

Automatic Bug Triaging via Deep Reinforcement Learning

Yong Liu ¹, Xuexin Qi ¹, Jiali Zhang ¹, Hui Li ^{1,*} , Xin Ge ¹ and Jun Ai ² 

- ¹ Information Science and Technology College, Dalian Maritime University, Dalian 116026, China; yong@dlmu.edu.cn (Y.L.); qxx@dlmu.edu.cn (X.Q.); zjlgzzy@126.com (J.Z.); ge_xin@dlmu.edu.cn (X.G.)
- ² School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093, China; aijun@outlook.com
- * Correspondence: li_hui@dlmu.edu.cn

Abstract: Software maintenance and evolution account for approximately 90% of the software development process (e.g., implementation, testing, and maintenance). Bug triaging refers to an activity where developers diagnose, fix, test, and document bug reports during software development and maintenance to improve the speed of bug repair and project progress. However, the large number of bug reports submitted daily increases the triaging workload, and open-source software has a long maintenance cycle. Meanwhile, the developer activity is not stable and changes significantly during software development. Hence, we propose a novel bug triaging model known as auto bug triaging via deep reinforcement learning (BT-RL), which comprises two models: a deep multi-semantic feature (DMSF) fusion model and an online dynamic matching (ODM) model. In the DMSF model, we extract relevant information from bug reports to obtain high-quality feature representation. In the ODM model, through bug report analysis and developer activities, we use a strategy based on the reinforcement learning framework, through which we perform training while learning and recommend developers for bug reports. Extensive experiments on open-source datasets show that the BT-RL method outperforms state-of-the-art methods in bug triaging.

Keywords: bug triaging; recurrent neural network; deep reinforcement learning



Citation: Liu, Y.; Qi, X.; Zhang, J.; Li, H.; Ge, X.; Ai, J. Automatic Bug Triaging via Deep Reinforcement Learning. *Appl. Sci.* **2022**, *12*, 3565. <https://doi.org/10.3390/app12073565>

Academic Editor: Chang Yong Song

Received: 26 February 2022

Accepted: 29 March 2022

Published: 31 March 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Bug triaging can effectively improve the efficiency of bug repair; hence, it has garnered increasing attention in studies regarding software testing and maintenance. As a bug is identified in a code, the bug should be reported to the bug tracking system; subsequently, the bug report should be assigned to a suitable developer who is proficient in fixing the bug. Initially, the bug reports are manually assigned to the developers. However, open-source software projects have a long maintenance lifecycle, and many bug reports are produced daily. For instance, 23,491 bug reports were produced by 553 developers in the OpenOffice project between 1 June 2011 and 7 April 2013. Furthermore, 341 bugs were reported per month, and each developer had to fix 42.48 bugs on average [1].

There are many traditional algorithms [2–4] to solve many fundamental problems [5–7], but the conventional algorithms in bug triaging cannot play a good role. Therefore, many researchers use machine learning to solve this problem. To improve the efficiency of bug triaging, automatic bug report assignment [8–11], as well as various automatic allocation methods, have been proposed [12–16]. John et al. [17] reported that text classification methods can be employed in bug triaging; additionally, experimental studies proved the effectiveness of this method. Subsequently, Ahsan et al. [18] proposed an automatic bug classification system based on a latent semantic index and a support vector machine (SVM). Xuan et al. [19] recommended bug triaging based on semi-supervised learning. The methods above used existing classifiers and only considered a single feature of the text in the bug reports. Therefore, Yang et al. [20] proposed a bug severity prediction and bug classification method for multi-feature and topic models based on bug reports. As data-imbalanced

distributions exist in bug report datasets, only considering the text information limits the allocation efficiency improvement. Therefore, Wu et al. [21] presented an algorithm, known as “developer recommendation with k-nearest-neighbor search and expertise ranking”, based on developer interest and developer expertise to recommend a list of developers.

However, most of the models mentioned above are offline. Offline means that the existing bug reports are used as the training set to train the model. After the model is stable, the newly submitted bug reports are assigned directly. It is uncertain whether such a model will perform well in any case because the previously trained model may not learn sufficient information, or the new bug may come from a completely new field, which will result in some disadvantages in the offline methods. In summary, there are two challenges, as follows, to be addressed.

The multidimensional feature challenge. How can we effectively integrate and consider the multidimensional features of bug reports? A uniform mechanism to integrate multiple features does not exist and specific features to be considered are not known. Hence, considering only a single feature cannot fully represent the bug report itself, and multidimensional features do not reflect the importance of features and the internal relationship between features.

Automatic learning challenge. How can bug reports be assigned in real time based on historical bug fixed records? Since historical records include a long time-span and the time distributions of bug fixing records for each developer are varied, it is difficult to learn from historical data and then assign tasks to suitable developers. Therefore, frequently changing developers may result in secondary task assignment.

Therefore, we present a novel method known as auto bug triaging via deep reinforcement learning (BT-RL), which comprises two sub-adversarial models, i.e., the deep multi-semantic feature fusion (DMSF) model and the online dynamic matching (ODM) model. In the DMSF model, we adopted a recurrent neural network (RNN) to extract the multidimensional features of bug reports and capture the diversity of text structure, and the multiplication between elements of the feature vector was obtained for feature fusion to obtain high-quality information. In the ODM model, the feature information of bug reports was “state”, and the bug report process was assigned to one of the developers with a probability of “action” in reinforcement learning (RL). By analyzing the multidimensional features, the probabilities of the developers to be selected for fixing the bugs can be obtained; hence, we can identify the developer with the highest probability value. Subsequently, based on the current specific feedback results and their evaluation, the corresponding rewards can be presented. Therefore, the model can perform self-training and adapt to the association between bug reports and developers to achieve the expected results.

We evaluated the performance of BT-RL experimentally based on four open bug repositories (NetBeans, OpenOffice, Mozilla, and Eclipse). The experimental results show that BT-RL is superior to the baseline. From the perspective of model allocation accuracy, BT-RL achieved 52%, 54%, 68%, and 78% in terms of the Top-5 on the OpenOffice, NetBeans, Mozilla, and Eclipse datasets, respectively, and improved the LDA_SVM by up to 4%, 11%, 9%, and 8%, respectively. Our model significantly improves the models proposed by researchers in recent years. After we train the model, it can be deployed on the server to work in real time. Figure 1 shows the procedure of the real-time bug triaging. Specifically, a developer identifies a bug, then he/she will submit a bug report with necessary information describing this bug to the bug management system. Memory replay will store the information of the bug reports and the developers. Then, the BT-RL model will leverage the information stored in memory replay to enhance its performance, and assign this bug report to the suitable developer.

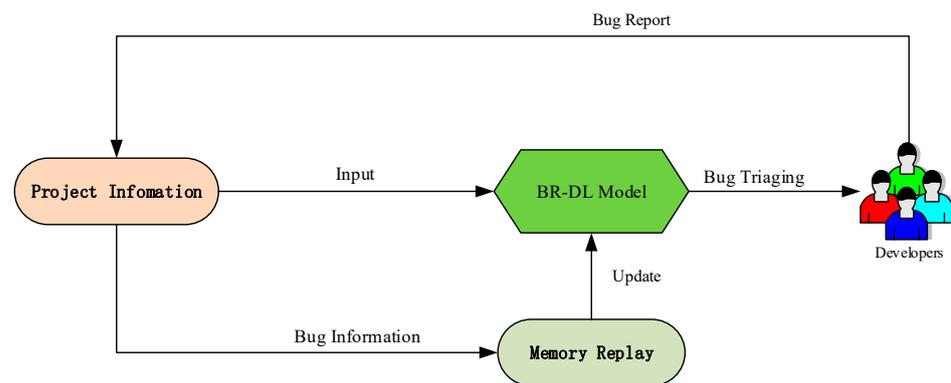


Figure 1. The online workflow of BT-RL.

To summarize, the main contributions of this study are as follows:

1. A bug triaging model based on RL is presented, in which feature extraction is performed using neural networks, and newly created bug reports can be assigned online in real time.
2. A bidirectional RNN that integrates multidimensional features is adopted, and it can capture the information of different levels more comprehensively while rendering the representation of bug reports more robust for bug triaging, compared with the SVM and the method proposed by Yang [20].
3. The BT-RL model on real-world datasets (NetBeans, OpenOffice, Mozilla, and Eclipse) is validated. Comprehensive experimental results show that the BT-RL model performs better than state-of-the-art models.

The remainder of this paper is structured as follows. Section 2 lists the related works. Section 3 introduces our motivation. We present the details of our approach in Section 4. We evaluate our approach and analyze its performance in Section 5. In Section 6, we discuss threats and validity in our experiments to our work. Finally, in Section 7, we conclude this work and discuss future work.

