2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC) | 979-8-3503-3750-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPC58990.2023.00046

Still Confusing for Bug-Component Triaging? Deep Feature Learning and Ensemble Setting to Rescue

Yanqi Su
Australian National University
Australia
Yanqi.Su@anu.edu.au

Zheming Han
Australian National University
Australia
u6865707@anu.edu.au

Zhipeng Gao
Zhejiang University
China
zhipeng.gao@zju.edu.cn

Zhenchang Xing*†
Data61, CSIRO
Australia
Zhenchang.Xing@data61.csiro.au

Qinghua Lu
Data61, CSIRO
Australia
Qinghua.Lu@data61.csiro.au

Xiwei Xu
Data61, CSIRO
Australia
Xiwei.Xu@data61.csiro.au

***Abstract*—To speed up the bug-fixing process, it is essential to triage bugs into the right components as soon as possible. Given the large number of bugs filed everyday, a reliable and effective bug-component triaging tool is needed to assist this task. LR-BKG is the state-of-the-art toolkit for doing this. However, the suboptimal performance for recommending the right component at the first position (low Top-1 accuracy) limits its usage in practice. We thoroughly investigate the limitations of LR-BKG and find out the gap between the manual feature design of LR-BKG and the characteristics of bug reports causes such suboptimal performance. Therefore, we propose an approach,** DEEPTRIAG**, which uses the large scale pre-trained models to extract deep features automatically from bug reports (including bug summary and description), to fill this gap.** DEEPTRIAG **transforms bug-component triaging into a multi-classification task (CodeBERT-Classifier) and a generation task (CodeT5-Generator). Then, we ensemble the prediction results from them to improve the performance of bug-component triaging further. Extensive experimental results demonstrate the superior performance of** DEEPTRIAG **on bug-component triaging over LR-BKG. In particular, the overall Top-1 accuracy is improved from 56.2% to 68.3% on Mozilla dataset and from 51.3% to 64.1% on Eclipse dataset, which verifies the effectiveness and generalization of our approach on improving the practical usage for bug-component triaging.**

***Index Terms*—Bug Triaging, Deep Learning, Text Classification**

I. INTRODUCTION

Nowadays, bug reports play an essential role in outlining information about what is going wrong about software. Typically, when a developer or user encounters an unexpected outcome, he/she may formulate this problem as a bug report (usually including the bug summary and bug description) and submit this bug to the bug tracking system (e.g., Bugzilla).

Considering the huge amount of bugs that happen day-to-day and the sheer amount of information contained in the bug report, it is very time-consuming and labor-intensive to manually triage the bug into the target component. Moreover, since the lack of knowledge or missing the context information of specific bugs, this process is often error prone [1]. For the wrongly assigned bug report, it has to be reassigned/tossed to the next possible component, which may cost a lot of human resources and/or wait a long time before the bug is assigned to the right component, leaving the developers or the users unsatisfied. Therefore, it is preferable to have toolkits that can automatically triage newly generated bugs to the right component. The previous works [1], [2] have investigated the bug-component triaging task. Su et al. [1] propose a model, namely LR-BKG, which achieves the state-of-the-art performance for bug-component triaging on both tossed bugs and non-tossed bugs. For bug reports having the reassignment process, we call them **tossed bugs**. If bug reports are assigned to the right components initially, they are **non-tossed bugs**.

However, as shown in their paper [1], we found that LR-BKG has a relatively suboptimal performance regarding Top-1 accuracy (i.e., 56.2%). After empirically investigating their experimental results, we find two main challenges their approach fails to handle. These two main challenges are about two concepts: **confusing components** and **few shot components**. For confusing components, if more than one bug of *component A* was once mistossed to *component B*, *component B* is the confusing component of *component A*. For example, there are 16 bugs belonging to Toolkit::Password Manager: Site Compatibility once mistossed to Toolkit::Password Manager. Thus, Toolkit::Password Manager is the confusing component of Toolkit::Password Manager: Site Compatibility. Few shot components are referred to as components with limited bugs. With these two concepts, the two main challenges are as follows:

Firstly, LR-BKG cannot accurately handle the challenge of confusing component distinction. As shown in Section II-A1, the mistossed bugs by LR-BKG are, more often than not, wrongly assigned to their confusing components. If the bugs which have been triaged into confusing components can be correctly assigned, the performance of LR-BKG will be significantly boosted. Secondly, LR-BKG has difficulty in precisely triaging bugs belonging to few shot components. That is, if the

*Corresponding author.
†Also with Australian National University.

2643-7171/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPC58990.2023.00046

target component contains limited bugs, LR-BKG is hard to route the newly generated bug to it. If this challenge is solved, the performance on few shot components will be improved.

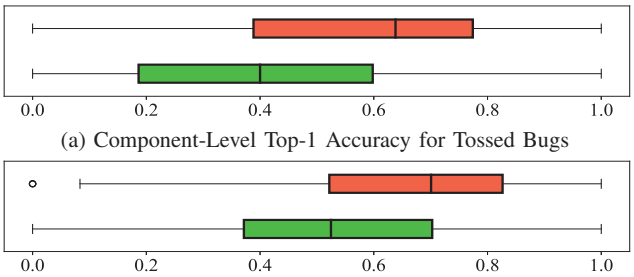
We attribute these two challenges to the gap between the characteristics of bug reports and the feature design of the state-of-the-art bug-component triaging tool (i.e., LR-BKG). First, the handcrafted features designed by LR-BKG are too shallow and coarse to capture the key information among the confusing components. Second, the manually designed features rely on the experience and knowledge of human experts, which can not properly handle specific cases (e.g., few shot components). To bridge the gap between the data and the tool, we adopt the large scale pre-trained models (i.e., CodeBERT and CodeT5) as feature extractors for this task. A huge number of works based on the pre-trained models have been proved to be effective for learning implicit connections and semantic features from natural language text [3], [4].

In particular, we propose a neural network based model, named DEEPTRIAG to do bug triaging among components, especially for improving the suboptimal Top-1 accuracy. DEEPTRIAG consists of three parts: in the first part, we train a classifier (CodeBERT-Classifier) to calculate the probabilities of a bug report belonging to different components based on the features extracted by CodeBERT [5]. In the second part, we transform bug-component triaging into a generation task (CodeT5-Generator), which generates the corresponding component based on the bug summary and description of bug report by using the encoder and decoder model, CodeT5 [6]. At last, we ensemble all the prediction results of the above classifier and generator to get better overall prediction results. The extensive experiments show that our model outperforms the state-of-the-art bug-component triaging tool, LR-BKG, by a large margin, and significantly improves the suboptimal performance on Top-1 accuracy for bug-component triaging. This work makes the following contributions:

- We thoroughly analyze the state-of-the-art tool, LR-BKG and find out its limitations on the bug-component triaging.
- We propose a deep-learning based approach for bug-component triaging, named DEEPTRIAG.
- For generalization, we evaluate LR-BKG and DEEPTRIAG on bug reports from both Mozilla and Eclipse.
- The experimental results show the superiority of our model over the state-of-the-art baseline, especially the Top-1 accuracy. Our replication package can be found here.

