Xi SQ, Yao Y, Xiao XS *et al.* Bug triaging based on tossing sequence modeling. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(5): 942–956 Sept. 2019. DOI 10.1007/s11390-019-1953-5

Bug Triaging Based on Tossing Sequence Modeling

Sheng-Qu Xi[1], Yuan Yao[1,*], *Member*, *CCF*, Xu-Sheng Xiao[2], *Member*, *ACM*, *IEEE*, Feng Xu[1], *Member*, *CCF* and Jian Lv[1], *Fellow*, *CCF*

[1] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*

[2] *Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland Ohio 44106-7071, U.S.A.*

E-mail: xsq@smail.nju.edu.cn; y.yao@nju.edu.cn; xusheng.xiao@case.edu; {xf, lj}@nju.edu.cn

Received March 1, 2019; revised August 5, 2019.

Abstract Bug triaging, which routes the bug reports to potential fixers, is an integral step in software development and maintenance. To make bug triaging more efficient, many researchers propose to adopt machine learning and information retrieval techniques to identify some suitable fixers for a given bug report. However, none of the existing proposals simultaneously take into account the following three aspects that matter for the efficiency of bug triaging: 1) the textual content in the bug reports, 2) the metadata in the bug reports, and 3) the tossing sequence of the bug reports. To simultaneously make use of the above three aspects, we propose iTriage which first adopts a sequence-to-sequence model to jointly learn the features of textual content and tossing sequence, and then uses a classification model to integrate the features from textual content, metadata, and tossing sequence. Evaluation results on three different open-source projects show that the proposed approach has significantly improved the accuracy of bug triaging compared with the state-of-the-art approaches.

Keywords bug triaging, tossing sequence, software repository mining

1 Introduction

Software defects, i.e., bugs, appear during software development and maintenance, and fixing bugs is a time-consuming and costly task. To better manage bugs and bug fixing histories, bug tracking systems such as Bugzilla are widely used[1]. When a new bug report is submitted or reported, the manager (a senior developer or the project leader) will choose a developer to fix the bug described in the report. During the bug fixing process, developers and reporters will communicate with each other through comments, and they can upload additional resources such as screenshots to support bug fixing. Furthermore, the developer can reassign the bug report to other developers[2]. Such reassignment is called bug tossing. Common reasons for bug tossing include that bugs are assigned to developers by mistake, or the developer wants to include other developers with additional expertise to fix the bug. After the bug is fixed, the status of the bug report will be changed.

Manually inspecting and assigning bug reports is tedious and time-consuming, especially in those software projects that have a large amount of bug reports and developers. For example, the Eclipse and Mozilla projects receive several hundreds of bug reports per day and properly assigning each of them to one of the several thousands of developers would be very time-consuming[2]. To aid in finding the appropriate developers, automatic bug triaging approaches have been proposed[3−6]. These approaches can be roughly divided into two classes. The first class adopts information retrieval techniques by constructing the representations of bug reports and developers, and then matching the bug reports to the most related developers. The second class adopts machine learning techniques by extracting features or learning representations of bug reports and treating the developers as labels. For both classes of methods, the key idea is to extract the rep-

Regular Paper

Special Section on Software Systems 2019

A preliminary version of the paper was published in the Proceedings of Internetware 2018.

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61690204, 61672274, and 61702252, and the Collaborative Innovation Center of Novel Software Technology and Industrialization at Nanjing University.

*Corresponding Author

resentations of bug reports. For this purpose, some existing methods use the vector space model (VSM) to represent a bug report (i.e., a bug report is treated as a vector of terms and their frequencies). However, these representations are less accurate as developers often use various terms to express the same meaning. Later, topic modeling (e.g., Latent Semantic Indexing/Analysis and Latent Dirichlet Allocation) which infers the inherent latent topics of a textual document has been used as a way to learn the semantic meanings of bug reports[6,7]. More recently, deep learning techniques such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) have also been used to improve the representation learning[8,9]. Although topic modeling and deep learning can learn better representations of the textual content in bug reports, none of them simultaneously consider the features of metadata and tossing sequence. Meanwhile, these two features have been shown to be effective in terms of improving the triaging accuracy[2,5].

In this paper, we propose to make use of both textual content and metadata in the bug reports while simultaneously modeling the tossing sequence. We put special focus on the tossing sequence due to the following reasons. First, for some bug reports, the bug reporter himself/herself is one of the developers in the project, and he/she is likely to fix the corresponding bugs in the future. For example, based on our empirical study of the datasets, we find that there are near 25% of the bug reports that are fixed by their corresponding reporters. Therefore, the bug reporter should be encoded into the tossing sequence. Second, while existing approaches aim to suggest fixers for a new bug report, some bug reports may already be assigned or reassigned to a few developers (we name such bug reports as halfway bug reports). The existing tossing sequences may provide valuable information and thus may be helpful for the bug triaging task. For example, a reassignment in the tossing sequence indicates the relationships between the two developers, and identifying one of them may help us find the other suitable fixers.

To simultaneously make use of the above three aspects, we propose iTriage which contains two steps. In the first step, since both textual contents and tossing sequences can be seen as sequences, we use a sequence-to-sequence model to interactively learn the textual feature for the bug content and the routing feature for the tossing sequence. The sequence-to-sequence model mainly consists of two components: encoder and decoder. The encoder component takes the textual content of a bug report as input, and outputs the hidden state/feature for each word as well as the final representation of the content. We add the attention mechanism between the encoder and the decoder, and the hidden state for each word is used as the input of attention. The decoder component takes the final representation of the encoder as input and outputs the tossing sequence for the input bug report. In particular, the decoder component takes the reporter token as the initial token, and then fits all the developers in the corresponding tossing sequence. Through such a sequence-to-sequence model, the features of bug content and tossing sequence are jointly learned for the bug triaging task.

In the second step, we create a classification model to predict the final responsible fixer. The inputs of the classification model include the textual feature, the routing feature, and the metadata feature. For the textual feature and the routing feature, we use the outputs of the sequence-to-sequence model in the first step. Specially, the routing feature is generated by the current tossing sequence of the bug report (new or halfway). For new bug reports, the routing feature only depends on the reporter. For halfway bug reports, the routing feature depends on both the reporter and the already-tossed developers. For metadata features, we use a lookup table to map and update the representations of each metadata.

In summary, this paper mainly makes the following contributions.

- We propose a bug triaging approach iTriage which simultaneously models textual content, metadata, and the tossing sequence. All the three aspects matter for the accuracy of bug triaging. The proposed approach can handle both new and halfway bug reports. To the best of our knowledge, this is the first work that pays special attention on the triaging of halfway bug reports.
- We propose a novel way to learn the representations of the tossing sequence by considering the following two aspects: 1) the bug reporter could be the corresponding fixer, and 2) the tossing sequence for a bug report presents the dependencies and relationships among potential fixers.
- We conduct evaluations on three open-source projects with over 670 000 bug reports in total, and the results show that iTriage outperforms the existing machine learning and information retrieval based approaches in terms of identifying the right fixer. For example, the proposed iTriage improves the state-

of-the-art TopicMinerMTM[6] by 9.39% on average in terms of top-1 accuracy.

• We extend our previous work SeqTriage[10] in two aspects: we 1) modify the model structure (i.e., by treating SeqTriage as an intermediate step to learn features and applying a classification model to suggest fixers directly) to integrate tossing sequence properly and make iTriage applicable to halfway bug reports, and 2) add metadata into iTriage to increase the prediction accuracy.

The rest of this paper is organized as follows. Section 2 introduces some background information. Section 3 presents the proposed approach. Section 4 states the research questions and describes the evaluation setup, and Section 5 shows the evaluation results. Section 6 discusses the threats to validity. Section 7 reviews related work, and Section 8 concludes the paper.