2. Related Works

In recent years, owing to sophisticated software project functions and logic complexity, software projects inevitably yielded more defects. The traditional manual method of bug triaging is time-consuming and labor-intensive. In addition, unsuitable bug triaging will affect bug repair efficiency. Accordingly, bug triaging is the main problem in software quality assurance. Several methods have been proposed hitherto for solving bug triaging.

Cubranic et al. [22] attempted to use an SVM and decision tree classification to solve this problem. Tamrawi et al. [13] proposed a method based on a fuzzy set and developer cache to recommend developers, using a fuzzy set to identify developers capable of repairing defects from the developer cache. Xie et al. [23] proposed a method that constructs a theme model based on the bug repair history, establishes a relationship between the developer and the error report theme, and simulates the developer's interest in error reports and professional knowledge. Nguyen et al. [24] proposed a topic-based log-normal regression model of fixed time to predict the defect resolution time and proposed bug repair suggestions. In addition, Zhang et al. [25] proposed a method that uses a topic model to obtain developers' interest and experience in specific error reports; additionally, they analyzed the relationship between candidate developers and the most active reporter. Yan et al. [26] analyzed document semantics and built a discriminative probability latent semantic analysis (DPLSA) model. The DPLSA model analyzed the historical experience of the developers by establishing a document subject distribution to recommend appropriate developers.

Xuan et al. [27] first analyzed the relationship between reviewers and repairers of bug reports, data scale of comments, and collaboration information between comments;

subsequently, they proposed a method based on a complex network to distribute bug reports. Somasundaram et al. [28] proposed the LDA_SVM and LDA_KL methods. First, they used LDA to obtain the topic distribution of all bug reports. LDA_SVM uses an SVM to classify bug reports through topic distribution. LDA_KL calculates the similarity between bug reports via KL divergence based on the topic distribution. Yin et al. [29] proposed a novel hybrid method based on diversified feature selection and an ensemble extreme learning machine. They used a genetic algorithm (GA)-based ensemble extreme learning machine (ELM) training classifier. Moreover, Khatun et al. [30] proposed an approach known as Bug fixing and Source commit activity-based Bug Assignment (BSBA), which combines the expertise and recency of both bug fixing and source commit activities. Yadav et al. [31] proposed a novel and improved two-phase bug triaging method that involves developer profile creation and assignment phases. Furthermore, developer profiles were built using individual contributions and performance assessment metrics.

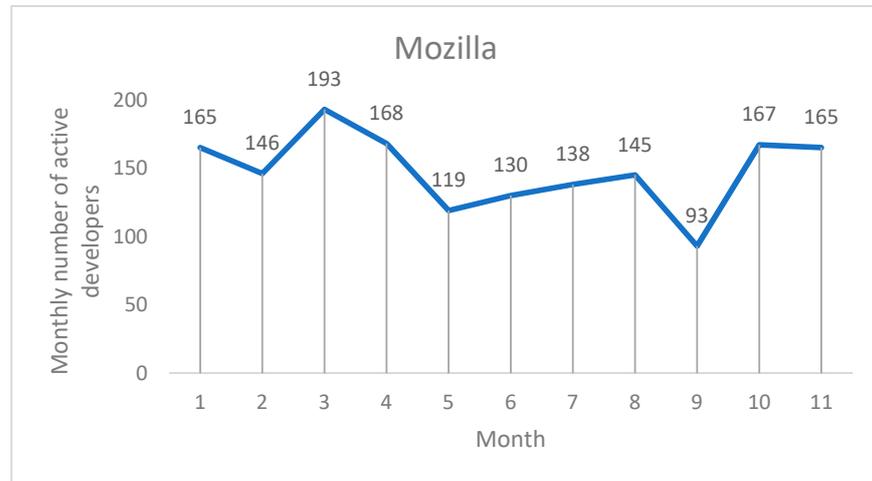
Hadi et al. [32] introduced a bug triaging method, called Dependency-aware Bug Triaging (DABT), which leverages natural language processing and integer programming to assign bugs to appropriate developers. Su et al. [33] proposed a learning to rank framework that learns to assign components to bugs from correct, erroneous, and irrelevant bug-component assignments in the history. To inform the learning, we constructed a bug tossing knowledge graph which incorporates not only goal-oriented component tossing relationships but also rich information about component tossing community, component descriptions, and historical closed and tossed bugs, from which three categories and seven types of features for bug, component, and bug–component relation can be derived. Wu et al. [34] proposed a novel spatial-temporal dynamic graph neural network (ST-DGNN) framework, including a joint random walk (JRWalk) mechanism and a graph recurrent convolutional neural network (GRCNN) model. Guo et al. [35] presented a new automatic bug triaging approach which is based on convolution neural network (CNN) and developer activities. Yutaro et al. [36] proposed a release-aware bug triaging method that aims to increase the number of bugs that developers can fix by the next release date during open-source software development.

3. Motivation

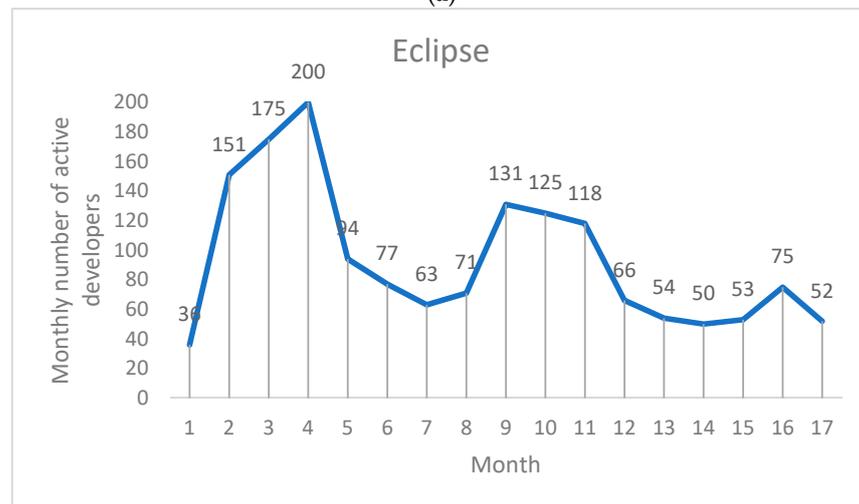
In recent years, more and more software projects are managed online, especially the bug report management in software test. Actually, the number of submitted bug reports is increasing every year, and the developers have to face more and more bug repair tasks. Figure 2 shows the average number of different developers participating in each of the four datasets per month, in which the activity of the developers can be determined based on the trend lines. As shown, the participation of the developers in the same dataset fluctuated significantly in different periods. Further investigation revealed that the developer activities varied significantly, and the projects were performed by different developers. In other words, many developers abandoned the project, whereas many new developers participated in the same period. Therefore, frequent changes in developers render preventing offline models from performing reasonable bug triaging in real time, which is a typical offline problem. Hence, we used the online mode of RL bug triaging, which can assign tasks while learning, and performed a more reasonable bug triaging based on the characteristics of personnel flow information in real time.

In addition, developers participating in development change frequently in the lifecycle of software projects. Generally speaking, a large software project contains many subprojects, and developers can divide each subproject into several components. Therefore, the size of the project is positively related to the number of subprojects and components. The statistics of collected bug reports are shown in Table 1. Regarding the 18,793 bug reports provided by Mozilla, 1021 developers participated in bug repair, and 772 developers participated in repairing the 41,830 bug reports of Eclipse. The last two columns of Table 1 show the average number of bug fixes per developer and the average number of bugs per month.

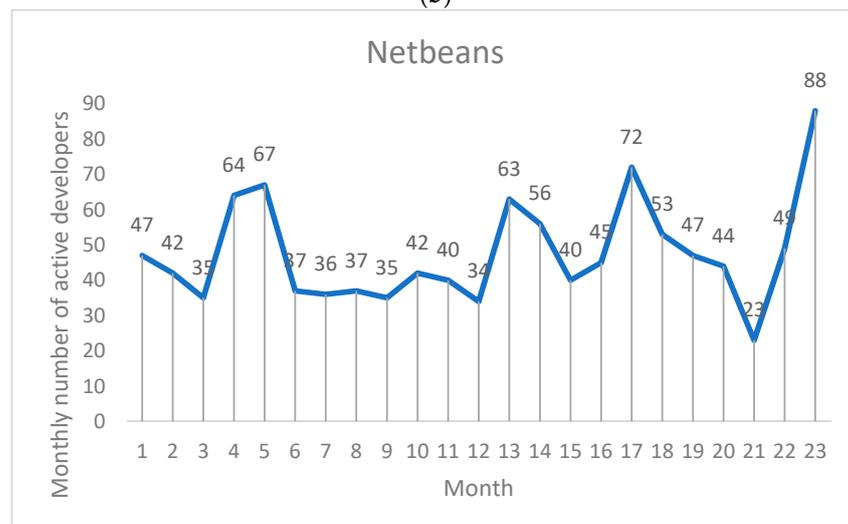
Any mistake in task assignments may result in unnecessary task redistributions and prolong the bug fixing time.