II. MOTIVATION

The state-of-the-art toolkit, LR-BKG [1] utilizes a learning to rank framework combining with bug tossing knowledge graph to perform bug triaging among components. It manually designs a rich set of features covering bug feature, component features and bug-component features. Bug-component features aim at capturing the bug-component relations. For each bug-component pair, it explicitly models bug-component relations with similarities between the bug summary and the information (component name, component description and bugs) of the component. For the similarities between the bug and bugs of components, LR-BKG takes Top-30 of them as features.



(a) Component-Level Top-1 Accuracy for Tossed Bugs

(b) Component-Level Top-1 Accuracy for Non-Tossed Bugs

Fig. 1: Component-Level Top-1 Accuracy Distribution

TABLE I: No. Bug Distribution of Product::Component

#Group	1	2	3	Sum
#Train Bug	31-250	250-500	500+	78,870
#P::C	91	46	49	186

LR-BKG has been shown the effectiveness of recommending top-10 components. That is, Top-10 accuracy can reach 88.1% on average. It falls short, however, in accurately recommending an actual top-1 component, namely Top-1 accuracy (46.9% for tossed bugs and 59.3% for non-tossed bugs). Due to the suboptimal performance on recommending components at top-1 position, it is still difficult and time-consuming for bug reporters to decide which components to choose from the ranking list. Therefore, it is necessary to improve Top-1 accuracy, especially for the toolkit usage in practice.

Given the limitations of the existing toolkit in the above aspect, we investigate the causes by analyzing data characteristics, experimental results as well as their approach design. The dataset we use is exactly the one in the paper [1], namely 98,587 bugs covering 186 components from the Mozilla's Bugzilla website. We found that there are two main disadvantages of LR-BKG which lead to the suboptimal Top-1 accuracy: (i) LR-BKG performs relatively poor on confusing component distinction. (ii) LR-BKG cannot handle the few shot components properly.

A. Limitations of the state-of-the-art toolkit

1) Poor performance on confusing component distinction: There are 19,717 test bugs covering 186 product::components. Among them, 5,061 bugs are tossed bugs (bugs having been tossed) and 14,656 bugs are non-tossed bugs (bugs initially assigned to correct components). From the experimental results of test bugs, a large number of them are mistossed into confusing components.

For tossed bugs, LR-BKG can correctly predict 2,376 bugs at the top-1 position, namely Top-1 accuracy 46.9%. There are 1,265 bugs mistossed into their confusing components. For these bugs, suggest if we could successfully assign them, i.e., if bugs mistossed to confusing components are assigned into the correct components, Top-1 accuracy will have a significant

improvement, overall from 46.9% to 71.9%. Besides the overall Top-1 accuracy, we also compute the Top-1 accuracy for tossed bugs in each component by LR-BKG, shown in Figure 1a. The green boxplot is the component-level Top-1 accuracy distribution for tossed bugs. The red one shows the ideal component-level Top-1 accuracy distribution if we could successfully handle tossed bugs mistossed into confusing components. From the comparison with these two boxplots, we can know that further distinguishing confusing components is also beneficial for the performance of bug-component triaging on each component.

For the non-tossed bugs, the Top-1 accuracy of LR-BKG is 59.3%, which is higher than tossed bugs due to the less confusing bug summary. However, there are still 2,043 non-tossed bugs mistossed into the confusing components, which means there will be a big improvement on overall Top-1 accuracy from 59.3% to 73.3% for non-tossed bugs as well, if we could distinguish confusing components better. The component-level Top-1 accuracy for non-tossed bugs is shown in Figure 1b, also indicating that the better confusing component distinction improves bug-component triaging for each component.

After investigating the failed cases on confusing components, we summarize that the main reason of this phenomenon is too coarse feature design of LR-BKG to distinguish confusing components. Since most features in LR-BKG are to calculate cosine similarities of the given bug summary and historical bug summaries in components. These cosine similarity-based features are hard to capture the subtle difference among bugs.

For example, from Section I, we know that Toolkit::Password Manager is the confusing component of Toolkit::Password Manager: Site Compatibility. Both of them are about issues of autofill, autocomplete or saving of logins, but Toolkit::Password Manager: Site Compatibility is for issues not working on the specific site. As a result, bugs in these two components tend to have similar text expression on the whole. The only difference is that bugs in Toolkit::Password Manager: Site Compatibility contain the specific site where issues happened. For example, Bug 1630553 in Toolkit::Password Manager: Site Compatibility states "Username is not captured for dismissed doorhanger on alipay.com". While another Bug 1612255 in its confusing component, Toolkit::Password Manager, says "Use username field edits to adjust the dismissed login capture doorhanger". The large proportion of common tokens (i.e., "username", "capture", "dismissed", "doorhanger") between these two bug summaries result in a relatively high cosine-similarity score. This demonstrates that they have a high probability of belonging to the same component. But they belong to different components since Bug 1630553 only happens on a specific site, i.e. "alipay.com". The associated features of LR-BKG cannot capture this subtle difference and thus mislead the prediction result.

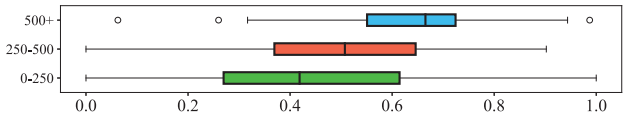


Fig. 2: Component-Level Top-1 Accuracy Distribution
Blue boxplot is for components with 500+ bugs;
Red boxplot is for components with 250-500 bugs;
Green boxplot is for components with 0-250 bugs;

In summary, too coarse-grained feature design of LR-BKG leads to the suboptimal performance on confusing component distinction. If we can solve this challenge, the performance of bug-component triaging will be improved significantly, especially Top-1 accuracy.

2) Unfriendly to few shot components: As mentioned in [7], although Mozilla has thousands of components, in the past two years, only 225 components out of 396 components had more than 49 bugs, which means the number of bugs contained in each component varies greatly. Based on this observation, we investigate bug distribution over different components and the effect of various bug distributions on bug-component triaging of LR-BKG.

We first divide the components into three different groups according to the training bug distribution of components (78,870 training bugs covering 186 components). As shown in Table I, there are 91 components contain bugs in the range of 31-250 (Group 1), namely almost half of the 186 components. For example, Core::Print Preview only has 31 bugs in total. 46 components have bugs in the range from 250 to 500 (Group 2). The top 49 components have more than 500 bugs (Group 3), such as the largest component Core::CSS Parsing and Computation having 3,762 bugs. From the data distribution, we can see that bugs are not evenly spread across different components. We further check the experimental results to investigate whether the LR-BKG can handle such skewed dataset properly. The results are shown in Fig 2.

From Fig 2, we observe that for LR-BKG, the components with more bugs tend to have a better bug-component triaging performance. For example, the median Top-1 accuracy of three groups is respectively 41.9% (Group 1), 50.7% (Group 2) and 66.5% (Group 3). Moreover, the performance of Group 3 is obviously better than Group 1 and 2. As a result, **we refer to the components with less than 500 bugs, which LR-BKG cannot handle properly as the few shot components**.

We attribute the poor performance of LR-BKG on the few-shot components to the following reason: The bug-component feature design is unfriendly for components with few bugs. In detail, LR-BKG takes Top-30 cosine similarities between the bug summary and historical bug summaries in components as features, which put the few shot components at a disadvantage, such as Core::Print Preview only having 31 historical bugs totally. Such few shot components have limited number of bugs, many dimensions of Top-30 similarities are more likely to be zero, which makes this feature less effective or even misleading. On the contrary, components with more bugs have a higher probability to be assigned.

