph neural network based model for multi-behavior session-based recommendation. GeoInformatica 26 (2), 429–447.
Yuan, X., Li, L., Shardt, Y.A.W., Wang, Y., Yang, C., 2021. Deep learning with spatiotemporal attention-based LSTM for industrial soft sensor model development. IEEE Trans. Ind. Electron. 68 (5), 4404–4414.
Zaidi, S.F.A., Lee, C.G., 2021. Learning graph representation of bug reports to triage bugs using graph convolution network. In: 2021 International Conference on Information Networking (ICOIN). pp. 504–507.
Zaidi, S.F.A., Woo, H., Lee, C.G., 2022. A graph convolution network-based bug triage system to learn heterogeneous graph representation of bug reports. IEEE Access 10, 20677–20689.
Zhang, Z., Xu, Y., Zhao, H., Li, Z., Zhang, S., Zhou, X., Zhou, X., 2020. Semantics-aware BERT for language understanding. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 9628–9635.
Zhu, T., Harrington, P., Li, J., Tang, L., 2014. Bundle recommendation in ecommerce. In: Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 657–666.

Jie Dai, born in 1993, Ph.D. His current research interests include recommender system and intelligent software maintenance.

QingShan Li, born in 1973, Ph.D., professor. His current research interests include agent-oriented software engineering, dynamic software evolution, software architecture, intelligent data analysis and decision support system.

Hui Xue, born in 1998, M.S. His current research interests include intelligent data analysis and software architecture.

Zhao Luo, born in 1998, M.S. His current research interests include data mining and intelligent software maintenance.

Yinglin Wang, born in 1996, M.S. His current research interests include data mining and intelligent software maintenance.

Siyuan Zhan, born in 1998, M.S. Her current research interests include intelligent data analysis and software architecture.