(a)



(b)



(c)

Figure 2. Cont.

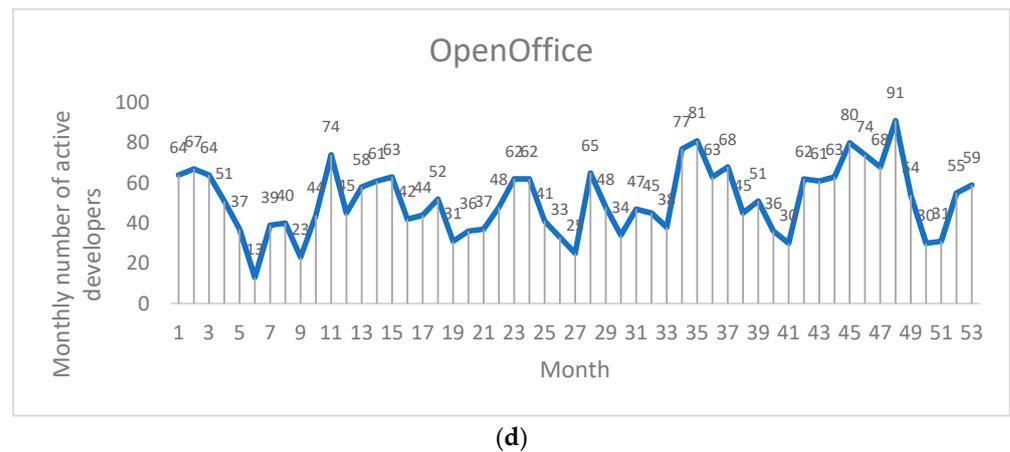


Figure 2. The statistics of the average number of different developers participating in each of the four datasets per month. (a) Mozilla; (b) Eclipse; (c) Netbeans; (d) OpenOffice.

Table 1. Statistics of collected bug reports.

| Project | Period | Bug Reports | Developers | Subprojects | Components | Avg_dev_fix | Avg_mon_bug |
|------------|-----------------------------------|-------------|------------|-------------|------------|-------------|-------------|
| NetBeans | 1 January 2008 to 11 January 2010 | 22,691 | 226 | 33 | 148 | 85.30 | 961 |
| OpenOffice | 1 March 2007 to 7 April 2013 | 23,491 | 553 | 36 | 116 | 42.48 | 341 |
| Mozilla | 23 June 2009 to 3 June 2010 | 18,793 | 1021 | 12 | 167 | 18.40 | 1705 |
| Eclipse | 1 January 2008 to 23 July 2009 | 41,830 | 772 | 7 | 114 | 54.25 | 2418 |

Since the offline model is static, the offline model cannot use data such as bug reports generated in real time to improve its performance. Therefore, we decided to employ reinforcement learning to solve this problem. The model automatically assigns new defects in the project to the right developers. At the same time, the bug memory pool stores the bugs, bug reports, and other information that the project generates regularly. Then, we used the bug memory pool information to update model parameters and continuously improve model performance.

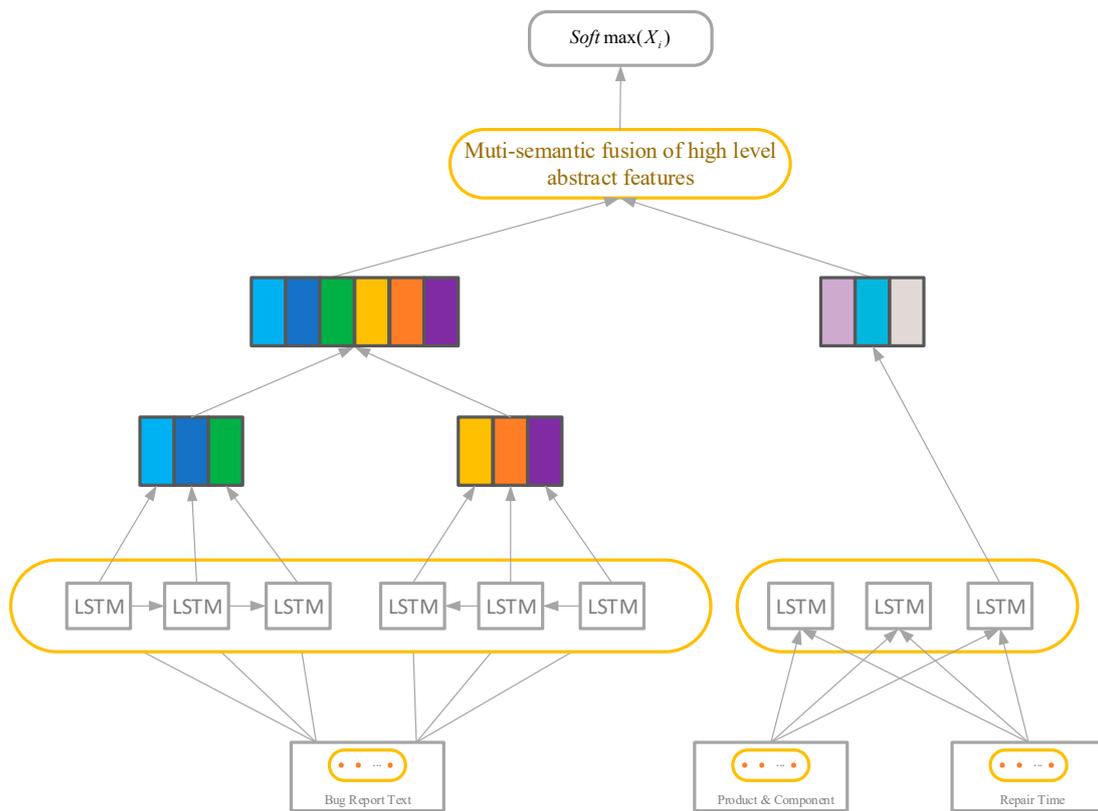
4. Bug Triaging via Deep Reinforcement (BT-RL) Model

In this section, we first present an overview of the workflow of BT-RL in Section 4.1. Subsequently, we present the details of the components used in BT-RL in the following sections.

4.1. Overview

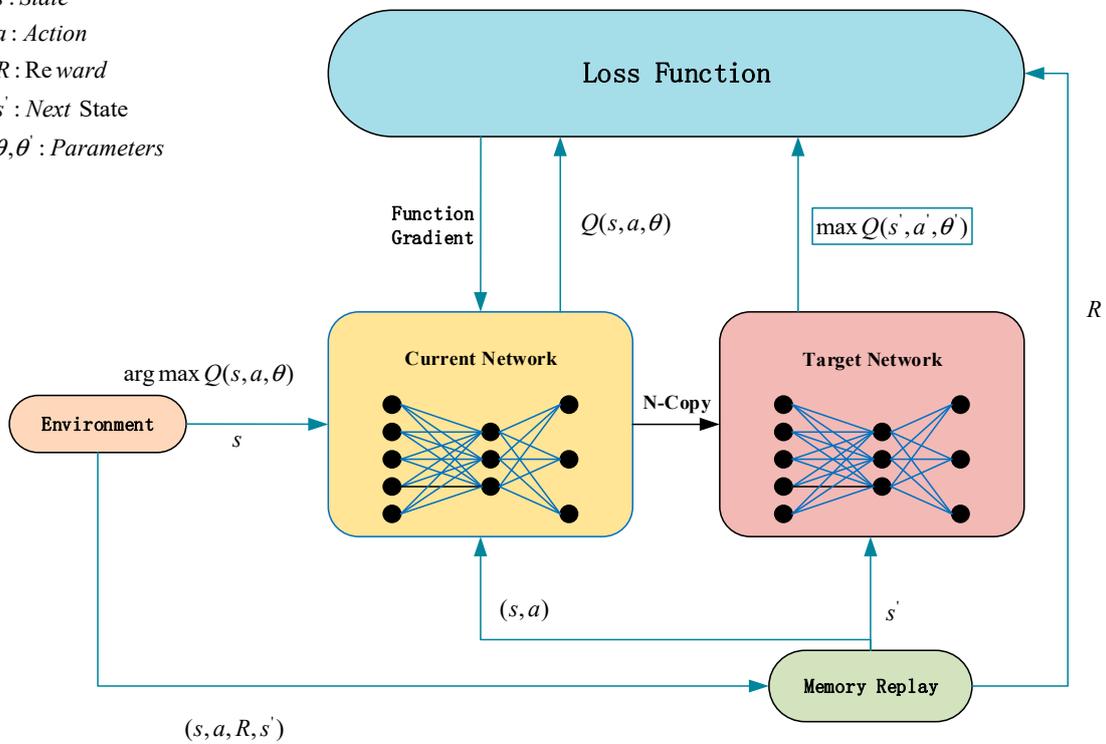
As mentioned above, most state-of-the-art bug triaging models are offline. Hence, the models cannot be adjusted based on the situation presented by the bug reports and developers; as such, the performance yielded may be unsatisfactory. Conversely, BT-RL is trained for each new task, and it fully learns the relevant information of the task. In other words, BT-RL explores and learns from the real environment by addressing bug triaging problems. As illustrated in Figure 3, the BT-RL method comprises two sections: (a) a DMSF model and (b) an ODM model. The DMSF model aims to extract high-level features of multidimensional information, whereas the ODM model aims to identify the most suitable developer to fix the bug by analyzing relevant information from the bug reports.