Feature design of LR-BKG is unfriendly to few shot components, which put them at a disadvantage.

B. Challenges and potential solutions

From the above two phenomena, we can conclude that there is a gap between bug report characteristics and feature design of LR-BKG. First, manual feature design of LR-BKG is too coarse to distinguish confusing components well. Second, feature design is unfriendly to few shot components. Both of them lead to the suboptimal Top-1 accuracy. To improve the suboptimal Top-1 accuracy, the challenge is to fill the gap between the data characteristics and feature design of LR-BKG. We try to overcome it from two aspects: data itself and the feature design.

From data perspective, besides bug summary, we introduce additional information from bug reports. As bug description is the supplement of bug summary, it contains more details about the bug, such as prerequisites, steps to reproduce, expected result, actual result, etc. Therefore, we attempt to make full use of bug description for alleviating the difficulty on differentiating the confusing components. For example, although the summary of Bug 1630553 and Bug 1612255 are quite similar but the detailed bug descriptions are different. Hence, bug description can be helpful for distinguishing confusing components further.

For features, we adopt large scale pre-trained models to extract features automatically during training. Compared with hand-crafted features designed beforehand, which heavily depend on the experience and domain knowledge of the human, features from pre-trained models are learned from data automatically. Besides, pre-trained models learn deep features from the bug itself without resorting to other similar bugs, which is more friendly to few shot components. Therefore, features from pre-trained models can be more suitable for data characteristics and the specific tasks, which is a possible solution to fill the gap between data characteristics (i.e., confusing components, few shot components) and high-level feature design.

The gap between bug characteristics and manual feature design of LR-BKG leads to the suboptimal Top-1 accuracy. In this study, we introduce more information from bug (i.e. bug description) and use neural network to learn features automatically during training to rescue.

III. Background

Large scale pre-trained models based on Transformer architectures [4] have recently achieved great success and become a milestone on the natural language processing (NLP) and code-related tasks. Since they can effectively capture the semantic information from a massive amount of data automatically, it is now a very common way to adopt the pre-trained models as backbone for downstream tasks rather than learning models from scratch. From Section II, due to the gap between bug characteristics and manual feature design of LR-BKG leading to a suboptimal bug-component triaging performance, we try to use the powerful pre-trained models for capturing features automatically during training to rescue.

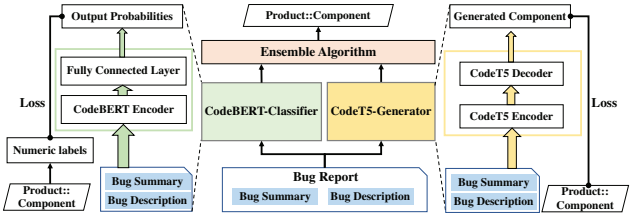


Fig. 3: Architecture of DEEPTRIAG

CodeBERT. Bug reports generally consist of natural language description mixed with code artifacts. Therefore, unlike BERT [3], RoBERTa [8], GPT [9] and XLNET [10] learning effective contextual representations from massive text for natural language processing tasks, CodeBERT [5], which is a bimodal pre-trained model for natural language (NL) and programming language (PL), is more suitable for dealing with bug reports. Following BERT [3] and RoBERTa [8], CodeBERT uses multi-layer bidirectional Transformer [4] as the model architecture. To use both bimodal instances of NL-PL pairs and unimodal codes, CodeBERT is trained with a hybrid objective function, along with standard masked language modeling [3] and replaced token detection [11]. Moreover, GraphCodeBERT [12] builds on CodeBERT by incorporating the data flow extracted from the code structure into CodeBERT. Note that code artifacts included in bug reports are usually some code snippets, identifiers, etc, instead of complete code methods. Thus, it is not necessary to capture code structure information of code artifacts from bug reports.

CodeT5. Based on the different architectures, pre-trained models based on Transformers [4] can generally be classified into three groups: encoder-only models (e.g., BERT [3], RoBERTa [8] and CodeBERT [5]), decoder-only models like GPT [9] and encoder-decoder models, such as T5 [13], CodeT5 [6]. For generation tasks, most encoder-only and decoder-only models have a suboptimal performance. Take CodeBERT [5] as an example. CodeBERT need to add an additional decoder for generation tasks, where this decoder cannot benefit from the pretraining [6]. CodeT5 [6] has the same architecture as T5 [13]. Moreover, CodeT5 uses a novel identifier-aware objective to distinguish identifiers and recover them when masked for fusing code-specific knowledge [6]. CodeT5 also leverages the code and its corresponding comments to learn a better NL-PL alignment. Above, CodeT5 is a more suitable model for generation tasks with dataset including both NL and PL.

IV. Approach

To address the aforementioned shortcomings of LR-BKG, we propose a deep-learning based model, named DEEPTRIAG. The architecture of the approach is shown in Fig. 3. DEEPTRIAG consists of three parts: the first part is a multi-classification model based on the large scale pre-trained model CodeBERT, i.e., CodeBERT-Classifier. For a given bug report

(including bug summary and bug description), the classifier quantitatively calculate the probability of the bug report belonging to each component. In the second part, we transform the bug-component triaging task into a generation task by using CodeT5 as a generator, namely CodeT5-Generator. It can generate the corresponding component according to the summary and description of the given bug report. Lastly, we ensemble the output results of the above two modules to get the final predicted component.

A. Multi-classification based Triaging

In this section, we transform bug-component triaging as a multi-classification task by regarding components as classes and classifying bug reports into their corresponding components.

Our multi-classification model is composed of two parts, the encoder and the fully connected layer. First, we transform the bug report into vector representation by the encoder. Then we project the vector representation into the dimension of components by using a fully connected layer. The final output scores represent the probability distribution of the bug report belonging to different components.

Specifically, given a bug report B, the input to encoder is the bug summary S and the bug description D, namely $\langle \mathrm{S}, \mathrm{D} \rangle$ pair. Then, the encoder maps the input into a contextualized vector h_B, where h_B denotes the embedding vector of the given bug report. To estimate the matching score between a bug report and different components, we input the embedding vector h_B into a fully connected layer, which projects h_B into the dimension of component labels [14], [15]. In other words, the linear layer is fomulated as follows:

$$\mathbf{l} = \phi\left(h_B\right) = \mathbf{a}\left(\mathbf{W}^T h_B + \mathbf{b}\right) \tag{1}$$

where $\mathbf{W}$, $\mathbf{b}$ and $\mathbf{a}$ are the weight matrix, bias vector, and activation function for the linear layer respectively. h_B is the embedding vector generated by the encoder. For a given bug report, we obtain the logits of all the component labels, $\mathbf{l} = (l_1, \ldots, l_K)$, K is the number of components. Then the logit scores of component label are normalized as follows:

$$\sigma(\mathbf{l})_i = \frac{e^{l_i}}{\sum_{j=1}^{K} e^{l_j}} \quad \text{for } i = 1, \ldots, K \tag{2}$$

where σ is the softmax function which normalizes the output scores to a probability distribution over predicted components.