2 Background and Preliminary Study

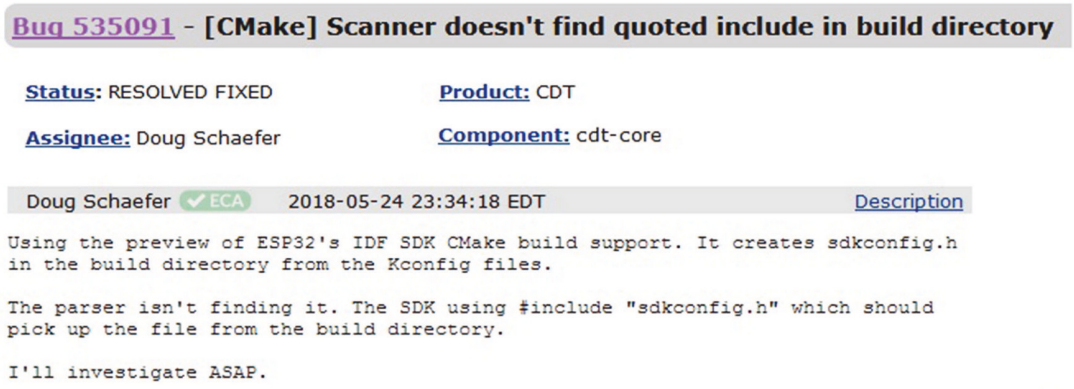
In this section, we present some background knowledge and a preliminary study about bug reports.

2.1 Bug Report

A sample bug report of the Eclipse project is shown in Fig.1. Fig.1(a) is the bug content which provides the information of the bug report. Fig.1(b) is the bug history which records the changes of this bug report.

The bug content of a bug report is widely used in automatic bug triaging. The title (also known as summary), and the description are the major types of textual information that are used by the existing approaches. In addition to the textual information, there are four types of metadata. The "Product" metadata and the "Component" metadata have been shown to be useful in bug report related studies[5,6,11]. The "Status" metadata identifies the resolution of bug reports, and the existing approaches mainly select the bug reports that are marked as fixed (referred to as "fixed bug reports") as their datasets. The "Assignee" metadata points out one developer who is responsible for fixing the bug report.

The bug history records the historical changes of the bug report. Each change is organized as a tuple of "who", "when", "what", "removed", and "added". The "what" element describes the name of the bug field that



(a)

Who	When	What	Removed	Added
dschaefer	2018-05-24 23:34:29 EDT	Status	NEW	ASSIGNED
		Assignee	cdt-core-inbox	dschaefer
genie	2018-05-29 00:14:04 EDT	See Also		https://git.eclipse.org/r/123499
genie	2018-05-29 01:50:39 EDT	See Also		https://git.eclipse.org/c/cdt/ ...
dschaefer	2018-05-29 12:08:54 EDT	Status	ASSIGNED	RESOLVED
		Resolution	---	FIXED
dschaefer	2018-05-29 14:12:21 EDT	Target Milestone	---	9.5.0

(b)

Fig.1. Example bug report in Eclipse Bugzilla (bug ID: 535091).

has been changed, with the previous value in "removed" and the new value in "added". For example, in the second row of the bug history in Fig.1, the assignee is changed from "cdt-core-inbox" to "dschaefer". Then, in the fifth and the sixth rows, "dschaefer" took five days to fix the bug, and changed the "Status" metadata to "RESOLVED FIXED". In this example, the bug report is routed from "dschaefer" to himself/herself, and no further tossing is performed. In other bug reports, the assignee may be changed from one developer to another, and each of such changes is a step of tossing. Additionally, one bug report could be tossed by several developers until the last developer fixes the bug. We refer to the last developer as the real fixer.

2.2 Preliminary Study

To investigate how the characteristics of bug reports can be used for better bug triaging, we conduct a preliminary study on three projects, i.e., Eclipse①, Mozilla②, and Gentoo③. As shown in Fig.1, we can see that the "Assignee" metadata is different from that in the bug history through the raw HTML file (i.e., "Doug Schaefer" vs "dschaefer"). Apparently, these two names stand for the same person in this case. To make it easier to match the same person, we use REST APIs④ to crawl bug reports and change histories from Bugzilla. In the REST APIs, each user is described by their ID, login name (e.g., dschaefer), and real name (e.g., Doug Schaefer). We can easily match the login name with the real name to identify the same person. The basic information of these datasets is presented in Table 1. We next present the results and the important findings of our preliminary study.

Table 1. Raw Dataset Information

Name	Period	Number of Reports
Eclipse	2008.01–2017.03	210 487
Mozilla	2008.01–2017.02	300 165
Gentoo	2009.01–2017.11	165 483

Reporter Locality. We conjecture that reporters tend to be focused on certain sub-areas of the project. Thus, we first study the locality of bug reporters (i.e., their focused sub-areas) in the collected datasets. The datasets we collected contain various sub-areas. For example, Eclipse has the product CDT to support C/C++ development, and JDT to support Java development. CDT and JDT are both based on PDE (plug-in development environment) but they are independent with each other. One developer could be response to develop one of them, and one end-user might only use either CDT or JDT based on his/her favorite language.

As a result, we may assume that the reporters only focus on several sub-areas of the project. To evaluate this assumption, we combine the "Product" and the "Component" metadata as a pair, and compute the number of product-component pairs for each reporter who has submitted reports. The results are shown in Table 2. As we can see, most reporters only focus on 1 or 2 sub-areas (i.e., product-component pairs). Only 7% of reporters have submitted bug reports to more than five product-component pairs. This gives us confidence that a reporter could be helpful in automatic bug triaging, as the fixer may be interested in the same product or component.

Table 2. Reporter Locality in Eclipse

Number of Sub-Areas	Count	Percentage (%)
1	11 841	68.49
2	2 327	13.46
3–5	1 829	10.58
> 5	1 290	7.46

Tossing Sequence. Tossing a bug report is time-consuming and each tossing step takes more than 10 days regularly[2]. Therefore, reducing the tossing sequence length can greatly reduce the bug fixing time. To understand the costs of tossing sequences, we next measure the average length of the tossing sequences. We use the number of developers that are involved in a tossing sequence as its length. The results are shown in Table 3. As we can see, over 40% of the bug reports could be fixed in the first assignment (either by the reporter self or by another developer), and there are near 60% of the bug reports that need further tossing. This indicates if we can directly route the bug report to the final developer in the sequence based on the existing tossing sequences to save time.

①https://bugs.eclipse.org/bugs, July 2019.
②https://bugzilla.mozilla.org, July 2019.
③https://bugs.gentoo.org, July 2019.
④https://wiki.mozilla.org/Bugzilla:REST_API, July 2019.

Table 3. Tossing Length in Eclipse

Tossing Length	Count	Percentage (%)
1	87 339	41.49
2	72 681	34.53
3–5	37 516	17.82
> 5	12 951	6.15

3 Proposed Approach

In this section, we present the proposed approach iTriage for bug triaging.

We start with an overview of the proposed approach in Fig.2. We first model the textual contents and tossing sequences, extracted from the bug repository (i.e., datasets), via a sequence-to-sequence model to interactively learn the textual features and the routing features. We name this model as feature learning model in the figure. Next, we integrate the textual feature, the routing feature, and the metadata feature of a bug report into a classifier to recommend developers. We name it as the fixer suggestion model.

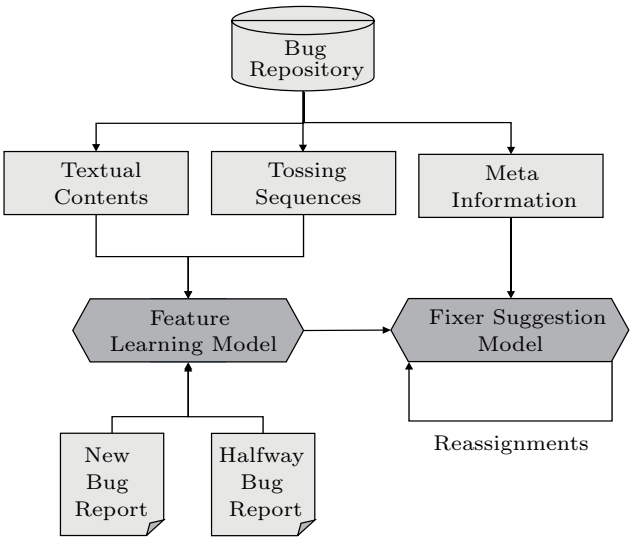


Fig.2. iTriage framework.