The details of BT-RL are described as Algorithm 1. First, we initialized the environment, including the replay memory pool D , learning rate L , batch size B , step-threshold C , and preprocessed bug reports. Subsequently, we used the ϵ -greedy strategy to select the developer, evaluate the selection, store it in D (i.e., line 10), and use it as a sample. During agent exploring after C steps, we selected samples B from D randomly for training. We obtained different rewards (r) based on different results and updated the network parameters based on the gap between reality and expectation. Finally, we updated θ via linear interpolation.



(a) Deep Multi-Semantic Feature Fusion Model

s : State
 a : Action
 R : Reward
 s' : Next State
 $\theta, \bar{\theta}$: Parameters



(b) Online Dynamic Matching Model

Figure 3. The framework of BT-RL.

Algorithm 1: BT-RL Algorithm via DQN

Input: input D to capacity N, B, C, L
Out-put: Top-K developers for each bug task
Initialize Q network with weights θ ;
Initialize Q_t network with weights $\theta_t = \theta$;
For episode = 1, M **do**;
 Initialize and preprocessed bug s_1
 $\phi_1 = \phi(s_1)$;
 For $t = 1, T$ **do**;
 With probability ϵ select random action a_t ;
 Otherwise select
 $a_t = \operatorname{argmax} Q(\phi(s_t), a; \theta)$;
 Execute action a_t in emulator and observe
 reward r_t and s_t ;
 Set $s_t = s_{t+1}$ and preprocess
 $\phi_{t+1} = \phi(s_{t+1})$;
 Store transition $\phi_t, a_t, r_t, \phi_{t+1}$ in D ;
 Set

$$y_t = \begin{cases} r_t & \text{if terminal} \\ r_t + \gamma \max_{a'} Q_t(\phi_{t+1}, a'; \theta_t) & ; \end{cases}$$

 Perform a gradient descent step;
 Update network parameter $Q_t = Q$;
 End For;
End For;

4.2. DMSF Model

As shown in Figure 3a, RNN structures, which are used for different neural network units, can extract features at different levels. The input primarily includes bug reports containing a specific description of the bug, i.e., the bug id, subproject, and component, and fixed time, which indicates the specific time point. These characteristics are important issues in bug triaging and are widely used in experiments.

Since the RNN can be used for embedding in the pooling module to extract the text features of the bug reports, a one-way recurrent network was used to extract the features of developer participation at a specific moment and implement the multi-semantic fusion of high-level abstract features. The model architecture of feature extraction is more suitable for obtaining the information of multisource data. In the following process, the advantages of feature extraction are primarily elaborated based on the following three aspects:

High-level feature extraction of text: First, this process uses word or one-hot embedding to convert the text information of bug reports into vectors of equal length. Second, the bidirectional RNN is used to extract text features. It not only obtains a feature vector that reduces the dimension, but also can be fused easily with other features. Third, the maximum feature pooling method is used to reduce the dimensionality of the feature vector and filter out the larger values in the feature vector. Finally, feature fusion is performed on the feature vectors to produce the high-level multi-semantic features of the current bug repair tasks.

High-level feature extraction of developer engagement: The bug repair records of developers in the same subproject and components are arranged from far to near to form the developer activity sequence feature, which is the input into the one-way RNN for extracting high-level features of developer participation. Moreover, the output of the last node of the RNN is used as the final high-level feature of developer participation.

Multi-semantic fusion of high-level features: For various high-level features of bug triaging, the appropriate form of fusion for the following activity should be used. Typical high-level feature fusion methods include concatenation, element-to-element addition, and element-to-element multiplication; nevertheless, the method to be used is yet to be determined.

After these steps, we can obtain a candidate list of developers who can fix the bug. Specifically, we used score to represent the matching degree of a developer d in developer set A to fix the bug, and the score can be expressed as:

$$\text{Score}(tf, af, d) = W_d(tf \odot af) + b_d, \quad (1)$$

where tf is the high-level feature of a text, af is the high-level feature of developer participation, \odot is the fusion method between features, W_d is the weight vector of a developer d , and b_d is the bias value. Among them, the levels of developer participation differ during bug report processing. We collected a sequence of bug reports from developers with the same subproject and component information as current bug reports in the historical bug fix record, with the maximum number of bug reports collected being 25. If there are more than 25 bug reports, the most recent 25 are selected. Among them, the repair interval for bug reports is six months before the current point in time.

Subsequently, we obtained the probability of assigning bug triaging tasks to developer d , and the probability was normalized by the high-level features after fusion using the Softmax layer.

$$P(y = d|tf, af) = \frac{\exp(\text{Score}(tf, af, d))}{\sum_{d'} \exp(\text{Score}(tf, af, d'))}, \quad (2)$$

Loss represents the mean squared error between the predicted and target values. When the error is more than minor, it means that the model's predicted value is closer to the target value. Therefore, our goal is to minimize the loss during model training. The loss function is:

$$\text{Loss} = E \left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a' + \omega)}_{\text{Target}} - Q(s, a, \omega) \right)^2 \right], \quad (3)$$

To better learn the experience of bug repairing online, we adopted the parameter approximator of the Q function (Q -learning). Q -Learning finds the optimal policy to maximize the expected value of the total reward at any successive steps from the current state.

During back propagation, we used the gradient descent method, and the gradient of the parameter for the loss function can be expressed as follows:

$$\frac{dL(\omega)}{d\omega} = E \left[(T) * \frac{dQ(s, a, \omega)}{d\omega} \right], \quad (4)$$

$$T = r + \gamma \max_{a'} Q(s', a', \omega), \quad (5)$$

Subsequently, we can train each bug report as a newly generated task, and each update is based on the previous results.

4.3. ODM Model

In this section, we present the various factors pertaining to bug triaging. The overall execution process of the model is as follows. Through the analysis of the features of bug data, the ϵ -greedy strategy was adopted for developer selection. Based on the evaluation of the allocation results, we presented corresponding rewards and punishments for building the label data to update the strategy dynamically based on the information in the memory pool. RL is suitable for addressing Markov decision process (MDP) problems. Some state information contains all relevant histories. If the current state is known and all historical information is no longer required, then the current state can be used to determine the future. The bug assignment problem is similar to the MDP problem. When the distribution of the sample does not drift with time or the scale of the elegant is small, and it is a sequence decision problem that is not well-labeled, the effect of RL will be excellent. In the training process based on self-learning, the label need not be considered.

4.3.1. MDP

The MDP is used to describe the RL problem. We regard the bug triaging problem as an MDP. The framework of our model is shown in Figure 3. In general, the RL task corresponds to a four-tuple: $E = \langle S, A, P, R \rangle$, where S is the state space, A is the action space, P is the state transition probability, and R is the reward function. This will be explained in detail in the following paragraphs.

1. States (S)

S represents the state space, which includes a series of bug triaging tasks, $s, s \in S$, and s_i represent the current bug report text, the subproject and component, and the fixed time of developer when making a decision on the i -th bug of all tasks.

2. Action (A)

A is the action space, which is a set of developers who can perform the tasks. Specifically, it is a developer sequence corresponding to the first three months of bug reports with the same subprojects and components as the current bug report before time t . For each bug s , each developer that can be selected is defined as $a, a \in A(s)$.