Since bug reports often contain code artifacts, we adopt the pre-trained model, CodeBERT [5] as the encoder for this task to capture the better implicit semantics information from both natural and programming language. We call this multi-classification model as CodeBERT-Classifier. For a given bug report, CodeBERT-Classifier calculates the probability scores for the bug report belonging to each component.

B. Generation based Triaging

Previous work [1] indicates that component names reveal the identity of components (e.g., key functionality or concept) in a concise form, which provides useful information for bug-component triaging (e.g., from the component name, Toolikit::Password Manager, we can know it's a component on password management). However, multi-classification based triaging transforms bug-component triaging into a classification task, which regards component names as numeric labels losing the information contained in them. Thus, we use a generation task to take full advantage of the information in component names. The basic idea is to treat the bug-component triaging as a "text-to-text" task, i.e., taking bug reports as input and generating corresponding components as output.

Specifically, given the bug report B, the bug summary $S = \{s_1, \ldots, s_N\}$ consisting of N tokens and bug description $D = \{d_1, \ldots, d_M\}$ including M tokens. The component C corresponds to L tokens $\{c_1, \ldots, c_L\}$. The generation model takes the bug summary S and bug description D as input and then learn to generate the corresponding C one token at a time based on the $\langle \mathrm{S}, \mathrm{D} \rangle$ and all preceding tokens that have been generated so far, shown as follows:

$$P_\theta(\mathrm{C} \mid \langle \mathrm{S}, \mathrm{D} \rangle) = \prod_{i=1}^{L} P_\theta\left(c_i \mid c_1, \ldots, c_{i-1}; \mathrm{S}, \mathrm{D}\right) \tag{3}$$

where θ represents our generation model. $P_\theta(\mathrm{C} \mid \langle \mathrm{S}, \mathrm{D} \rangle)$ can be seen as the conditional likelihood of the component C given the $\langle \mathrm{S}, \mathrm{D} \rangle$. We train the model θ by maximizing the conditional likelihood over the training dataset. During the inference phase, the output sequences are generated by using the beam search algorithm [16] when decoding. For each given bug report B, we generate a list of components with their corresponding output scores. The output score indicates the possibility of B belonging to the component. Then, we normalize the output scores by using the softmax function, shown in Equation 2 to get the generative scores. Since components are fed into the generation model for training, the information contained in components can be learned by the model.

We use CodeT5 [6] as the generation model since CodeT5 is a unified pre-trained encoder-decoder Transformer model, which has a better performance than the encoder-only (or decoder-only) pre-trained models for generation tasks. Besides, CodeT5 exploits the user-written code comments with a bimodal dual generation task for better NL-PL alignment [6], which is more suitable for bug reports including code artifacts than T5 [13].

C. Ensemble

Considering CodeBERT-Classifier and CodeT5-Generator use different pre-trained models (i.e., CodeBERT, an encoder model and CodeT5, an encoder-decoder model) and treat the bug-component triaging as different tasks (classification and generation task), they have advantages on different cases. Thus, we use an ensemble algorithm to combine the above two modules for the better performance.

Specifically, given a bug report B, CodeBERT-Classifier outputs a set of probability scores $P = \{p_1, \ldots, p_K\}$ for

the bug report belonging to all K components. CodeT5-Generator generates a list of components with corresponding generative scores that B might belong to. Note that generating K components by CodeT5-Generator is time-consuming, we do a trade-off between the performance and the efficiency by only generating Top-10 components (due to Top-10 outputs covering the target component in most cases). For components out of the generated Top-10 components, the generative scores are set to zero. Moreover, if CodeT5-Generator outputs text not corresponding to any components, we just throw them away. After applying the above rules, we get the generative scores of all K components, namely G=$\{g_1, \ldots, g_K\}$. Then we ensemble the probability scores and generative scores by the following equation:

$$Z = \omega G + P \tag{4}$$

where Z = $\{z_1, \ldots, z_K\}$ are the final predicted scores for all K components. ω is a weight value to increase the weight of generative scores G to make G and P comparable, since generative scores are generally much smaller than probability scores. Then, all components are ranked by their final predicted scores. Finally, we recommend the component with the highest score as the prediction result and triage the bug report into the component.

V. Evaluation

This section answers the following three research questions:

- RQ1: How effective is our approach in bug-component triaging?
- RQ2: Is bug description useful for improving the component recommendation?
- RQ3: How effective are modules in DeepTriag for bug-component triaging?

A. Experimental Setup

1) Dataset: To thoroughly evaluate the performance and verify the generalization of DeepTriag, we use two datasets from different open source software projects, Mozilla and Eclipse. LR-BKG has been evaluated on the dataset from Mozilla in the work [1]. To make a fair comparison with LR-BKG, we use the same dataset and splitting strategy as LR-BKG [1] on Mozilla, which contains 98,587 closed bugs covering 6 products and 186 components. 29,100 out of these are tossed bugs. In particular, to simulate real-world context, the dataset is divided into 80% and 20% according to chronological order. Finally, we obtain 78,870 bugs (including 24,039 tossed bugs) with creation time before 25th February, 2020 as "historical" training data. The rest 19,717 bugs (including 5,061 tossed bugs) are "future" testing dataset. Besides the Mozilla dataset, to verify the generalization of our model, we also collect dataset from Eclipse, namely 198,416 closed bugs (38,673 tossed bugs and 159,743 non-tossed bugs) from 6 products and 46 components. We use the same splitting strategy as Mozilla to get 158,733 train data (33,070 tossed bugs) and 39,683 test data (5,603 tossed bugs).

(a) Mozilla

Tool	Category	Top-1	Top-3	Top-5	Top-10	MRR
BugBug	Tossed	0.378	0.608	0.680	0.760	-
	Non-Tossed	0.468	0.642	0.697	0.764	-
	Overall	0.445	0.633	0.692	0.763	-
LR-BKG	Tossed	0.469	0.701	0.772	0.848	0.608
	Non-Tossed	0.593	0.779	0.836	0.892	0.702
	Overall	0.562	0.759	0.820	0.881	0.678
DeepTriag	Tossed	0.554	0.764	0.819	0.881	0.675
	Non-Tossed	0.727	0.857	0.890	0.930	0.802
	Overall	0.683	0.833	0.872	0.917	0.769

(b) Eclipse

Tool	Category	Top-1	Top-3	Top-5	Top-10	MRR
LR-BKG	Tossed	0.486	0.731	0.807	0.879	0.629
	Non-Tossed	0.517	0.711	0.776	0.849	0.637
	Overall	0.513	0.714	0.780	0.853	0.636
DeepTriag	Tossed	0.571	0.820	0.882	0.942	0.710
	Non-Tossed	0.652	0.839	0.883	0.933	0.758
	Overall	0.641	0.836	0.883	0.935	0.751

TABLE II: Top-k Accuracy & MRR

2) Evaluation Metrics: In this work, our aim is to improve the suboptimal performance for recommending the correct component. Therefore, we adopt the widely-accepted evaluation metrics, **Top-k accuracy** and **Mean Reciprocal Rank** (MRR) to measure the performance of our model. Top-k accuracy is $\sum_{\mathrm{b_i \in B}} \mathrm{isCorrect}(b_i, \text{Top-k})/|B|$ where B represents the set of all test bugs and we set the $\mathrm{isCorrect}(b_i, \text{Top-k})$ function return 1 if Top-k components contain the target component to the input bug b_i; and return 0 otherwise. MRR measures a recommender system's performance based on the graded relevance of the recommended items and their positions in the candidate set, i.e., $1/|B| \sum_{i=1}^{|B|} 1/rank_i$ where B is the set of test bugs and $rank_i$ refers to the rank position of the correct component for the i-th bug.