3.1 Feature Learning Model

We use a sequence-to-sequence model to learn the interactive features for bug reports and tossing sequences (including the reporter). The overview of this part is shown in Fig.3. It has two components: encoder and decoder. The encoder component is used to generate the textual representations based on the input bug reports. The decoder component is used to simulate the bug tossing process. The general idea is to learn the tossing sequence based on the content of bug reports. Furthermore, the attention mechanism is applied between the encoder and the decoder components. In the following, we first describe the encoder component and the decoder component, and then move on to the attention mechanism.

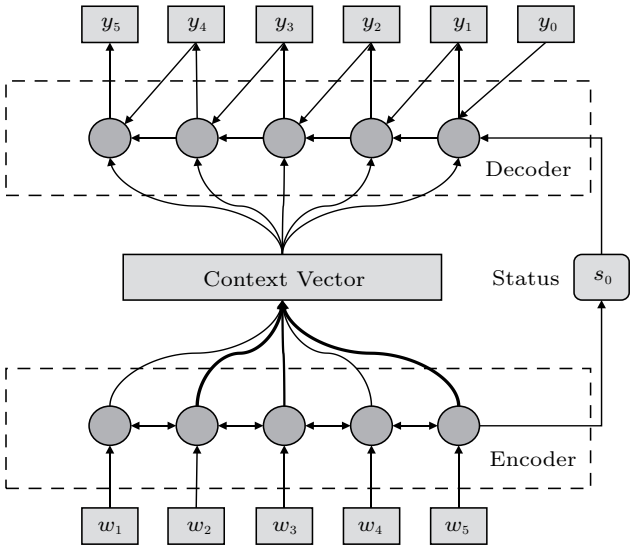


Fig.3. Feature learning model.

3.1.1 Encoder Component

As mentioned in the introduction, the key of bug triaging approaches is to extract the representations of bug reports. In our encoder component, we aim to learn the overall representation for the bug report as well as the presentation for each single word in the bug content. Formally, given a bug report with n words $\{w_1, w_2, \ldots, w_n\}$, the encoder component generates the overall representation s_0 for these n words and the representations $\{h_1, h_2, \ldots, h_n\}$ for each word (i.e., h_i is the presentation for word w_i).

To generate these representations, a recent trend is to use RNNs with neurons LSTM or GRU as they have been successfully applied in various sequence modeling tasks (e.g., language translation[12], speech recognition[13], and textual classification[14,15]). In this work, we use bidirectional RNNs as a way to generate textual representations, and we use the GRU neuron as it has less parameters. The structure of GRU is listed as follows[16],

$$\begin{aligned}
\boldsymbol{z}_i &= \sigma(\boldsymbol{W}_z \cdot \boldsymbol{x}_i + \boldsymbol{U}_z \cdot \boldsymbol{h}_{i-1}),\\
\boldsymbol{r}_i &= \sigma(\boldsymbol{W}_r \cdot \boldsymbol{x}_i + \boldsymbol{U}_r \cdot \boldsymbol{h}_{i-1}),\\
\boldsymbol{u}_i &= \tanh(\boldsymbol{W} \cdot \boldsymbol{x}_i + \boldsymbol{U} \cdot (\boldsymbol{r}_i \circ \boldsymbol{h}_{i-1})),\\
\boldsymbol{h}_i &= (1 - \boldsymbol{z}_i) \circ \boldsymbol{h}_{i-1} + \boldsymbol{z}_i \circ \boldsymbol{u}_i,
\end{aligned}$$

where $\circ$ indicates the element-wise product, σ is the sigmoid activation function, x_i is the current input of the GRU neuron (obtained from word w_i), and h_i is the current hidden state (the representation of word w_i). $\boldsymbol{z}_i$ and $\boldsymbol{r}_i$ are the update gate vector and the reset gate vector defined by GRU, respectively. Note that s_0 equals h_n in the GRU neuron. In this work, we apply word embedding before feeding words into GRU, as word embedding can generate meaningful low dimension representations and has shown its effectiveness in many text related tasks.

There are two outputs from the encoder component. One is the textual representation for each word (the input of the context vector in Fig.3), and the other is the overall representation of the text (denoted as status in Fig.3).

3.1.2 Decoder Component

Intuitively, modeling the tossing sequences can help bug triaging by capturing the dependencies between developers. For this purpose, we use RNNs in the decoder component to model tossing sequences. The decoder component works as follows. For the first prediction target in the sequence (i.e., the first developer in the tossing sequence), there are three inputs. The first input is the initial state (i.e., status s_0 in Fig.3). The second one is the initial token as input (i.e., y_0 in Fig.3). Note that instead of using a $\langle start \rangle$ token as the initial state, we use the reporter as the first token. The third input is the context vector $\boldsymbol{c}$ which is computed based on the output of each GRU neuron. Next, for the other prediction targets, we use the previous state, the real developer in the previous prediction, and the context vector as input. By doing so, each prediction is closely related to the previous predictions. Specially, the context vectors are different for different prediction targets (see Subsection 3.1.3). Consider the example in Fig.3. The decoder component first takes the encoder status s_0 and the reporter token y_0 as input, computes the first context vector $\boldsymbol{c}_1$, and then predicts the first developer y_1. Next, the decoder component predicts the next developer y_2 based on context vector $\boldsymbol{c}_2$, the hidden state s_1 from the previous GRU neuron, and the previous token y_1. This process repeats until it predicts the special token $\langle end \rangle$ which indicates the end of sequence.

In the decoder component, we aggregate different context vectors for different prediction targets. To compute the context vector $\boldsymbol{c}$ for each prediction target, we resort to the attention mechanism (see Subsection 3.1.3). The basic idea is to pool the representations from each word. That is, let $A = \{a_1, a_2, \ldots, a_n\}$ be the attention weights for $\{h_1, h_2, \ldots, h_n\}$, and the j-th context vector c_j is computed as

$$\boldsymbol{c}_j = \sum_{i=1}^{n} a_i \cdot \boldsymbol{h}_i,$$

where a_i is a scalar and $\boldsymbol{h}_i$ is a vector.

3.1.3 Attention

We assume that not all words contribute equally to the representation of the text for a given prediction target. When the decoder generates a single prediction, we first define the score of word i as

$$e_i = \tanh(\boldsymbol{W}_e \cdot s_{j-1} + \boldsymbol{U}_e \cdot \boldsymbol{h}_i),$$

where we use $\boldsymbol{W}_e$ and $\boldsymbol{U}_e$ as the weight matrices, and s_{j-1} is the previous hidden state defined above. Then, the attention weight a_i for word i could be defined as follows.

$$a_i = \frac{\exp(e_i)}{\sum_{k=1}^{n} \exp(e_k)}.$$

3.2 Fixer Suggestion Model

In this fixer suggestion model, we treat the problem as a classification problem and use a classification model to predict the final fixer for a given bug report. Note that we can deal with both new bug reports and halfway bug reports. Specially, we treat each developer as a label and recommend the developer with the maximum softmax probability.

The structure of this model is shown in Fig.4. As we can see, the prediction of a bug report depends on three kinds of features, i.e., textual feature, routing feature, and metadata feature. For the previous two, they are directly obtained from the feature learning model. That is, we directly input the content of the given bug report into the encoder part of the feature learning model to obtain its textual feature. For the routing feature, if the bug report is a new bug report, we input the reporter in the decoder part of the feature learning model to obtain the routing feature; if the bug report is a halfway one, we input the bug reporter as well as the existing developers in the tossing sequence into the decoder to obtain the routing feature. For the metadata feature, we add an embedding layer. That is, we treat the metadata embedding as the metadata feature and look it up in an embedding matrix by the unique index of metadata.