3. Transition probability ($P_{ss'}^a$)

We define the state transition probability as $P_{ss'}^a$, and it can be expressed as:

$$P_{ss'}^a = P(s'|s, a) = P(S_{t+1} = s' | S_t = s, A_t = a), \tag{6}$$

which represents the probability that the agent will move to the next bug, s' , at time $t + 1$ when it executes a developer, $a \in A(s)$, in bug s at time t .

4. Reward Function (R)

R is the reward function that provides scalar feedback regarding the performance of an agent. It is the learning direction of the model and determines the skills and efficiency of the model. When an agent selects a for bug s , the reward function provides feedback to the agent to perform corresponding rewards and punishments for different results. Based on the model requirements, we represent the reward function in a sparse form as follows:

$$r_t = \begin{cases} r^{reach}, & \text{if } i \in Topk \\ r^{crash}, & \text{if otherwise} \end{cases} \tag{7}$$

The value of R can take two values. Suppose the target developer is in the recommended top-k sequence. In that case, the reward is defined as r^{reach} , typically a positive value, and encourages the model to develop in a favorable direction. In contrast, we define r^{crash} . i represent the ground-truth developer in the dataset.

4.3.2. Policy and Expectation Function

For the entire process, the expected cumulative reward at time t can be formulated as shown in Equation (8):

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{m=0}^{\infty} \gamma^m R_{t+m+1}, \tag{8}$$

where the term $0 \leq \gamma \leq 1$ is the discount rate, depending on how one intends to prioritize early rewards over later ones, and m indicates the number of bug reports.

A value function, v_{π} , is considered, which is the expected cumulative reward that the agent can receive in the long term, starting from a specified state and following the policy π thereafter. Specifically, the value of a state can be measured using two parameters, i.e., the state value functions and action value functions, depending on whether it depends on the state alone or both the state and the action.

State value function ($v_\pi(s)$) represents the expected cumulative return from using strategy π to assign bug s , and this implies that the strategy uniquely determines the function distribution.

$$v_\pi(s) = E_\pi \left[\sum_{m=0}^{\infty} \gamma^m R_{t+m+1} | S_t = s \right], \tag{9}$$

where the state-action value function ($q_\pi(s, a)$) represents the expected cumulative return using strategy π to assign a for bug s . It can be formulated as follows:

$$q_\pi(s, a) = E_\pi \left[R_{t+1} + \gamma \sum_{m=0}^{\infty} \gamma^m R_{t+m+2} | S_t = s, A_t = a \right], \tag{10}$$

During training, to avoid falling into a local optimum, we used the policy π to balance the exploration and utilization of models, as follows:

$$\pi(s_t) = \begin{cases} \operatorname{argmax}_{a \in A} Q(s_t, a), & \text{if } \mu \leq \varepsilon \\ \tilde{a}, & \text{otherwise} \end{cases}, \tag{11}$$

where μ is a random value generated from $[0, 1]$ per round, ε is the exploration rate, and \tilde{a} is the random developer.

Each of the state value pairs is assigned a Q -value, which quantifies the expected discounted return achieved by performing action a at state s .

$$Q_{i+1}(s, a) = E_{s'} [r + \lambda \max Q_i(s', a') | s, a], \tag{12}$$

where γ denotes the developer's instant return, the discount rate, and s' is the next state after s .

5. Results

5.1. Datasets

We conducted the bug triaging experiments on several major open-source datasets, including NetBeans, OpenOffice, Mozilla, and Eclipse. The statistics of the four datasets are shown in Table 2.

Table 2. Raw datasets for NetBeans, OpenOffice, Mozilla, and Eclipse.

| Project | Period | Bug Reports | Developers | Token Length | Subprojects | Components |
|------------|-----------------------------------|-------------|------------|--------------|-------------|------------|
| NetBeans | 1 January 2008 to 11 January 2010 | 15,824 | 129 | 6997 | 14 | 134 |
| OpenOffice | 1 March 2007 to 7 April 2013 | 20,516 | 169 | 4637 | 36 | 116 |
| Mozilla | 23 June 2009 to 3 June 2010 | 14,565 | 229 | 5545 | 12 | 167 |
| Eclipse | 1 January 2008 to 23 July 2009 | 26,949 | 291 | 9017 | 7 | 114 |

Part of the bug reports extracted from the bug report repositories of the datasets (including Mozilla, OpenOffice, NetBeans, and Eclipse) was used in the experimental data. The content in the dataset primarily included the bug id, developer, report, component, subproject, and repair status. Prior to the experiment, data preprocessing operations were performed. As the data were unbalanced, the data must be preprocessed, of which several operations for preprocessing are listed below:

1. When extracting bug reports, repair status "fixed" is retained.
2. Remove bug reports fixed by invalid developers (specifically defined as "unassigned", "issues", "needs confirmation", "nobody", "webmaster", "inbox").
3. Remove bug reports fixed by inefficient developers (inefficient developers are those who repair less than n (which equals 20) fixed reports).
4. Count and filter high- and low-frequency words to reduce noise.

5. Extract the developer engagement information and collect the developer engagement sequence that is the same as the subproject and module information for the current bug report. The maximum number of recorded bug reports is 25 (if the upper limit is exceeded, the closest 25 bug reports are selected). The time interval is three months ahead of the current time node. It is observed that 80% of the developer engagement sequences in 3 months can yield 20 bug reports or more.

Using the dataset above, we segregated the entire dataset into multiple segments of training and test datasets and performed experiments using the baseline and our method, respectively. We obtained all datasets in GitHub (<https://github.com/zjl95/DRL-bug-traige>, accessed on 17 February 2022).

5.2. Baselines

We compared the performance of the BT-RL model based on the following four baselines. For studies that provided the source code, we directly reproduced their methods using the datasets. Otherwise, we rebuilt their models with reference to the studies. LDA_KL: Kalyanasundaram et al. [28] used latent Dirichlet allocation (LDA) to obtain the topic distribution of all bug reports and calculated the similarity between bug reports via the Kullback–Leibler (KL) divergence based on the topic distribution. LDA_SVM: Kalyanasundaram et al. [28] used the SVM to classify bug reports through topic distribution. However, the contribution of the developers who participated in the collaboration could not be disregarded. DERTOM: Xie et al. [23] proposed approach models according to developers' interest and expertise in bug-resolving activities based on topic models that were built from their historical bug resolving records. DRETOM recommended a ranked list of developers who can participate in resolving the new bug according to the developers' interest and expertise in resolving it. DeepTriaging: Mani et al. [37] proposed a novel bug report representation using a deep bidirectional recurrent neural network with attention, which learns syntactic and semantic features from long word sequences in an unsupervised manner. Attention enables the model to remember and attend to important parts of text in a bug report.

5.3. Evaluation Metrics

We used accuracy as the evaluation criterion. Accuracy is the proportion of correct predictions out of the total number of cases examined, which compares estimates of pre-test and post-test probabilities. ACC_Topk represents the percentage of correctly allocated samples among all the samples. We primarily considered the accuracy of K , where K ranged from 1 to 5. K refers to the real developer being among the top- K in the recommended list. ACC_Topk can be formulated as follows:

$$ACC_Topk = \frac{number_K}{test_datasets'} \quad (13)$$

where $number_k$ represents the number of bug reports of the real developer in the top- K recommendation list, and $test_datasets'$ represents the number of all test bug reports.

5.4. Experimental Results

In this section, we discuss and analyze the experimental results based on specific experimental problems.

Equation (1) Compared with the baseline (unsupervised method), does the proposed BT-RL model improve the accuracy of bug triaging?

We selected some unsupervised methods, including LDA_KL, LDA_SVM, DERTOM, and DeepTriaging, for bug triaging. We aimed to prove whether BT-RL can achieve a better allocation effect on open-source datasets than the baseline method, and the accuracy rate of Top1–Top5 was used as the evaluation standard. We partitioned the entire dataset into train:test window sizes of 1000:50 for the experiments. Tables 3–5 show the accuracy of

BT-RL and other methods in bug triaging. Our comparative experiments refer to the data of the original paper, and the reference list provides the references.