3) Baselines: We compare our approach with the following baselines: the first baseline is BugBug [2], which has been developed and used by Mozilla for bug management. In particular, we use the component classifier within BugBug tool for this bug-component triaging task. It transforms bug-component triaging into the multi-classification task by using the logistic regression model with features collected from bug summary, description, keywords/flags. The second one is LR-BKG. Details are shown in Section II.

4) Experimental Settings: DeepTriag is implemented by the PyTorch framework. We use the "codebert-base" model for CodeBERT-Classifier and "codet5-base" model for CodeT5-Generator. These two tasks are trained individually. During the fine-tuning procedure, we optimize the parameters by using AdamW [17], with the learning rate for CodeBERT Classifier 2e-5 and for CodeT5-Generator 5e-5. Both of the training procedure last 10 epochs. For ensembling settings, ω is set to 5. All model training and experiments were done on a GPU machine (one GeForce RTX 3080 GPU with 10GB memory, 10 cores 3.7GHZ CPU, 32GB memory).

TABLE III: Component-Level Top-1 Accuracy Comparison

Top-1 Accuracy	DEEPTRIAG > LR-BKG	DEEPTRIAG = LR-BKG	DEEPTRIAG < LR-BKG
# P::C (Mozilla)	165	11	10
# P::C (Eclipse)	37	8	1

B. Bug Triaging Effectiveness Evaluation (RQ1)

In this research question, we evaluate the overall effectiveness of our approach compared with the above baselines. The experimental results regarding Top-k accuracy and MRR are shown in Table II. Note that BugBug is designed for Mozilla, which is difficult to migrate to other projects. Therefore, we only get the experimental results of BugBug on Mozilla. Besides, the MRR of BugBug can not be estimated because some prediction results of BugBug are out of the components in the dataset. From the table, **it is obvious that our approach, DEEPTRIAG, is substantially better than the above baselines regarding all Top-k accuracies and MRR on both dataset**, which demonstrates the effectiveness and generalization of our approach for this bug-component triaging task.

BugBug has the worst performance on Mozilla dataset. As for Top-1 accuracy, BugBug only achieved 37.8% for tossed bugs and 46.8% for non-tossed bugs and overall 44.5%, much lower than LR-BKG and our approach. Similar to the CodeBERT-Classifier of DEEPTRIAG, BugBug also transforms bug-component triaging into multi-classification task. The major difference is BugBug takes the one-hot representation as features from bug summary, description and keywords/flags, while DEEPTRIAG automatically learns the deep features from the bug report. The one-hot features utilized by BugBug can only capture the lexical-level information, which is unable to learn the semantic links between bugs and components. This may explain its poor performance on the bug-component triaging task.

Compared with LR-BKG, our approach improves Top-1 accuracy from 46.9% to 55.4% for tossed bugs, from 59.3% to 72.7% for non-tossed bugs and overall from 56.2% to 68.3% on Mozilla dataset. On Eclipse dataset, Top-1 accuracy is improved from 48.6% to 57.1% for tossed bugs, from 51.7% to 65.2% for non-tossed bugs and overall from 51.3% to 64.1%. We attribute this to the following reasons: Firstly, instead of using manual feature engineering, we use the pre-trained models to extract deep features from bug reports automatically during training. The generated vector representation can effectively capture the semantic information and implicit connections between bugs and components in a finer granularity. Secondly, in addition to bug summary, we introduce bug description for obtaining more information to distinguish confusing components further. We thoroughly discuss the effect of bug description on improving Top-1 accuracy in Section V-C.

We also conduct the component-level evaluation of DEEPTRIAG and LR-BKG in terms of Top-1 accuracy, shown in Table III. DEEPTRIAG achieves the higher Top-1 accuracy than LR-BKG for 165 out of 186 components indicating

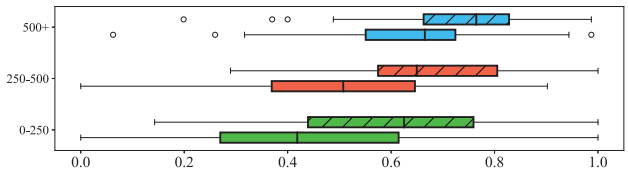


Fig. 4: Top-1 Accuracy Distribution of Components (DEEPTRIAG vs LR-BKG on Mozilla)
Blue boxplot is for components with 500+ bugs (Group 3);
Red boxplot is for components with 250-500 bugs (Group 2);
Green boxplot is for components with 0-250 bugs (Group 1);
Notes: DEEPTRIAG corresponds to the boxplots with slash;

that the performance of DEEPTRIAG is better than LR-BKG on most components (88.7%). For Eclipse, DEEPTRIAG performs better on 37 out of 46 components (80.4%) than LR-BKG. Only one component has lower Top-1 accuracy. The superior performance of DEEPTRIAG on both bug-level and component-level further justifies the advantage of our model.

As discussed in Section II, LR-BKG obtains a relatively poor Top-1 accuracy. We identify two major situations that LR-BKG fails to handle properly, the confusing component (cf. Section II-A1) and the few-shot component (cf. Section II-A2). Regarding the limitations of LR-BKG, we investigate whether our model, DEEPTRIAG, can better handle the confusing component and few-shot component challenges. Take the Mozilla dataset as an example. The details are as follows:

Confusing component. Failing to distinguish confusing components effectively is one of the main reasons for the suboptimal Top-1 accuracy of LR-BKG. Hence, we analyze the experimental results to find out whether DEEPTRIAG can help to alleviate this situation. Considering 5,061 tossed bugs, there are 1,265 bugs mistossed into confusing components by LR-BKG. With the help of DEEPTRIAG, 323 out of 1,265 mistossed bugs can be tossed into correct components. As for 14,656 non-tossed bugs, LR-BKG mistosses 2,043 bugs into confusing components. DEEPTRIAG assigns 975 out of 2,043 bugs into the correct ones. Due to the valuable information within the bug description and the deep features effectively extracted by the pre-trained models, DEEPTRIAG can effectively reduce the number of failure cases on confusing components when performing the bug-component triaging. However, the confusing component distinction is still an open research question need to be further investigated, since some bugs still remain in the confusing components.