Finally, we combine the three features by concatenating them into one vector, and then feed the vector into a fully-connected layer before the softmax layer. The softmax layer serves to predict the probability distribution over all the developers for the given bug report. We choose the one with the highest probability as the suggested fixer.

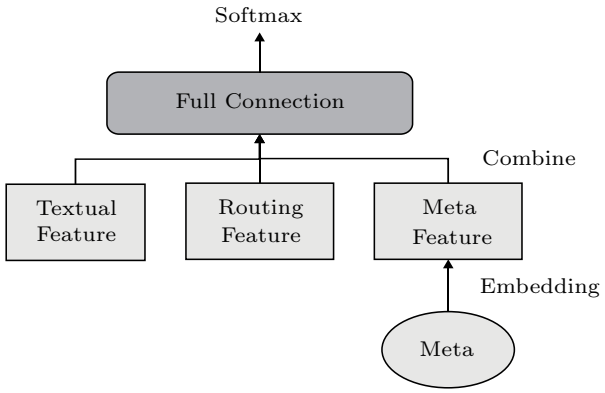


Fig.4. Fixer suggestion model.

4 Research Questions

In this section, we present the research questions.

4.1 Research Questions

We seek to answer the following four research questions in this paper.

*RQ*1: *Can the proposed approach outperform the existing approaches in terms of bug triaging accuracy?* First of all, we study whether the proposed iTriage approach can outperform the existing approaches. For this research question, we do not distinguish between new bug reports and halfway bug reports, and conduct experiments on all the bug reports to make the comparisons. The compared methods are listed as follows.

• $SVM + BOW$[3]. This method uses the TF-IDF value of each word as the textual features for bug reports, and applies SVM to treat the bug triaging problem as a classification problem.

• $TopicMinerMTM$[6]. This is the state-of-the-art information retrieval method for bug triaging. It applies topic models to learn the textual representations of bug reports, and integrates the metadata of product and component. Similarities between developers and bug reports are computed to recommend developers.

• $DBRNN + A$[9]. Since we build our model upon RNNs, we compare iTriage with another bug triaging method DBRNN+A which is also built upon RNNs. This method utilizes RNNs with attention to learn textual representations of bug reports, and then directly adds a softmax layer to recommend developers.

• *CNN Triager*[8]. This method is similar to DBRNN+A except that it applies CNNs to learn the representations of bug reports.

• *DeepTriage*[17]. DeepTriage is another method that uses RNNs to model textual content. It further considers metadata and the activity degrees of developers.

*RQ*2: *How does the proposed approach perform for halfway bug reports?* As mentioned above, this is the first work that pays special attention to halfway bug reports. Thus, for the second research question, we would like to evaluate whether our approach is accurate for halfway bug reports. To do so, we group the test data according to their tossing length, and report the accuracy results. For example, for a bug report in the test set whose tossing length is 4, we hide the last developer (i.e., the fixer), and put this sample in the groups whose length is less than 4.

*RQ*3: *Is the routing feature and meta feature useful in bug triaging?* For the third research question, we want to look into our approach. We integrate three different types of features to recommend the fixer, and we would like to evaluate whether all the three types of features are useful for bug triaging. Specifically, we modify the fixer suggestion model by removing the routing feature, the metadata feature, and both of them. For this research question, we also experiment on all the bug reports including both new bug reports and halfway bug reports.

4.2 Dataset Construction

To evaluate our approach, we collect the bug reports in three bug tracking systems as described in Subsection 2.2. Then, we preprocess the raw datasets. Not all bug reports are suitable for training a model. Anvik *et al.*[3] filtered out some noise data, such as bug reports with the resolution being "wontfix" or "worksforme". Following the previous studies[2−4,6,18], we only keep bug reports with the resolution being "FIXED" and the status being "CLOSED", "RESOLVED", or "VERIFIED". Furthermore, we also remove "REOPENED" bug reports as they are different from the original ones⑤. Xia *et al.*[6] noticed that the "Assigned To"

⑤The reopen behavior might take place long after the bug is fixed, and developers are likely to add comments to explain new observations such as unsuitable fixs or reappear due to version changes.

metadata might have the values of "unassigned", "issues", or "AJDT-inbox" in many bug reports. Obviously, they do not specify particular developers. We do not want to recommend them as the final fixers and remove them by keyword matching. We also exclude some inactive developers who have fixed less than 10 bug reports. The statistics of the resulting datasets are shown in Table 4.

Table 4. Statistics of Datasets

Name	Time	$\#\text{Reports}_{\text{raw}}$	$\#\text{Developers}_{\text{raw}}$	$\#\text{Reports}_{\text{processed}}$	$\#\text{Developers}_{\text{processed}}$	#Terms
Eclipse	2008.01–2017.03	210 487	1 785	120 468	1 014	12 854
Mozilla	2008.01–2017.02	300 165	2 164	161 098	1 497	16 975
Gentoo	2009.01–2017.11	165 483	1 475	94 530	973	9 242

For each bug report in the resulting datasets, we extract the summary and the description, as well as the metadata (product and component) and the tossing sequence. For each summary and description, we follow the preprocessing step of the existing work[9] by filtering out the code snippets, the hex code, the URLs, and the stack traces. Then, we employ NLTK⑥ to tokenize the texts, keep the stopwords, remove punctuation marks and low-frequency words, and transfer them to lower cases. Finally, we truncate the preprocessed text. We use 100 as the maximum of text length, and notice that more than 80% of bug reports are within the threshold; thus we do not abandon too much data. The tossing sequence is also limited to the length of 5, as we find out most tossing sequences are within this length.

In order to simulate the real scenarios, we first sort all the bug reports by their creation time. Then, we averagely divide the sorted bug reports into 11 non-overlapping frames, and execute 10-fold iterations to cover all frames as shown in Fig.5. For example, in fold 1, we use frame 1 as training data, and frame 2 as test data; in fold 2, we use the first two frames as training data, and frame 3 as test data; and so on and so forth. Such setup is widely used by existing work[4,6,18]. For each iteration, we recommend one developer as the fixer, and compute the average accuracy in each test frame. Here a prediction is accurate if it recommends the right fixer as in the datasets.

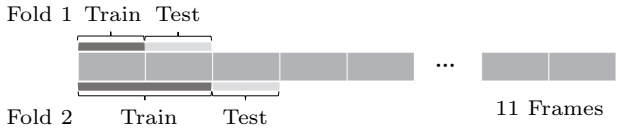


Fig.5. 10-fold experimental setting. We split the dataset into 11 frames. For each fold K, we choose first K frames as training data and the $K+1$ frame as test data.

4.3 Implementation Details

The implementation details are as follows.

For the classification models based on bag-of-word representations (i.e., SVM+BOW), we use the Gensim library⑦ to generate TF-IDF textual vectors. SVM is implemented by using libsvm[19] with python interface⑧. We tune the parameters of SVM by ourselves and report the best results.

For TopicMinerMTM, we implement it by ourselves since the source code is not publicly available. Specially, we implement it with CVB0[20] learning. We follow the settings of the weights described in the paper that introduces TopicMinerMTM[6].

For the deep learning based methods, we use Keras⑨ to implement them and let them share the same settings (e.g., embedding dimensions, hidden state dimensions). For the proposed method, the textual input is truncated by the length of 100. Each word is embedded with 100 dimensions and is initialized with pre-trained embedding from Glove[21]. The tossing sequence length is limited to 5, since most of tossing sequences are in this range. Reporters and developers are embedded with 50 dimensions. For GRU neurons, the hidden state size is set to 100. Since the encoder uses bidirectional RNNs, the output size of each word is 200 in total (100 for one direction). The embedding size of metadata is set to 100. We also employ dropout[22] to avoid over-fitting, with dropout rate set to 0.5. The above settings are similarly applied for the competitors for fairness. For the proposed model, we use "categorical_crossentropy" provided by Keras as our loss function and we use "nadam" optimizer[23] which automatically adjusts the learning rate to train the model. For other parameters, we limit the vocabulary size up to

⑥http://www.nltk.org, July 2019.