Table 3. The accuracy of BT-RL and other benchmarks on four projects of the first experimental window.

| Project | Top-k | LDA_KL | LDA_SVM | DERTOM | DeepTriaging | BT-RL |
|------------|-------|--------|---------|--------|--------------|-------|
| NetBeans | Top5 | 0.28 | 0.48 | 0.48 | 0.33 | 0.54 |
| | Top4 | 0.26 | 0.48 | 0.44 | 0.33 | 0.52 |
| | Top3 | 0.20 | 0.46 | 0.36 | 0.33 | 0.48 |
| | Top2 | 0.18 | 0.40 | 0.32 | 0.18 | 0.44 |
| | Top1 | 0.10 | 0.26 | 0.10 | 0.04 | 0.34 |
| Mozilla | Top5 | 0.46 | 0.62 | 0.54 | 0.52 | 0.68 |
| | Top4 | 0.42 | 0.60 | 0.52 | 0.52 | 0.66 |
| | Top3 | 0.36 | 0.60 | 0.46 | 0.48 | 0.64 |
| | Top2 | 0.32 | 0.50 | 0.40 | 0.38 | 0.52 |
| | Top1 | 0.16 | 0.20 | 0.16 | 0.20 | 0.28 |
| Eclipse | Top5 | 0.58 | 0.74 | 0.72 | 0.47 | 0.78 |
| | Top4 | 0.50 | 0.70 | 0.62 | 0.37 | 0.72 |
| | Top3 | 0.42 | 0.66 | 0.58 | 0.27 | 0.68 |
| | Top2 | 0.36 | 0.52 | 0.52 | 0.18 | 0.58 |
| | Top1 | 0.16 | 0.40 | 0.32 | 0.12 | 0.46 |
| OpenOffice | Top5 | 0.44 | 0.50 | 0.46 | 0.20 | 0.52 |
| | Top4 | 0.40 | 0.44 | 0.38 | 0.20 | 0.46 |
| | Top3 | 0.32 | 0.42 | 0.36 | 0.20 | 0.42 |
| | Top2 | 0.22 | 0.30 | 0.28 | 0.16 | 0.30 |
| | Top1 | 0.16 | 0.26 | 0.20 | 0.10 | 0.18 |

Table 4. The accuracy of BT-RL and other benchmarks on four projects of the second experimental window.

| Project | Top-k | LDA_KL | LDA_SVM | DERTOM | DeepTriaging | BT-RL |
|------------|-------|--------|---------|--------|--------------|-------|
| NetBeans | Top5 | 0.48 | 0.70 | 0.62 | 0.44 | 0.76 |
| | Top4 | 0.36 | 0.60 | 0.60 | 0.35 | 0.72 |
| | Top3 | 0.28 | 0.58 | 0.54 | 0.35 | 0.64 |
| | Top2 | 0.20 | 0.48 | 0.44 | 0.31 | 0.54 |
| | Top1 | 0.10 | 0.36 | 0.28 | 0.21 | 0.40 |
| Mozilla | Top5 | 0.36 | 0.52 | 0.52 | 0.38 | 0.52 |
| | Top4 | 0.32 | 0.48 | 0.46 | 0.33 | 0.50 |
| | Top3 | 0.26 | 0.38 | 0.42 | 0.33 | 0.44 |
| | Top2 | 0.18 | 0.32 | 0.34 | 0.29 | 0.32 |
| | Top1 | 0.06 | 0.12 | 0.18 | 0.13 | 0.22 |
| Eclipse | Top5 | 0.42 | 0.52 | 0.44 | 0.40 | 0.64 |
| | Top4 | 0.38 | 0.46 | 0.42 | 0.23 | 0.62 |
| | Top3 | 0.24 | 0.44 | 0.42 | 0.19 | 0.56 |
| | Top2 | 0.18 | 0.28 | 0.36 | 0.13 | 0.42 |
| | Top1 | 0.08 | 0.22 | 0.26 | 0.08 | 0.26 |
| OpenOffice | Top5 | 0.40 | 0.46 | 0.42 | 0.15 | 0.48 |
| | Top4 | 0.40 | 0.38 | 0.40 | 0.08 | 0.40 |
| | Top3 | 0.32 | 0.32 | 0.32 | 0.04 | 0.38 |
| | Top2 | 0.22 | 0.28 | 0.30 | 0.04 | 0.28 |
| | Top1 | 0.16 | 0.14 | 0.16 | 0.04 | 0.10 |

Table 5. The accuracy of BT-RL and other benchmarks on four projects of the third experimental window.

| Project | Top-k | LDA_KL | LDA_SVM | DERTOM | DeepTriaging | BT-RL |
|------------|-------|--------|---------|--------|--------------|-------|
| NetBeans | Top5 | 0.50 | 0.52 | 0.48 | 0.31 | 0.54 |
| | Top4 | 0.46 | 0.46 | 0.44 | 0.31 | 0.48 |
| | Top3 | 0.40 | 0.44 | 0.40 | 0.27 | 0.48 |
| | Top2 | 0.36 | 0.40 | 0.38 | 0.25 | 0.46 |
| | Top1 | 0.20 | 0.34 | 0.34 | 0.23 | 0.38 |
| Mozilla | Top5 | 0.46 | 0.48 | 0.30 | 0.10 | 0.50 |
| | Top4 | 0.36 | 0.38 | 0.24 | 0.10 | 0.44 |
| | Top3 | 0.32 | 0.36 | 0.20 | 0.10 | 0.36 |
| | Top2 | 0.28 | 0.22 | 0.16 | 0.06 | 0.30 |
| | Top1 | 0.14 | 0.10 | 0.04 | 0.04 | 0.22 |
| Eclipse | Top5 | 0.50 | 0.50 | 0.48 | 0.40 | 0.58 |
| | Top4 | 0.40 | 0.50 | 0.42 | 0.35 | 0.54 |
| | Top3 | 0.32 | 0.40 | 0.38 | 0.25 | 0.44 |
| | Top2 | 0.22 | 0.36 | 0.34 | 0.23 | 0.36 |
| | Top1 | 0.12 | 0.32 | 0.28 | 0.21 | 0.22 |
| OpenOffice | Top5 | 0.34 | 0.46 | 0.38 | 0.27 | 0.50 |
| | Top4 | 0.32 | 0.40 | 0.36 | 0.23 | 0.40 |
| | Top3 | 0.28 | 0.34 | 0.30 | 0.23 | 0.40 |
| | Top2 | 0.22 | 0.30 | 0.22 | 0.02 | 0.34 |
| | Top1 | 0.06 | 0.22 | 0.20 | 0.02 | 0.32 |

As shown in Tables 3–5, the BT-RL method showed more significant improvements in the accuracy of Top1–Top5 compared with the LDA_KL, LDA_SVM, DERTOM, and DeepTriaging methods on the four datasets. We conducted experiments on bug triaging in different time periods. Compared with other benchmark methods, BT-RL yielded better experimental results, proving its reliability. First, because all baselines were trained based on topic models, they were used to identify the hidden topic information in the document set or corpus. By adopting the bag-of-words method, the documents were converted into word frequency vectors, but the relationship between words was not considered. We used Word2vec for text vectorization, which not only solved the problem of dimensionality disaster, but also mined the associated attributes between words, thereby improving the semantic representation of vectors. Second, regarding vector fusion, other benchmark methods used stitching; however, we used multiplication between elements for a more accurate information representation, which will be advantageous in the follow-up processes. Third, as the baselines were based on partial data training and partial data verification, and when the two data parts did not overlap, the effect was poor. However, for a new task, BT-RL was learning as it was being trained, and this mechanism enabled BT-RL to fully learn the information in the data.

Equation (2) Validation considering effects of multiple sources of information.

We regarded different factors as the input of the model, such as only the text information of the bug report or the activity information of the developer, or both. We wished to investigate whether the factors above affected the accuracy of the BT-RL method in the bug dispatch of open-source data (NetBeans, OpenOffice, Mozilla, and Eclipse) projects. Hence, we designed three groups of comparative experiments: (1) only consider text information (BT-RL-T), (2) only consider developer activity information (BT-RL-D), and (3) consider both (1) and (2) (BT-RL-TD). Table 6 shows the effect of considering different data as inputs on the results of BT-RL.