Few shot component. To verify whether DEEPTRIAG can better handle the few shot components than LR-BKG, we recalculate the Top-1 accuracy distribution of the three different component groups (referred to Table I), as shown in Figure 4. For a comparison purpose, we put the results from LR-BKG (without slash) and DEEPTRIAG (with flash) together. From Figure 4, we can observe that DEEPTRIAG effectively improves bug triaging on few shot components on Mozilla dataset. For Group 1, the median of Top-1 accuracy increases from 41.9% to 62.5%, an obvious improvement compared with LR-BKG. The median of Top-1 accuracy for Group

(a) Mozilla

Tool	Category	Top-1	Top-3	Top-5	Top-10	MRR
LR-BKG	Tossed	0.469	0.701	0.772	0.848	0.608
	Non-Tossed	0.593	0.779	0.836	0.892	0.702
	Overall	0.562	0.759	0.820	0.881	0.678
LR-BKG+BD	Tossed	0.440	0.680	0.757	0.839	0.584
	Non-Tossed	0.542	0.753	0.814	0.880	0.665
	Overall	0.516	0.734	0.799	0.869	0.644
LRBKG-BPE+BD	Tossed	0.395	0.622	0.695	0.787	0.536
	Non-Tossed	0.497	0.688	0.749	0.818	0.613
	Overall	0.471	0.671	0.735	0.810	0.593
DEEPTRIAG-BD	Tossed	0.529	0.723	0.782	0.845	0.645
	Non-Tossed	0.659	0.805	0.844	0.890	0.745
	Overall	0.626	0.784	0.828	0.879	0.719
DEEPTRIAG	Tossed	0.554	0.764	0.819	0.881	0.675
	Non-Tossed	0.727	0.857	0.890	0.930	0.802
	Overall	0.683	0.833	0.872	0.917	0.769

(b) Eclipse

Tool	Category	Top-1	Top-3	Top-5	Top-10	MRR
LR-BKG	Tossed	0.486	0.731	0.807	0.879	0.629
	Non-Tossed	0.517	0.711	0.776	0.849	0.637
	Overall	0.513	0.714	0.780	0.853	0.636
LR-BKG+BD	Tossed	0.481	0.750	0.836	0.909	0.635
	Non-Tossed	0.549	0.762	0.828	0.894	0.675
	Overall	0.539	0.761	0.829	0.897	0.669
LRBKG-BPE+BD	Tossed	0.435	0.722	0.808	0.890	0.599
	Non-Tossed	0.507	0.719	0.789	0.869	0.636
	Overall	0.497	0.720	0.792	0.872	0.631
DEEPTRIAG-BD	Tossed	0.526	0.767	0.841	0.920	0.666
	Non-Tossed	0.598	0.779	0.836	0.904	0.707
	Overall	0.588	0.777	0.837	0.906	0.701
DEEPTRIAG	Tossed	0.571	0.820	0.882	0.942	0.710
	Non-Tossed	0.652	0.839	0.883	0.933	0.758
	Overall	0.641	0.836	0.883	0.935	0.751

TABLE IV: Top-k Accuracy & MRR (+/- Bug Description)

2 rises from 50.7% to 65.0%. Moreover, DEEPTRIAG can also improve the performance of components with more than 500 bugs in general. The median Top-1 accuracy improves from 66.5% to 76.5%. The handcrafted features designed by LR-BKG heavily rely on the number of similar bugs and how similar these bugs to the given bug. The very limited number of bugs within the few shot components makes the bug-component features of LR-BKG less effective. Compared with LR-BKG, for a given bug report, DEEPTRIAG learns the deep features from the bug itself without resorting to other similar bugs, which can explain the reason why DEEPTRIAG can better handle the few shot components.

We conclude that our approach can significantly improve the Top-1 accuracy over the state-of-the-art baseline by a large margin, which shows the effectiveness of our approach to assign bugs into the target components at the first position directly and accurately. Moreover, DEEPTRIAG *can better handle the situation of confusing components and few shot components compared with LR-BKG.*

C. Bug Description Effectiveness (RQ2)

One advantage of DEEPTRIAG is to introduce bug description into bug-component triaging. To investigate the impact of bug description on bug-component triaging, we conduct the bug description analysis, as follows:

LR-BKG+BD. Since LR-BKG does not consider bug description as input, for a fair comparison, we further incorporate bug description to LR-BKG. Specifically, we preprocess the bug description using the same way as text preprocessing of LR-BKG, namely using camel case to split words, converting words into lowercase, then splitting paragraphs into sentences by punctuations, tokenizing and stemming them by the NLTK. The only difference from LR-BKG is that we filter out tokens in bug description appearing less than 15 times to remove noises to avoid the dimension explosion of one-hot representation. After text preprocessing, we concatenate bug summary and bug description as a whole as input instead of inputting bug summary only.

LRBKG-BPE+BD Inspired by the Byte-Pair Encoding (BPE) [18] that pre-trained models use, we evolve LR-BKG with BPE instead of word-based tokenization to alleviate the large vocabulary size issue. That is, we replace the text preprocessing of LR-BKG with the BPE.

DEEPTRIAG-BD. DEEPTRIAG-BD drops bug description from the input of DEEPTRIAG, i.e., only considering the bug summary as input.

The experimental results are shown in Table IV. To measure the improvement achieved by adding bug description as input for our approach, we compare the Top-1 accuracy of DEEPTRIAG with DEEPTRIAG-BD. The experimental results on Mozilla dataset are shown in Table IVa. For tossed bugs, DEEPTRIAG increases the Top-1 accuracy from 52.9% to 55.4%. For non-tossed bugs, it improves from 65.9% to 72.7%. Overall, it reaches 68.3% from 62.6%. As for experimental results on Eclipse dataset in Table IVb, the Top-1 accuracy increases from 52.6% to 57.1% for tossed bugs, from 59.8% to 65.2% for non-tossed bugs and from 58.8% to 64.1% overall. From the results, it is clear that bug description does make contributions to the performance of DEEPTRIAG for both tossed and non-tossed bugs, which proves that bug description indeed contains useful information for bug-component triaging. **Therefore, it is useful and necessary to introduce bug description for further improving the performance of bug-component triaging.**

Referring to [1], LR-BKG only takes bug summary as input. Considering that bug description is helpful for DEEPTRIAG, we further investigate if bug description can boost the performance of LR-BKG. However, as shown in Table IVa, on Mozilla dataset, the Top-1 accuracy of LR-BKG does not rise but falls from 46.9% to 44.0% and from 59.3% to 54.2% for tossed and non-tossed bugs surprisingly. Overall, it drops from 56.2% to 51.6%. On the contrary, the Top-1 accuracy of LR-BKG increases from 51.3% to 53.9% overall on Eclipse dataset (from 48.6% to 48.1% and from 51.7% to 54.9% for tossed and non-tossed bugs) from Table IVb. To analyze the performance of LR-BKG after adding bug descriptions, we further investigate the characteristics of bug description as well as how LR-BKG uses bug description in detail.

By analyzing bug description, we find that, even though bug description contains detailed and valuable information (preconditions, steps to reproduce, expected/actual result, etc.), it also involves much noisy and useless information (e.g., various user free-form expressions). The valuable information can be easily buried in a large amount of irrelevant data.

For example, considering some LR-BKG's manually designed features calculated based on one-hot vector, the dimension of the vector sharply increased from 20k (based on bug summaries only) to 759k (after adding the bug descriptions) on Mozilla dataset. It is worth mentioning that we filter tokens from bug description appearing no more than 15 times to remove the meaningless tokens. Then, the vector dimension remains 31k. Even so, LR-BKG still cannot take advantage of bug description information on Mozilla dataset. As a result, the large amount of noise and irrelevant information makes the feature vector of LR-BKG less effective, which is the reason why the performance of LR-BKG+BD drops significantly on Mozilla dataset.