⑦https://radimrehurek.com/gensim/, July 2019.

⑧https://www.csie.ntu.edu.tw/c̆jlin/libsvm/#python, July 2019.

⑨https://keras.io/, July 2019.

50 000 words that are most frequently used in bug reports, and use minibatch stochastic gradient descent with the batch size of 64.

We use a PC with a 3.40 GHz Intel i7-4770 CPU, 32 GB physical memory and a Nvidia GTX1080 GPU to execute all these methods. For SVM+BOW and TopicMinerMTM, they only execute on CPU. And for deep learning based methods, we use GPU to accelerate the training and testing steps.

5 Evaluation Results

In this section, we present the evaluation results for the three research questions.

5.1 Results for RQ1

The comparison results on three datasets are in Table 5 and Fig.6, where we report the top-1 accuracy and the top-5 accuracy. Here, Table 5 shows the average results over the 10 folds and Fig.6 shows the results on each fold. The x-axis in the figure indicates the frame number, and the y-axis is the prediction accuracy. First of all, we can observe that iTriage outperforms all its competitor methods in all the frames for top-1 accuracy. For example, across all three projects, iTriage on average improves the prediction accuracy of SVM+BOW by 22.93%, DBRNN+A by 21.96%, CNN Triager by 21.76%, TopicMinerMTM by 9.39%, and DeepTriage by 5.83%. Although the relative improvements in the top-5 case are smaller than that in the top-1 case, we have also tested the significance of the improvements with Wilcoxon signed rank test, and the results show that the proposed iTriage significantly outperforms all the competitors with p-value < 0.01, which means the proposed approach can still significantly outperform the existing approaches in terms of the average improvements as shown in Table 5. Second, methods use only textual information without considering the metadata and the tossing sequences (i.e., SVM+BOW, DBRNN-A, and CNN Triager) are less accurate than the other competitors. This supports the effectiveness of metadata and tossing sequences. Third, iTriage performs better than the other deep learning competitors. The main reason is that we integrate more information as input. Notice that, deep learning methods that use only textual information are still a little better than SVM+BOW. Fourth, we find that the accuracy results in Mozilla are relatively lower. The reason is that Mozilla has more developers than the other projects (near 49% more than Eclipse and 53% more than Gentoo). Fifth, in all the 10 folds, the average accuracy of each method is relatively stable. The reason might be that with the number of bug reports increasing, the number of developers and the size of vocabulary are also increasing. In other words, the complexity/difficulty of the problem increases as the training data increases.

For completeness, we also report the training time and the testing time for the compared methods. The results are shown in Tables 6–9. As we can see from the tables, the proposed approach spends less or comparable time with its competitors in terms of both the training stage and the testing stage. Moreover, the proposed approach scales linearly in both the training stage and the testing stage w.r.t. the data size.

Summary. Overall, the results show that the proposed iTriage can outperform the existing bug triaging baselines while introducing little additional time overhead.

5.2 Results for RQ2

Next, we study how iTriage performs for halfway bug reports. The results are shown in Fig.7. The x-axis in the figures indicates the tossing length, and the y-axis is the prediction accuracy. Notice that, tossing length zero means a new bug report.

As we can see from Fig.7, the accuracy of our approach generally increases as the tossing length grows on all the three datasets. In other words, if we have more developers in the current tossing sequence, we can identify the right fixer more accurately. The reason is as follows. The developers in the predicted tossing sequence are related to each other; thus, given the pre-

Table 5. Top-1 Accuracy and Top-5 Accuracy on Average

Project	SVM+BOW	TopicMinerMTM	DBRNN+A	CNN Triager	DeepTriage	iTriage
Eclipse@1	0.280 57	0.368 95	0.287 11	0.289 23	0.416 55	0.478 60
Eclipse@5	0.514 67	0.800 56	0.522 44	0.541 07	0.802 57	0.824 29
Mozilla@1	0.204 31	0.290 43	0.207 40	0.208 32	0.323 15	0.373 08
Mozilla@5	0.412 54	0.651 86	0.429 22	0.433 50	0.682 74	0.701 63
Gentoo@1	0.227 64	0.459 10	0.247 10	0.249 99	0.485 78	0.543 32
Gentoo@5	0.510 20	0.813 70	0.544 70	0.554 60	0.787 50	0.799 10

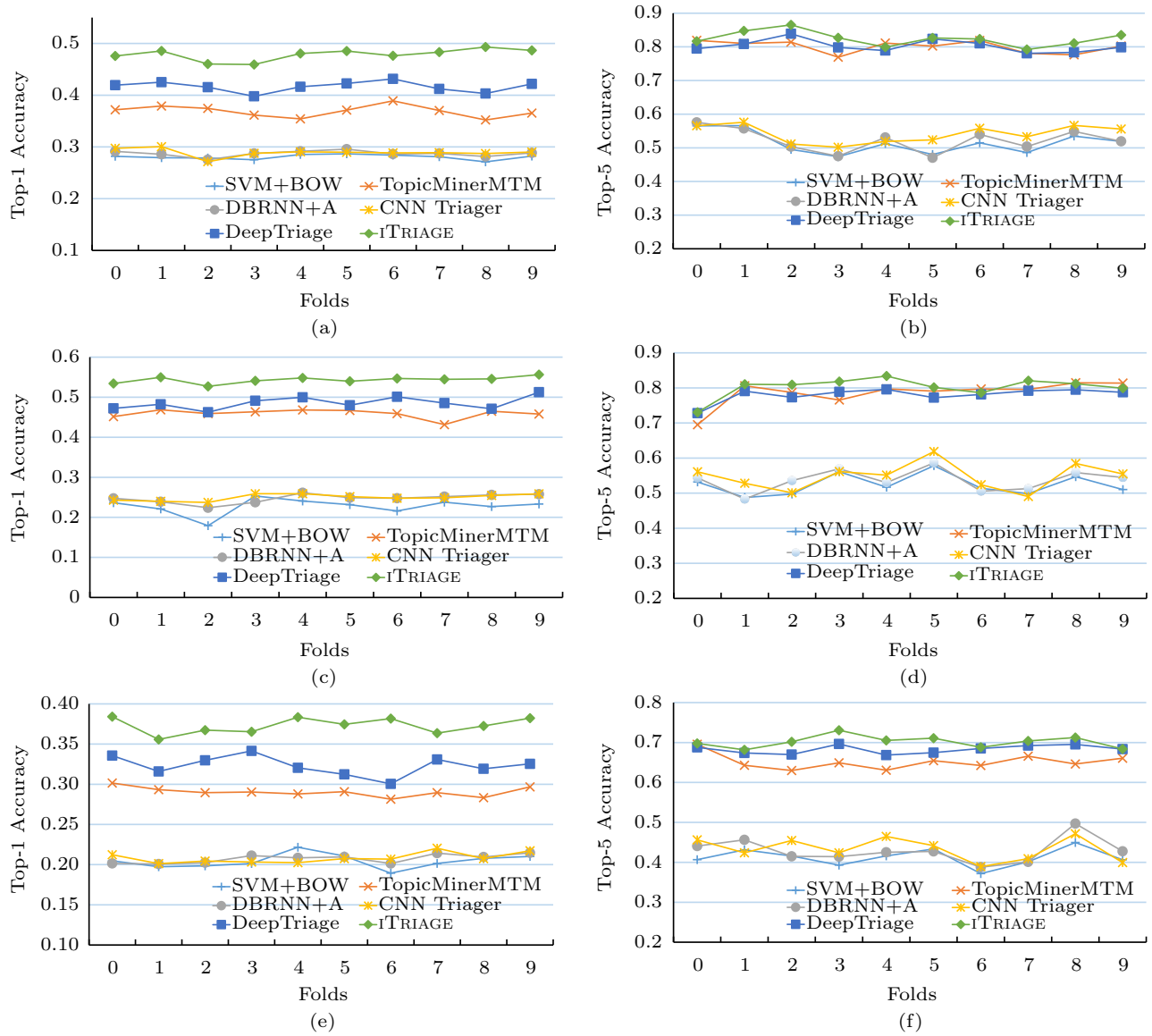


Fig.6. Top-1 and top-5 accuracy comparisons in (a) Eclipse data (top-1), (b) Eclipse data (top-5), (c) Mozilla data (top-1), (d) Mozilla data (top-5), (e) Gentoo data (top-1), and (f) Gentoo data (top-5).