Table 6. The accuracy of BT-RL considering different information.

| Project | Top-k | BT-RL-T | BT-RL-D | BT-RL-TD |
|------------|-------|---------|---------|----------|
| NetBeans | Top5 | 0.40 | 0.54 | 0.54 |
| | Top4 | 0.32 | 0.50 | 0.52 |
| | Top3 | 0.32 | 0.46 | 0.48 |
| | Top2 | 0.28 | 0.44 | 0.44 |
| | Top1 | 0.26 | 0.32 | 0.34 |
| Mozilla | Top5 | 0.52 | 0.64 | 0.68 |
| | Top4 | 0.50 | 0.64 | 0.66 |
| | Top3 | 0.48 | 0.62 | 0.64 |
| | Top2 | 0.44 | 0.52 | 0.52 |
| | Top1 | 0.20 | 0.38 | 0.28 |
| Eclipse | Top5 | 0.42 | 0.50 | 0.78 |
| | Top4 | 0.38 | 0.50 | 0.72 |
| | Top3 | 0.32 | 0.50 | 0.68 |
| | Top2 | 0.26 | 0.46 | 0.58 |
| | Top1 | 0.22 | 0.26 | 0.46 |
| OpenOffice | Top5 | 0.36 | 0.48 | 0.52 |
| | Top4 | 0.34 | 0.42 | 0.46 |
| | Top3 | 0.28 | 0.38 | 0.42 |
| | Top2 | 0.22 | 0.24 | 0.30 |
| | Top1 | 0.14 | 0.20 | 0.18 |

Table 6 shows that BT-RL-TD yielded significant improvement in terms of the accuracy of Top1–Top5 compared with BT-RL-T and BT-RL-D on the four datasets. The accuracy of Top5 of BT-RL-TD increased by 35.00% compared with that of BT-RL-T, whereas it was the same as that of BT-RL-D for the NetBeans dataset. The accuracy of Top5 of BT-RL-TD increased by 30.76% and 6.25% compared with those of BT-RL-T and BT-RL-D, respectively, for the Mozilla dataset. The accuracy of Top5 of BT-RL-TD increased by 85.71% and 56.00% compared with those of BT-RL-T and BT-RL-D, respectively, for the Eclipse dataset. The accuracy of Top5 of BT-RL-TD increased by 44.44% and 8.33% compared with those of BT-RL-T and BT-RL-D, respectively, for the OpenOffice dataset. Therefore, it was effective to consider the multiple information sources of bug reports.

Equation (3) Verify effects of different feature fusion methods.

We wish to understand whether the use of different fusion methods for text and developer engagement features to assign bug reports in BT-RL will affect the accuracy. Hence, we designed three experiments to verify the effects of different feature fusion methods on BT-RL: (1) use BT-RL-Con, (2) use BT-RL-Add, and (3) use BT-RL-Mul.

The premise of adding and multiplying elements is to ensure that the two fused vector dimensions are the same. If the dimensions are different, they can be transformed into the same dimension vector through linear transformation. Table 7 shows the effects of different data fusion approaches on the BT-RL results.

Table 7 shows that BT-RL-Mul yielded significant improvement in terms of the accuracy of Top1–Top5 compared with BT-RL-Add and BT-RL-Con for the four datasets. The accuracy of Top5 of BT-RL-Mul increased by 8.00% and 17.39% compared with those of BT-RL-Add and BT-RL-Con, respectively, for the NetBeans dataset. The accuracy of Top5 of BT-RL-Mul increased by 21.42% and 36.00% compared with those of BT-RL-Add and BT-RL-Con, respectively, for the Mozilla dataset. The accuracy of Top5 of BT-RL-Mul increased by 30.00% and 39.28% compared with those of BT-RL-Add and BT-RL-Con, respectively, for the Eclipse dataset. The accuracy of Top5 of BT-RL-Mul increased by 13.04% and 18.18% compared with those of BT-RL-Add and BT-RL-Con, respectively, for the OpenOffice dataset. Addition and multiplication between elements increased the amount of information in each dimension, which was performed intentionally for the final distribution.

Table 7. The accuracy of BT-RL using different data fusion methods.

| Project | Top-k | BT-RL-T | BT-RL-D | BT-RL-TD |
|------------|-------|---------|---------|----------|
| NetBeans | Top5 | 0.50 | 0.46 | 0.54 |
| | Top4 | 0.46 | 0.42 | 0.52 |
| | Top3 | 0.44 | 0.40 | 0.48 |
| | Top2 | 0.42 | 0.30 | 0.44 |
| | Top1 | 0.24 | 0.26 | 0.34 |
| Mozilla | Top5 | 0.56 | 0.50 | 0.68 |
| | Top4 | 0.50 | 0.48 | 0.66 |
| | Top3 | 0.40 | 0.38 | 0.64 |
| | Top2 | 0.38 | 0.30 | 0.52 |
| | Top1 | 0.38 | 0.30 | 0.28 |
| Eclipse | Top5 | 0.60 | 0.56 | 0.78 |
| | Top4 | 0.60 | 0.56 | 0.72 |
| | Top3 | 0.56 | 0.50 | 0.68 |
| | Top2 | 0.44 | 0.38 | 0.58 |
| | Top1 | 0.28 | 0.28 | 0.46 |
| OpenOffice | Top5 | 0.46 | 0.44 | 0.52 |
| | Top4 | 0.40 | 0.40 | 0.46 |
| | Top3 | 0.32 | 0.28 | 0.42 |
| | Top2 | 0.32 | 0.28 | 0.30 |
| | Top1 | 0.24 | 0.22 | 0.18 |

The experimental results of different windows obtained in the NetBeans, Mozilla, Eclipse, and OpenOffice datasets are shown in Tables 3–5, respectively. The research questions posed herein were answered based on the obtained results. In most cases, the accuracy of the Top5 developers' sequence recommended by our method was higher than that of the baseline.

6. Threats to Validity

6.1. Internal Validity

Threats to internal validity included the accuracies of BT-RL and the rebuilt models (LDA_KL, LDA_SVM, DERTOM, and DeepTriaging). We verified the accuracy of our code in reducing errors in BT-RL. In addition, we rebuilt LDA_KL, LDA_SVM, DERTOM, and DeepTriaging with reference to previous studies, and we verified their accuracies.

6.2. External Validity

The threat to the external validity of BT-RL was that we only applied our experiments on NetBeans, OpenOffice, Mozilla, and Eclipse projects, which might not be representative of all bug reports. However, these datasets are widely used, and BT-RL demonstrates favorable bug triaging performance on top-K accuracy. In addition, although we preprocessed the dataset for imbalance, noise still appeared in the dataset, although we did not observe it. In our future studies, we plan to propose better strategies to reduce noise in the NetBeans, OpenOffice, Mozilla, and Eclipse datasets. In addition, we plan to collect new datasets with different domains of the projects. Subsequently, we plan to extend BT-RL to other bug report projects to verify the generalizability of BT-RL.

6.3. Construct Validity

One threat to validity is that we assessed whether real developers existed in the top-K of the model recommendation list as the criteria for reward and punishment. To design a reasonable reward method to measure the value of the action correctly in a DQN is difficult, and it is a problem that requires further research. In our future studies, we will design a more stable reward method to guide the agent to suggest accurate developer recommendations based on bug-related information.

Another threat to the validity of our method is the parameter setting. Although we did not perform specific experimental verification of the network parameters involved in the experiment, our parameter design was based on the experience of other researchers.

7. Conclusions and Future Works

Herein, we proposed an effective BT-RL algorithm model to address the bug triaging problem using the RL framework. This model can assign bug reports that are newly generated from software development in the online form. The model comprises an ODM model and a DMSF fusion model. The ODM model selects high-quality sample data for the DMSF fusion model. The DMSF fusion model obtains the feature extracted from the information and predicts the appropriate developer, as well as provides rewards as a weak signal to supervise the dynamic matching process. Extensive experiments demonstrated that our model performed better than state-of-the-art methods.

In future work, we will further investigate the method to solve the bug triaging problem. Specifically, we plan to collect more datasets to analyze the influence of underlining factors on the performance of bug triaging. In addition, we will further investigate more measurements for the validation of the performance.

Author Contributions: Formal analysis, J.Z.; Methodology, H.L.; Software, X.Q.; Validation, X.G.; Writing—original draft, Y.L.; Writing—review & editing, J.A. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (No. 61902050), the Natural Science Foundation of Liaoning Province (No. 2021-MS-136), and the Fundamental Research Funds for the Central Universities (No. 3132019355).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: We obtained all datasets in GitHub (<https://github.com/zjl95/DRL-bug-traige>, accessed on 17 February 2022).