Compared with Mozilla dataset, bug description on Eclipse dataset contains fewer meaningless tokens, since the vector dimension is increased from 20k to 229k when adding the bug description, far less than 759k in Mozilla dataset. After filtering tokens appearing no more than 15 times, it remains 26k. Thus, based on more valuable information mixed with less meaningless tokens, the overall Top-1 accuracy increases from 51.3% to 53.9%. But compared to DEEPTRIAG (64.1%), LR-BKG+BD still fails to make full use of bug description.

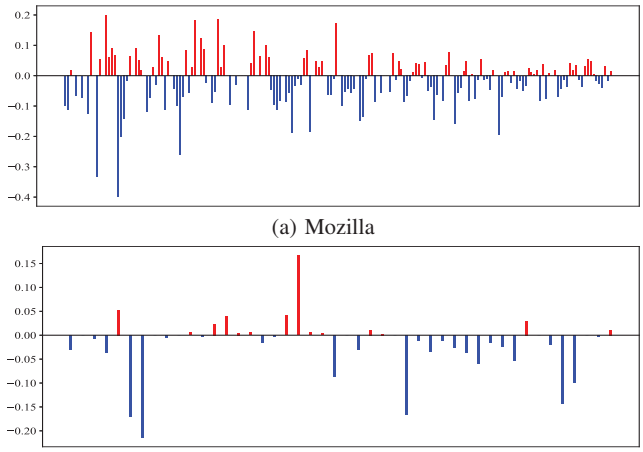
Moreover, for the dimension explosion issue of LR-BKG after adding the bug description, we try to optimize LR-BKG+BD with BPE, namely LRBKG-BPE+BD. Tokenizing bug summary and description by using BPE can effectively reduce the vector dimension from 759k to 27k for Mozilla and from 229k to 26k for Eclipse by setting the maximum vocabulary size to 30k. But the experimental results of LRBKG-BPE+BD from Table IV show that it even has the worse performance than LR-BKG+BD, which shows the BPE tokenization algorithm is not suitable for traditional machine learning approaches.

The performance of LR-BKG after adding bug description indicates that, the hand-crafted features of LR-BKG are sensitive to the data noise, and difficult to capture the valuable information from a large amount of meaningless data. Compared with LR-BKG, the deep features learned by DEEPTRIAG are more robust and can extract useful information from bug description automatically and effectively, which can further boost the bug-component triaging performance.

Bug description contains valuable information for bug-component triaging, but it is mixed with much irrelevant noise, which is difficult for manual feature engineering to extract useful information from it. DEEPTRIAG*, which uses the pre-trained models to extract deep features from bug description automatically, can make full use of bug description and effectively boost the performance of bug-component triaging.*

D. Ablation on DEEPTRIAG *(RQ3)*

DEEPTRIAG consists of CodeBERT-Classifier, CodeT5-Generator and an ensemble algorithm. In this section, we conduct an ablation study to evaluate the effectiveness of them. The results of ablation study are shown in Table V.



(a) Mozilla

(b) Eclipse

Fig. 5: CodeBERT-Classifier VS CodeT5-Generator
X-axis represents components; Y-axis is the difference of Top-1 accuracy between CodeBERT-Classifier and CodeT5-Generator; Red lines are CodeBERT-Classifier better than CodeT5-Generator; Blue lines are on the contrary.

From Table V, the individual CodeBERT-Classifier or CodeBERT-Generator can achieve much better performance than BugBug and LR-BKG regarding all Top-1 accuracies and MRR, which demonstrates the effectiveness of deep features extracted from pre-trained models. Comparing CodeBERT-Classifier with CodeT5-Generator, CodeT5-Generator has better performance than CodeBERT-Classifier on Top-1 accuracy. But, as k gradually increases, CodeBERT-Classifier, more often than not, exhibits higher Top-k accuracy than CodeT5-Generator. Instead of treating components as numeric labels like CodeBERT-Classifier, CodeT5-Generator takes components as input during training. Thus, CodeT5-Generator can learn more information from components which is useful for the bug-component triaging. However, the disadvantage of CodeT5-Generator is that it can generate outputs not corresponding to any of possible components. According to our observation, we found that given a bug report, the first generated output is always meaningful while the following generated ones sometimes are out of the range of possible components. Therefore, CodeT5-Generator has an advantage in Top-k accuracy when k is small. CodeBERT-Classifier performs better as k increasing.

Moreover, we discover that CodeBERT-Classifier and CodeT5-Generator make their own contributions to different components. In other words, they can complement and enhance the performance of each other on component level, as shown in Figure 5 intuitively. As for Mozilla dataset (Figure 5a), only 29 components out of 186 components have the same performance for CodeBERT-Classifier and CodeT5-Generator. CodeBERT-Classifier performs better than CodeT5-Generator on 69 components, while performing weaker than CodeT5-Generator on 88 components. From Figure 5b, on Eclipse dataset, CodeBERT-Classifier has the same perfor-

(a) Mozilla

Tool	Category	Top-1	Top-3	Top-5	Top-10	MRR
CodeBERT Classifier	Tossed	0.535	0.719	0.771	0.853	0.648
	Non-Tossed	0.700	0.833	0.866	0.902	0.776
	Overall	0.658	0.804	0.842	0.890	0.743
CodeT5 Generator	Tossed	0.546	0.730	0.775	0.818	0.650
	Non-Tossed	0.707	0.833	0.862	0.889	0.777
	Overall	0.666	0.807	0.840	0.871	0.744
DEEPTRIAG	Tossed	0.554	0.764	0.819	0.881	0.675
	Non-Tossed	0.727	0.857	0.890	0.930	0.802
	Overall	0.683	0.833	0.872	0.917	0.769

(b) Eclipse

Tool	Category	Top-1	Top-3	Top-5	Top-10	MRR
CodeBERT Classifier	Tossed	0.550	0.796	0.862	0.925	0.689
	Non-Tossed	0.635	0.818	0.866	0.920	0.741
	Overall	0.623	0.815	0.865	0.920	0.733
CodeT5 Generator	Tossed	0.577	0.727	0.762	0.816	0.667
	Non-Tossed	0.650	0.764	0.788	0.828	0.721
	Overall	0.639	0.758	0.784	0.827	0.713
DEEPTRIAG	Tossed	0.571	0.820	0.882	0.942	0.710
	Non-Tossed	0.652	0.839	0.883	0.933	0.758
	Overall	0.641	0.836	0.883	0.935	0.751

TABLE V: Ablation Study Evaluation

mance as CodeT5-Generator on 7 components, worse than CodeT5-Generator on 25 components and better than CodeT5-Generator on 14 components. This phenomenon indicates that CodeBERT-Classifier and CodeT5-Generator have their own advantage on different components. We attribute it to the differences in model architectures and learning objectives of these two modules.

Based on above findings, we combine CodeBERT-Classifier and CodeT5-Generator for making full use of their individual advantages on Top-k accuracy and their component-level differences. From Table V, it is obvious that our approach, DEEPTRIAG, outperforms CodeBERT-Classifier and CodeT5-Generator individually. The superior performance of DEEPTRIAG indicates that CodeBERT-Classifier and CodeT5-Generator can complement each other for better performance of bug-component triaging, which verifies the effectiveness and necessity of employing the ensemble strategy.