Table 6. Training Time (min) on Eclipse

Approach	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total
SVM+BOW	5.94	12.59	19.31	26.13	32.89	38.86	45.48	52.50	59.20	66.01	358.91
TopicMinerMTM	62.54	132.59	203.39	275.18	346.34	409.20	478.86	552.82	623.35	695.01	3 779.28
DBRNN+A	7.26	15.39	23.61	31.94	40.20	47.50	55.59	64.18	72.37	80.69	438.73
CNN Triager	6.28	13.31	20.42	27.63	34.78	41.09	48.09	55.52	62.60	69.80	379.52
DeepTriage	6.67	14.14	21.69	29.35	36.94	43.64	51.07	58.96	66.48	74.12	403.06
iTRIAGE	7.83	16.60	25.46	34.45	43.36	51.23	59.95	69.21	78.04	87.01	473.14

vious tossing sequence, we can guess the final tossed developer based on the developers' tossing habit. Additionally, when the tossing length is larger than 3, the accuracy does not increase any more. We manually inspect several bug reports and point out two potential reasons. First, for long tossing sequences, there are cases when two or more developers reassign the bug reports to each other repeatedly. Second, there are cases when the finally tossed developer is far from the first tossed developer in long tossing sequences (e.g., from

Table 7. Training Time (min) on Mozilla

Approach	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total
SVM+BOW	12.06	24.61	39.95	52.96	68.41	81.87	94.23	107.02	120.17	135.51	736.79
TopicMinerMTM	124.92	254.96	413.86	548.66	708.69	848.13	976.19	1 108.67	1 244.88	1 403.75	7 632.71
DBRNN+A	15.12	30.86	50.09	66.41	85.78	102.66	118.16	134.19	150.68	169.91	923.86
CNN Triager	12.54	25.59	41.54	55.07	71.13	85.13	97.98	111.28	124.95	140.90	766.11
DeepTriage	13.10	26.74	43.40	57.54	74.32	88.94	102.37	116.26	130.54	147.20	800.41
iTriage	15.63	31.90	51.78	68.65	88.67	106.12	122.14	138.72	155.76	175.64	955.01

Table 8. Training Time (min) on Gentoo

Approach	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Total
SVM+BOW	3.16	6.28	9.34	12.23	15.36	18.25	21.59	24.98	28.09	31.02	170.30
TopicMinerMTM	33.55	66.72	99.23	129.93	163.18	193.84	229.30	265.33	298.39	329.45	1 808.92
DBRNN+A	3.81	7.58	11.27	14.76	18.54	22.02	26.05	30.14	33.89	37.42	205.48
CNN Triager	3.43	6.82	10.14	13.28	16.68	19.81	23.44	27.12	30.50	33.68	184.90
DeepTriage	3.64	7.24	10.77	14.10	17.71	21.04	24.89	28.80	32.39	35.76	196.34
iTriage	4.27	8.49	12.63	16.54	20.77	24.67	29.18	33.77	37.98	41.93	230.23

Table 9. Average Predicting Time (s) per Bug Report

Dataset	SVM+BOW	TopicMinerMTM	DBRNN+A	CNN Triager	DeepTriage	iTriage
Eclipse	0.36	3.79	0.44	0.39	0.41	0.47
Mozilla	0.56	5.84	0.72	0.58	0.60	0.73
Gentoo	0.22	2.33	0.26	0.24	0.25	0.29

totally different product or component). Further investment is needed for such cases.

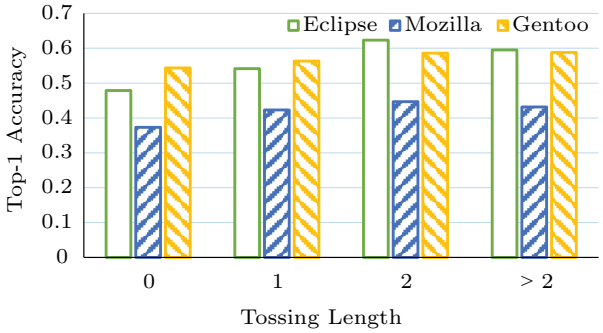


Fig.7. Accuracy of iTriage with different tossing lengths. In general, the proposed iTriage is more accurate for halfway bug reports.

Summary. Overall, the results show that the proposed iTriage can make use of the existing developers in the tossing sequence so as to improve the triaging accuracy.

5.3 Results for RQ3

For RQ3, we study the effectiveness of components, i.e., the three types of features in our fixer suggestion model. The result is listed in Table 10. In the table, we test three variants of the proposed approach: "text" which means only the textual feature is used, "text+meta" which means that both the textual feature and the metadata feature are used, and "text+routing" which means that both the textual feature and the routing feature are used.

As we can see from Table 10, the accuracy of "text" group is relatively low, indicating the importance of the other two types of features. Then, when either the metadata feature or the routing feature is added, the performance significantly increases, that is, both components are useful. Furthermore, we manually inspect some results and find that even if the performance of "text+meta" and "text+routing" is close, they do well at different cases. For "text+meta", it is more likely to predict the most active developer within the product and the component. For "text+routing", it is more likely to predict the reporter directly. Such a phenomenon can be further leveraged and we leave it as future work. Finally, iTriage is better than all the component methods. This means the combination of the three aspects of features is helpful for the bug triaging task.

Table 10. Accuracy of Feature Combinations

Name	Text	Text+Meta	Text+Routing	iTriage
Eclipse	0.297 4	0.459 3	0.461 8	0.478 6
Mozilla	0.213 4	0.321 9	0.352 7	0.373 0
Gentoo	0.249 3	0.457 4	0.503 7	0.543 3

Summary. Overall, the results show that all the three aspects (i.e., textual content modeling, metadata modeling, and tossing sequence modeling) of iTriage are useful to improve the triaging accuracy.

6 Threats to Validity

In this section, we identify some threats to validity of this work.

For internal validity, we mainly consider two aspects. First, for the ground truth of the suitable fixer, we follow the previous approaches to select the "Assignee" (also known as "Assigned To") metadata[6,18]. However, the actual fixer might not be the best developer to fix the bug report, and other developers could also have the ability to fix it. To this end, some researchers (e.g., [3]) proposed to construct a dataset where the developers who may be able to fix the bug are manually labeled. However, this would be too expensive to achieve given that our dataset is much larger. Second, when processing the raw datasets, we filter out the developers who have fixed less than 10 bugs. It is possible that these developers would fix the future bugs, and our model will produce false negatives. However, these developers are generally inactive in these projects, and after deleting them we still have plenty of active developers for each project. Collecting the bug reports for a longer period or focusing on only bug reports of the finished projects can alleviate such issues. Additionally, we have deleted the reopened bug reports in the datasets. The reason is that these bug reports may have two different fixers and tossing sequences which could mislead the training process of the proposed approach. We leave the treatment for these reopened bug reports as future work.

For external validity, in this paper, we have crawled and analyzed three open source projects. However, open source projects might be different with industry projects. Therefore, we are not certain whether our results can be generalized to industry projects. However, if the two facts (i.e., reporter locality and tossing relationships) also hold for industry projects, there is a high chance that the proposed approach will work well for the industry projects.