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

References

1. Defect Statistics. Apache OpenOffice. Available online: <https://www.openoffice.org/stats/defects.html> (accessed on 25 February 2022).
2. Deng, W.; Zhang, X.X.; Zhou, Y.Q.; Liu, Y.; Zhou, X.B.; Chen, H.L.; Zhao, H.M. An enhanced fast non-dominated solution sorting genetic algorithm for multi-objective problems. *Inform. Sci.* **2022**, *585*, 441–453. [[CrossRef](#)]
3. Ran, X.; Zhou, X.; Lei, M.; Tepsan, W.; Deng, W. A novel k-means clustering algorithm with a noise algorithm for capturing urban hotspots. *Appl. Sci.* **2021**, *11*, 11202. [[CrossRef](#)]
4. Li, G.; Li, Y.; Chen, H.; Deng, W. Fractional-Order Controller for Course-Keeping of Underactuated Surface Vessels Based on Frequency Domain Specification and Improved Particle Swarm Optimization Algorithm. *Appl. Sci.* **2022**, *12*, 3139. [[CrossRef](#)]
5. Wang, X.; Wang, H.Y.; Du, C.Z.; Fan, X.; Cui, L.; Chen, H.; Deng, F.; Tong, Q.; He, M.; Yang, M.; et al. Custom-molded offloading footwear effectively prevents recurrence and amputation, and lowers mortality rates in high-risk diabetic foot patients: A multicenter, prospective observational study. *Diabetes Metab. Syndr. Obes. Targets Ther.* **2022**, *15*, 103–109.
6. Cui, H.; Guan, Y.; Chen, H. Rolling element fault diagnosis based on VMD and sensitivity MCKD. *IEEE Access* **2021**, *9*, 120297–120308.52. [[CrossRef](#)]
7. Deng, W.; Li, Z.; Li, X.; Chen, H.; Zhao, H. Compound fault diagnosis using optimized MCKD and sparse representation for rolling bearings. *IEEE Trans. Instrum. Meas.* **2022**, *71*, 3508509. [[CrossRef](#)]
8. Anvik, J.; Hiew, L.; Murphy, G.C. Coping with an open bug repository. In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, San Diego, CA, USA, 16–17 October 2005; pp. 35–39.
9. Gu, Z.; Barr, E.T.; Hamilton, D.J.; Su, Z. Has the bug really been fixed? In Proceedings of the 32th ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, 2–8 May 2010; pp. 55–64.
10. Collofello, J.S.; Woodfield, S.N. Evaluating the effectiveness of reliability-assurance techniques. *J. Syst. Softw.* **1989**, *9*, 191–195. [[CrossRef](#)]

11. Wu, W.; Zhang, W.; Yang, Y.; Wang, Q. Time series analysis for bug number prediction. In Proceedings of the 2th International Software Engineering and Data Mining, Chengdu, China, 23–25 June 2010; pp. 589–596.
12. Kawakami, K. Supervised Sequence Labelling with Recurrent Neural Networks. Ph.D. Thesis, Technical University of Munich, München, Germany, 2008.
13. Tamrawi, A.; Nguyen, T.T.; Al-Kofahi, J.M.; Nguyen, T.N. Fuzzy set and cache-based approach for bug triaging. In Proceedings of the 19th ACM and the 13th on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; pp. 365–375.
14. Naguib, H.; Narayan, N.; Brgge, B.; Helal, D. Bug report assignee recommendation using activity profiles. In Proceedings of the 10th Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 22–30.
15. Nigam, K.; Mccallum, A.K.; Thrun, S.; Mitchell, T. Text classification from labeled and unlabeled documents using EM. *Mach. Learn.* **2000**, *39*, 103–134. [\[CrossRef\]](#)
16. Zhang, W.; Wang, S.; Wang, Q. KSAP: An approach to bug report triaging using KNN search and heterogeneous proximity. *Inf. Softw. Technol.* **2016**, *70*, 68–84. [\[CrossRef\]](#)
17. Anvik, J.; Lyndon, H.; Murphy, G.C. Who should fix this bug? In Proceedings of the 28th International Software Engineering, Shanghai, China, 20–28 May 2006; pp. 361–370.
18. Syed, N.A.; Javed, F.; Franz, W. Automatic Software Bug Triaging System (BTS) Based on Latent Semantic Indexing and Support Vector Machine. In Proceedings of the 2009 Fourth International Conference on Software Engineering Advances, Porto, Portugal, 20–25 September 2009; pp. 216–221.
19. Xuan, J.; Jiang, H.; Ren, Z.; Yan, J.; Luo, Z. Automatic Bug Triaging using Semi-Supervised Text Classification. In Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010), Redwood City, San Francisco Bay, CA, USA, 1–3 July 2010; pp. 209–214.
20. Yang, G.; Zhang, T.; Lee, B. Towards semi-automatic bug triaging and severity prediction based on topic model and mul-ti-feature of bug reports. In Proceedings of the 38th Annual Computer Software and Applications Conference, Vasteras, Sweden, 21–25 July 2014; pp. 97–106.
21. Wu, W.; Zhang, W.; Yang, Y.; Wang, Q. Drex: Developer recommendation with k-nearest neighbor search and expertise ranking. In Proceedings of the 18th Asia-Pacific Software Engineering Conference, Ho Chi Minh City, Vietnam, 5–8 December 2011; pp. 389–396.
22. Davor, C.; Murphy, G.C. Automatic bug triaging using text categorization. In Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, Banff, AB, Canada, 20–24 June 2004; pp. 92–97.
23. Xie, X.; Zhang, W.; Yang, Y.; Wang, Q. DRETOM: Developer recommendation based on topic models for bug resolution. In Proceedings of the 8th International Conference on Predictive Models in Software Engineering, Lund, Sweden, 21–22 September 2012; pp. 19–28.
24. Nguyen, T.T.; Nguyen, A.T.; Nguyen, T.N. Topic-based, time-aware bug assignment. *ACM SIGSOFT Softw. Eng. Notes* **2014**, *39*, 1–4. [\[CrossRef\]](#)
25. Zhang, T.; Yang, G.; Lee, B.; Lua, E.K. A novel developer ranking algorithm for automatic bug triaging using topic model and de-veloper relations. In Proceedings of the 21st Asia-Pacific Software Engineering Conference, Jeju, Korea, 1–4 December 2014; Volume 1, pp. 223–230.
26. Yan, M.; Zhang, X.; Yang, D.; Xu, L.; Kymer, J.D. A component recommender for bug reports using discrim-inative probability latent semantic analysis. *Inf. Softw. Technol.* **2016**, *73*, 37–51. [\[CrossRef\]](#)
27. Xuan, J.; Jiang, H.; Zhang, H.; Ren, Z. Developer recommendation on bug commenting: A ranking approach for the developer crowd. *Sci. China Inf. Sci.* **2017**, *60*, 072105. [\[CrossRef\]](#)
28. Somasundaram, K.; Murphy, G.C. Automatic categorization of bug reports using latent dirichlet allocation. In Proceedings of the 5th India Software Engineering Conference, Kanpur, India, 22–25 February 2012; pp. 125–130.
29. Yin, Y.; Dong, X.; Xu, T. Rapid and Efficient Bug Triaging Using ELM for IOT Software. *IEEE Access* **2018**, *6*, 52713–52724. [\[CrossRef\]](#)
30. Khatun, A.; Sakib, K. A Bug Triaging Approach Combining Expertise and Recency of Both Bug Fixing and Source Commits. In Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, Funchal, Portugal, 23–24 March 2018; pp. 351–358. [\[CrossRef\]](#)
31. Yadav, A.; Singh, S.K. A novel and improved developer rank algorithm for bug assignment. *Int. J. Intell. Syst. Technol. Appl.* **2020**, *19*, 78–101.
32. Jahanshahi, H.; Chhabra, K.; Cevik, M.; Basar, A. DABT: A Dependency-aware Bug Triaging Method. In *Evaluation and Assessment in Software Engineering*; ACM: New York, NY, USA, 2021; pp. 221–230.
33. Su, Y.; Xing, Z.; Peng, X.; Xia, X.; Wang, C.; Xu, X.; Zhu, L. Reducing Bug Triaging Confusion by Learning from Mistakes with a Bug Tossing Knowledge Graph. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021; pp. 191–202.
34. Wu, H.; Ma, Y.; Xiang, Z.; Yang, C.; He, K. A Spatial-Temporal Graph Neural Network Framework for Automated Software Bug Triaging. *Knowl.-Based Syst.* **2022**, *241*, 108308. [\[CrossRef\]](#)
35. Guo, S.; Zhang, X.; Yang, X.; Chen, R.; Guo, C.; Li, H.; Li, T. Developer activity motivated bug triaging: Via convolutional neural network. *Neural Processing Lett.* **2020**, *51*, 2589–2606. [\[CrossRef\]](#)

-
36. Kashiwa, Y.; Ohira, M. A Release-Aware Bug Triaging Method Considering Developers' Bug-Fixing Loads. *IEICE TRANSACTIONS Inf. Syst.* **2020**, *103*, 348–362. [[CrossRef](#)]
 37. Mani, S.; Sankaran, A.; Aralikkatte, R. Deeptriaging: 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, Kolkata, India, 3–5 January 2019; pp. 171–179.