Each individual module in DEEPTRIAG *is effective and helpful for bug-component triaging. Moreover, CodeBERT-Classifier and CodeT5-Generator greatly complement and enhance with each other on Top-k accuracy and component-level performance.*

VI. THREATS TO VALIDITY

Threats to internal validity relate to the wrong implementation of our code. To reduce errors in our code, we have double checked and fully tested our code, still there could be errors that we did not notice. To reduce the impact of undetected errors in our code, we also publish our source code and dataset to enable other researchers for replicating and extending our work.

Threats to external validity are about the generalization of our approach. We evaluate our approach on bug reports from Mozilla and Eclipse. Both of the projects build on the bugzilla, a general-purpose bug management platform, which might reduce the diversity of our dataset. However, we use the dataset from Mozilla involving six products and 186 components and the dataset from Eclipse covering six products and 46 components, which mentions diverse frontend and backend features and alleviates the threats to external validity to a certain extent. We plan to evaluate our approach on the dataset from other bug tracking system, such as GitHub in the future.

Threats to construct validity relate to the suitability of our evaluation metrics. The metrics we use contain Top-k accuracy and Mean Reciprocal Rank (MRR), which are widely used for evaluating the performance of recommendation [19].

VII. RELATED WORK

To reduce the burden of bug management and facilitate the process of bug fixing, an amount of software engineering research has been invested into automatic bug report management techniques, such as bug localization [20], [21], duplicate bug detection [22], [23], bug field assignment [24], [25], etc. Bug field assignment includes many aspects, e.g. the severity [26]–[29] and priority [30]–[32] labels of bug reports, bug triaging [24], [25], [33], [34]. Among them, bug triaging among components is the most relevant research to our work.

Bug triaging contains two aspects, namely assigning bugs among assignees and assigning bugs among components. Most work currently focuses on bug-assignee triaging for assigning bugs to appropriate experts. Anvik et al. [33] used the SVM (Support Vector Machine) to assign the bug to the right bug fixer. Jeong et al. [34] constructed a probability bug tossing graph based on developer's tossing relationships to improve the performance of bug fixer assignment. Bhattacharya et al. [35], [36] extended Jeong's work by adding multi-feature to the bug tossing graph to improve the accuracy of bug assignments among developers. Xia et al. [37] proposed a model, named DevRec, to automatically recommend developers for bug resolution from bug report based analysis and developer based analysis. Bader et al. [38] proposed a learning to rank approach that ranks the top developers for a given bug report.

Compared with bug fixer assignment, the research focused on the task of bug-component triaging is limited. Somasundaram et al. [24] used the topic model to predict bug-component for a given bug. Sureka [25] performed the bug-component prediction by the Naive Bayes classifier with TFIDF features and a dynamic language model classifier. BugBug, which has already been used in the Mozilla's development, used the logistic regression model built in XGBoost [39] to perform the bug-component triaging. Most recently, Su et al. [1] proposed the state-of-the-art approach, named LR-BKG, for the bug-component triaging. LR-BKG improves the performance of bug-component triaging by learning from mistakes with a bug tossing knowledge graph. Although LR-BKG has significantly improved the performance of bug-component triaging, we found that it cannot handle the confusing component and few-shot component situation effectively. To address the limitations of the LR-BKG, in this study, we proposed a deep-learning based model, DEEPTRIAG, to assign

the bug report to the right component with much higher Top-1 accuracy.

VIII. Conclusions and Future Work

This paper investigates the limitations of the state-of-the-art toolkit, LR-BKG for bug-component triaging. We find the gap between characteristics of bug reports and manual feature design of LR-BKG leading to the suboptimal performance for bug assignment. Therefore, we propose a novel approach, DeepTriag by using large pre-trained models to extract features from bug reports automatically to fill this gap and then ensemble the results from CodeBERT-Classifier and CodeT5-Generator to get the final outcome. Experimental results demonstrate the effectiveness of DeepTriag on bug-component triaging, especially for Top-1 accuracy rising from 56.2% to 68.3% on Mozilla dataset and from 51.3% to 64.1% on Eclipse dataset. In the future, we will further distinguish confusing components by combining more kinds of information besides text (e.g. video, screenshots, gif, etc.) and strengthen few shot components by using siamese or even triplet network.

ACKNOWLEDGMENT

This research is partially supported by the Shanghai Rising-Star Program (23YF1446900) and the National Science Foundation of China (No. 62202341).

References

[1] Y. Su, Z. Xing, X. Peng, X. Xia, C. Wang, X. Xu, and L. Zhu, "Reducing bug triaging confusion by learning from mistakes with a bug tossing knowledge graph," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 191–202.

[2] S. L. Marco Castelluccio, "bugbug," GitHub, https://github.com/mozilla/bugbug.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[6] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[7] S. L. Marco Castelluccio, "bugbugwebsite," Mozilla Hacks, https://hacks.mozilla.org/2019/04/teaching-machines-to-triage-firefox-bugs/.

[8] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[9] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[10] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.

[11] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[13] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[14] A. Adhikari, A. Ram, R. Tang, and J. Lin, "Docbert: Bert for document classification," *arXiv preprint arXiv:1904.08398*, 2019.

[15] M. Munikar, S. Shakya, and A. Shrestha, "Fine-grained sentiment classification using bert," in *2019 Artificial Intelligence for Transforming Business and Society (AITB)*, vol. 1. IEEE, 2019, pp. 1–5.

[16] P. Koehn, "Pharaoh: a beam search decoder for phrase-based statistical machine translation models," in *Conference of the Association for Machine Translation in the Americas*. Springer, 2004, pp. 115–124.

[17] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[18] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[19] X. Liu, L. Huang, and V. Ng, "Effective api recommendation without historical software repositories," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 282–292.

[20] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[21] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.

[22] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 45–54.

[23] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 461–470.

[24] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceeding of the 5th Annual India Software Engineering Conference, ISEC 2012, Kanpur, India, February 22-25, 2012*, S. K. Aggarwal, T. V. Prabhakar, V. Varma, and S. Padmanabhuni, Eds. ACM, 2012, pp. 125–130. [Online]. Available: https://doi.org/10.1145/2134254.2134276

[25] A. Sureka, "Learning to classify bug reports into components," in *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, ser. Lecture Notes in Computer Science, C. A. Furia and S. Nanz, Eds., vol. 7304. Springer, 2012, pp. 288–303. [Online]. Available: https://doi.org/10.1007/978-3-642-30561-0_20

[26] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 1–10.

[27] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 346–355.

[28] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 215–224.

[29] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 249–258.

[30] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 261–270.

[31] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 200–209.

[32] Q. Umer, H. Liu, and I. Illahi, "Cnn-based automatic prioritization of bug reports," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1341–1354, 2019.

[33] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association

for Computing Machinery, 2006, p. 361–370. [Online]. Available: https://doi.org/10.1145/1134285.1134336

[34] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, H. van Vliet and V. Issarny, Eds. ACM, 2009, pp. 111–120. [Online]. Available: https://doi.org/10.1145/1595696.1595715

[35] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[36] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2275–2292, 2012.

[37] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 72–81.

[38] B. Alkhazi, A. DiStasi, W. Aljedaani, H. Alrubaye, X. Ye, and M. W. Mkaouer, "Learning to rank developers for bug report assignment," *Applied Soft Computing*, vol. 95, p. 106667, 2020.

[39] "xgboost," GitHub, https://github.com/dmlc/xgboost.