7 Related Work

In this section, we review related work. Bug reports play an important role in software development and maintenance. Many researchers have studied related topics to help developers fix bugs more efficiently. We mainly discuss about two research topics in the existing work. The first one is to route the bug reports to suitable fixers, and the second one is to optimize the quality of the bug reports.

7.1 Bug Triaging

In the area of automatic bug triaging, researchers have proposed a series of machine learning (e.g., [2-4, 24]) and information retrieval based approaches (e.g., [5, 6, 25]). Machine learning based approaches usually regard developers as labels, and predict suitable developers with a classification model. Many classification models have been used by existing methods. For example, Anvik *et al.*[3] modeled this task as a text classification problem with Naive Bayes classifier, and Cubranic and Murphy[26] used the SVM classifier instead. As for information retrieval methods, they constructed the representations of both developers and bug reports and then computed their similarities. For example, Tamrawi *et al.*[18] used fuzzy set and developer caching to calculate the similarities between developers and bug reports; Naguib *et al.*[25], Yang *et al.*[5], and Xia *et al.*[6] used topic models to compare the similarities between the bug reports and the developers. Instead of these two categories of methods, some developers reformulated the bug triaging problem. For example, Alenezi *et al.*[27] focused on redistributing the load of overloaded developers. Zou *et al.*[28] introduced feature selection to improve the accuracy of bug triaging and reduced the training sets. Park *et al.*[29] proposed a cost-aware approach to balance accuracy and cost.

In terms of the input used by these existing approaches, the textual content of bug reports is the key part. Additionally, metadata, tossing sequence, relationships between developers, the developer roles, and the code annotations have also been used. For example, Yang *et al.*[5] and Bhattacharya and Neamtiu[4] considered the meta data. Jeong *et al.*[2] and Bhattacharya Neamtiu[4] used tossing graph to optimize the recommendation list; Hu *et al.*[30] modeled the relationship between developers and source code components, as well as their associated bugs. Zhang *et al.*[31] used more social relationships between developers. There are also some other studies related to the tossing graph. For example, Wang *et al.*[32] proposed to analyze the collaborations in bug repositories. Wu *et al.*[33] studied what factors affect the length of tossing path, and found that working theme, product, component, and degree centrality are key factors.

Recently, deep learning techniques have been used to improve the performance of bug triaging. For example, Lee *et al.*[8] proposed to use CNNs and Mani *et al.*[9] proposed to use Bidirectional RNNs with attention. However, the focus of these approaches is on the textual content of bug reports. Different from these

approaches, the key novelty of our approach is to simultaneously integrate the following three aspects: 1) the textual content in the bug reports, 2) the metadata in the bug reports, and 3) the tossing sequences of bug reports.

7.2 Bug Report Quality

Several researches have proposed to study the quality of bug reports. For example, Hooimeijer and Weimer[34] measured the bug report quality and predicted whether bug reports can be resolved within a given time. Demeyer and Lamkanfi[35] noticed that bug reports may contain errors such as wrong components, and proposed to predict such cases for a particular bug report; Herraiz *et al.*[36] held the opinion that the bug reports of Bugzilla are too complex, and found that the severity of bug reports can be reduced from seven options to three. Wu *et al.*[37] found that bug reports are often incomplete and proposed to check the completion of a given bug report. Zanetti *et al.*[38] focused on the validity of bug reports. They extracted features from collaboration networks between team members to determine whether the bug report is valid or not. Fan *et al.*[39] extended the extracted features and improved the performance. These techniques could enhance the overall quality of bug reports, which also benefits automatic bug triaging process.

The detection of duplicate bug reports has also been widely studied. Anvik *et al.*[3] conducted a large-scale empirical study on the bug repositories of Eclipse and Firefox, and found that a large percentage of bug reports are identified as duplicate by the developers. Hiew[40] made the first attempt to detect duplicate bug reports by calculating the similarities between bug reports. Runeson *et al.*[41] studied the duplicate detection problem with industrial projects and they considered more textual features (e.g., software versions, testers, the submission date). Wang *et al.*[42] used natural language information and execution information to suggest a small set of the most similar bug reports. Nyugen *et al.*[43] employed topic models to obtain the textual representations and then computed their similarities to detect duplicate bug reports. Tian *et al.*[44] proposed to consider the similarity between a new bug report and multiple existing bug reports (instead of only the most similar one) to decide whether the bug reported in the new bug report is actually a duplicate bug. Besides, some researchers began to conduct empirical studies on duplicate bug reports[45−47].

All these existing researches are complementary with our work. That is, we may consider to first improve the quality of bug reports before assigning them to suitable fixers.

8 Conclusions

In this paper, we proposed a novel approach, iTriage, to simultaneously integrate the following three aspects: 1) the textual contents in the bug reports, 2) the metadata in the bug reports, and 3) the tossing sequences of bug reports. The proposed approach consists of two models, namely, the feature learning model and fixer suggestion model. The feature learning model is a sequence-to-sequence model to learn the textual feature and the routing feature at the same time. The fixer suggestion model is a classification model to combine the textual feature, the metadata feature, and the routing feature to predict the bug fixers. The experimental results showed that iTriage improves the accuracy of the bug triaging compared with the state-of-the-art approaches.

In the future, we plan to explore the usage of code snippets, URLs, and stack traces in the textual content, which are currently excluded by our model. We believe that these contents can provide extra information to improve the accuracy of the bug triaging. We also plan to further investigate the tossing sequence, and understand why developers toss bug reports to each other.

References

[1] Bertram D, Voida A, Greenberg S, Walker R. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In *Proc. the 2010 ACM Conference on Computer Supported Cooperative Work*, February 2010, pp.291-300.

[2] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In *Proc. the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, August 2009, pp.111-120.

[3] Anvik J, Hiew L, Murphy G C. Who should fix this bug? In *Proc. the 28th International Conference on Software Engineering*, May 2006, pp.361-370.

[4] Bhattacharya P, Neamtiu I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Proc. the 2010 IEEE International Conference on Software Maintenance*, September 2010, Article No. 41.

[5] Yang G, Zhang T, Lee B. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In *Proc. the 38th Annual Computer Software and Applications Conference*, July 2014, pp.97-106.

[6] Xia X, Lo D, Ding Y, Al-Kofahi J M, Nguyen T N, Wang X. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 2017, 43(3): 272-297.

[7] Zhang T, Yang G, Lee B, Lua E K. A novel developer ranking algorithm for automatic bug triage using topic model and developer relations. In *Proc. the 21st Asia-Pacific Software Engineering Conference*, December 2014, pp.223-230.

[8] Lee S R, Heo M J, Lee C G, Kim M, Jeong G. Applying deep learning based automatic bug triager to industrial projects. In *Proc. the 11th Joint Meeting on Foundations of Software Engineering*, September 2017, pp.926-931.

[9] Mani S, Sankaran A, Aralikatte R. DeepTriage: Exploring the effectiveness of deep learning for bug triaging. arXiv:1801.01275, 2018. https://arxiv.org/pdf/1801.01275.pdf, June 2019.

[10] Xi S Q, Yao Y, Xiao X S, Xu F, Lu J. An effective approach for routing the bug reports to the right fixers. In *Proc. the 10th Asia-Pacific Symposium on Internetware*, September 2018, Article No. 11.

[11] Zhang X F, Yao Y, Wang Y J, Xu F, Lu J. Exploring metadata in bug reports for bug localization. In *Proc. the 24th Asia-Pacific Software Engineering Conference*, December 2017, pp.328-337.

[12] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv:1409.0473, 2014. https://arxiv.org/pdf/1409.0473.pdf, June 2019.

[13] Hinton G, Deng L, Yu D *et al*. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012, 29(6): 82-97.

[14] Johnson R, Zhang T. Supervised and semi-supervised text categorization using LSTM for region embeddings. arXiv:1602.02373, 2016. https://arxiv.org/pdf/1602.02373.pdf, June 2019.

[15] Yang Z C, Yang D Y, Dyer C, He X D, Smola A, Hovy E. Hierarchical attention networks for document classification. In *Proc. the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, June 2016, pp.1480-1489.

[16] Cho K, van Merrienboer B, Gülçehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv:1406.1078, 2014. https://arxiv.org/pdf/1406.1078.pdf, June 2019.

[17] Xi S Q, Yao Y, Xu F, Lu J. Bug triaging approach based on recurrent neural networks. *Journal of Software*, 2018, 29(8): 2322-2335. (in Chinese)

[18] Tamrawi A, Nguyen T T, Al-Kofahi J M, Nguyen T N. Fuzzy set and cache-based approach for bug triaging. In *Proc. the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, September 2011, pp.365-375.

[19] Chang C C, Lin C J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2011, 2(3): Article No. 27.

[20] Hoffman M, Bach F R, Blei D M. Online learning for latent Dirichlet allocation. In *Proc. the 24th Annual Conference on Neural Information Processing Systems*, December 2010, pp.856-864.

[21] Pennington J, Socher R, Manning C. Glove: Global vectors for word representation. In *Proc. the 2014 Conference on Empirical Methods in Natural Language Processing*, October 2014, pp.1532-1543.

[22] Hinton G E, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov R R. Improving neural networks by preventing co-adaptation of feature detectors. arXiv:1207.0580, 2012. https://arxiv.org/pdf/1207.0580.pdf, June 2019.

[23] Sutskever I, Martens J, Dahl G, Hinton G. On the importance of initialization and momentum in deep learning. In *Proc. the 30th International Conference on Machine Learning*, June 2013, pp.1139-1147.

[24] Lin Z P, Shu F D, Ye Y, Hu C Y, Wang Q. An empirical study on bug assignment automation using Chinese bug data. In *Proc. the 3rd International Symposium on Empirical Software Engineering & Measurement*, October 2009, pp.451-455.

[25] Naguib H, Narayan N, Brügge B, Helal D. Bug report assignee recommendation using activity profiles. In *Proc. the 10th Working Conference on Mining Software Repositories*, May 2013, pp.22-30.

[26] Cubranic D, Murphy G C. Automatic bug triage using text categorization. In *Proc. the 16th International Conference on Software Engineering & Knowledge Engineering*, June 2004, pp.92-97.

[27] Alenezi M, Magel K, Banitaan S. Efficient bug triaging using text mining. *Journal of Software*, 2013, 8(9): 2185-2190.

[28] Zou W Q, Hu Y, Xuan J F, Jiang H. Towards training set reduction for bug triage. In *Proc. the 35th Annual Computer Software and Applications Conference*, July 2011, pp.576-581.

[29] Park J, Lee M W, Kim J, Hwang S, Kim S. CosTriage: A cost-aware triage algorithm for bug reporting systems. In *Proc. the 25th AAAI Conference on Artificial Intelligence*, August 2011, Article No. 22.

[30] Hu H, Zhang H Y, Xuan J F, Sun W G. Effective bug triage based on historical bug-fix information. In *Proc. the 25th International Symposium on Software Reliability Engineering*, November 2014, pp.122-132.

[31] Zhang W, Wang S, Wang Q. KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity. *Information and Software Technology*, 2016, 70: 68-84.

[32] Wang S, Zhang W, Yang Y, Wang Q. DevNet: Exploring developer collaboration in heterogeneous networks of bug repositories. In *Proc. the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, October 2013, pp.193-202.

[33] Wu H R, Liu H Y, Ma Y T. Empirical study on developer factors affecting tossing path length of bug reports. *IET Software*, 2018, 12(3): 258-270.

[34] Hooimeijer P, Weimer W. Modeling bug report quality. In *Proc. the 22nd IEEE/ACM International Conference on Automated Software Engineering*, November 2007, pp.34-43.

[35] Demeyer S, Lamkanfi A. Predicting reassignments of bug reports — An exploratory investigation. In *Proc. the 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp.327-330.

[36] Herraiz I, Germán D M, González-Barahona J M, Robles G. Towards a simplification of the bug report form in Eclipse. In *Proc. the 2008 International Working Conference on Mining Software Repositories*, May 2008, pp.145-148.

[37] Wu L, Xie B Y, Kaiser G E, Passonneau R J. BUGMINER: Software reliability analysis via data mining of bug reports. In *Proc the 23rd International Conference on Software Engineering & Knowledge Engineering*, July 2011, pp.95-100.

[38] Zanetti M S, Scholtes I, Tessone C J, Schweitzer F. Categorizing bugs with social networks: A case study on four open source software communities. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.1032-1041.

[39] Fan Y R, Xia X, Lo D, Hassan A E. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering*. doi:10.1109/TSE.2018.2864217.

[40] Hiew L. Assisted detection of duplicate bug reports [Ph.D. Thesis]. University of British Columbia, 2006.

[41] Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. In *Proc. the 29th International Conference on Software Engineering*, May 2007, pp.499-510.

[42] Wang X Y, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. the 30th International Conference on Software Engineering*, May 2008, pp.461-470.

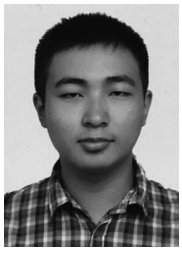
[43] Nguyen A T, Nguyen T T, Nguyen T N, Lo D, Sun C. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proc. the 27th IEEE/ACM International Conference on Automated Software Engineering*, September 2012, pp.70-79.

[44] Tian Y, Sun C, Lo D. Improved duplicate bug report identification. In *Proc. the 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp.385-390.

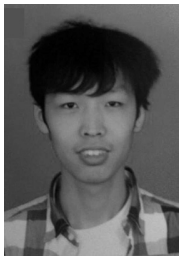
[45] Bettenburg N, Premraj R, Zimmermann T, Kim S. Duplicate bug reports considered harmful ... really? In *Proc. the 24th International Conference on Software Maintenance*, September 2008, pp.337-345.

[46] Cavalcanti Y C, de Almeida E S, da Cunha C E A, Lucrédio D, Meira S R. An initial study on the bug report duplication problem. In *Proc. the 14th European Conference on Software Maintenance & Reengineering*, March 2010, pp.264-267.

[47] Cavalcanti Y C, Almeida E S, Cunha C E A *et al*. The bug report duplication problem: An exploratory study. *Software Quality Journal*, 2013, 21(1): 39-66.



Sheng-Qu Xi currently is a Ph.D. student of a five-year educational system in State Key Laboratory for Novel Software, the Department of Computer Science and Technology at Nanjing University, Nanjing. He received his B.S. degree in computer science from Nanjing University, Nanjing, in 2014. His major research interest includes mining bug reports and deep learning for software engineering.



Yuan Yao is an assistant researcher in State Key Laboratory for Novel Software, the Department of Computer Science and Technology at Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2015. His current research interest includes mining networked data with applications to social media analytics and software analytics.

Xu-Sheng Xiao is an assistant professor in the Department of Electrical Engineering and Computer Science at Case Western Reserve University, Cleveland. He received his Ph.D. degree in computer science from North Carolina State University, Raleigh, advised by Prof. Tao Xie and Prof. Laurie Williams in 2014. He was a visiting student at University of Illinois at Urbana-Champaign in 2013–2014. His general research interests span between software engineering and computer security.

Feng Xu is a professor in State Key Laboratory for Novel Software, the Department of Computer Science and Technology at Nanjing University, Nanjing. He received his B.S. and M.S. degrees in computer science from Hohai University, Nanjing, in 1997 and 2000, respectively. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2003. His research interests include software defect localization, data mining, recommender systems and trust management.

Jian Lv is a professor in the Department of Computer Science and Technology and the director of the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing. He received his B.S. and Ph.D. degrees in computer science from Nanjing University, Nanjing, in 1982 and 1988 respectively. He serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of CCF. His research interests include software methodologies, software automation, software agents, and middleware systems.