Deep Learning Based Valid Bug Reports Determination and Explanation

Jianjun He[1,2], Ling Xu[1,2*], Yuanrui Fan[3], Zhou Xu[1,2], Meng Yan[1,2], Yan Lei[1,2]
[1]Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China
[2]School of Big Data and Software Engineering, Chongqing University, Chongqing, China
{jjhe, xuling, zhouxullx, mengy, yanlei}@cqu.edu.cn
[3]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
yrfan@zju.edu.cn

***Abstract*—Bug reports are widely used by developers to fix bugs. Due to the lack of experience, reporters may submit numerous invalid bug reports. Manually determining valid bug reports is a laborious task. Automatically identifying valid bug reports can save time and effort for bug analysis. In this paper, we propose a deep learning-based approach to determine and explain valid bug reports using only textual information i.e., summaries and descriptions of bug reports. Convolutional neural network (CNN) is applied to capture their contextual and semantic features. Moreover, by analyzing the spatial structure of CNN, we backtrack the trained CNN model to get phrases that can explain valid bug reports determination. After inspecting the phrases manually, we summarize some valid bug report patterns. We evaluate our approach on five large-scale open-source projects containing a total of 540491 bug reports. On average, across the five projects, our approach achieves 0.85, 0.80, 0.69 and improves the state-of-the-art approach by 8.97%, 9.59%, 9.52% in terms of AUC, F1-score for valid bug reports, and F1-score for invalid bug reports, respectively. From the summarized patterns, we can find that determining valid bug reports is mainly due to three categories of patterns: Attachment, Environment, and Reproduce.**

***Index Terms*—Valid Bug Report, Deep Learning, Model Explainability, Software Quality Assurance**

I. Introduction

Large-scale software systems use bug tracking systems (such as Bugzilla and JIRA) to report and manage bug reports [1]. Developers, testers or end-users submit bug reports when they encounter bugs. Bug reports can help to guide software maintenance and repair activities [2], [3]. In the process of fixing bugs, bug triagers first manually determine the validity of a bug report. If this bug report is valid, it is assigned to an appropriate fixer. A valid bug is defined as a real bug that contains complete information and can be reproduced [4]. For large software projects with many users, hundreds of bug reports are submitted to the bug tracking system every day [5]. However, many of these bug reports are invalid. For example, in Eclipse, about 70% of bug reports are invalid [4]. Manually determining the validity of so many bug reports is a laborious and challenging task. Approaches to automatically determine the validity of bug reports early can significantly reduce the cost of software management and maintenance.

Every bug report has a resolution. The resolution of a bug report can be *DUPLICATE* (i.e., a bug that has been reported by other reporters), *INVALID* (i.e., a bug that is not a software defect), *WORKSFORME* (i.e., a bug that cannot be reproduced), *INCOMPLETE* (i.e., a bug that lacks necessary information), *FIXED* (i.e., a bug that has been successfully fixed), or *WONTFIX* (i.e., a real bug that has not been fixed). Following prior study [4], we consider a bug report whose resolution is *FIXED* or *WONTFIX* as a valid bug report, and we consider a bug report whose resolution is *DUPLICATE*, *INVALID*, *WORKSFORME* or *INCOMPLETE* as an invalid bug report.

In the literature, several approaches have been proposed to address this issue. Zanetti et al. [6] focused on the links between reporters. They used collaborative networks on bug reports from open-source projects. Based on ASSIGN (i.e., assigned fixer of bug report) and CC (i.e., email address of fixer) relationship of bug reports, they built a collaboration network. And then, they extracted nine features from this network and applied a support vector machine (SVM) classifier to determine whether a bug report is valid. In comparison with Zanetti et al., Fan et al. [4] extracted 33 features from bug reports, including the collaboration network, reporter experience, completeness, readability, and text. Based on these features, they used a random forest (RF) classifier to identify valid bug reports.

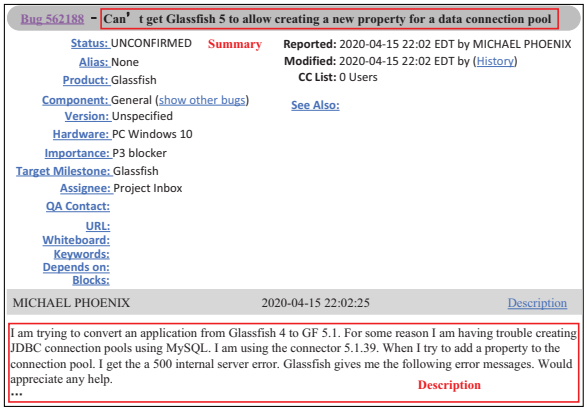


Fig. 1: A newly reported bug report.

These approaches are based on feature engineering, which transforms the original data into features and then feeds these

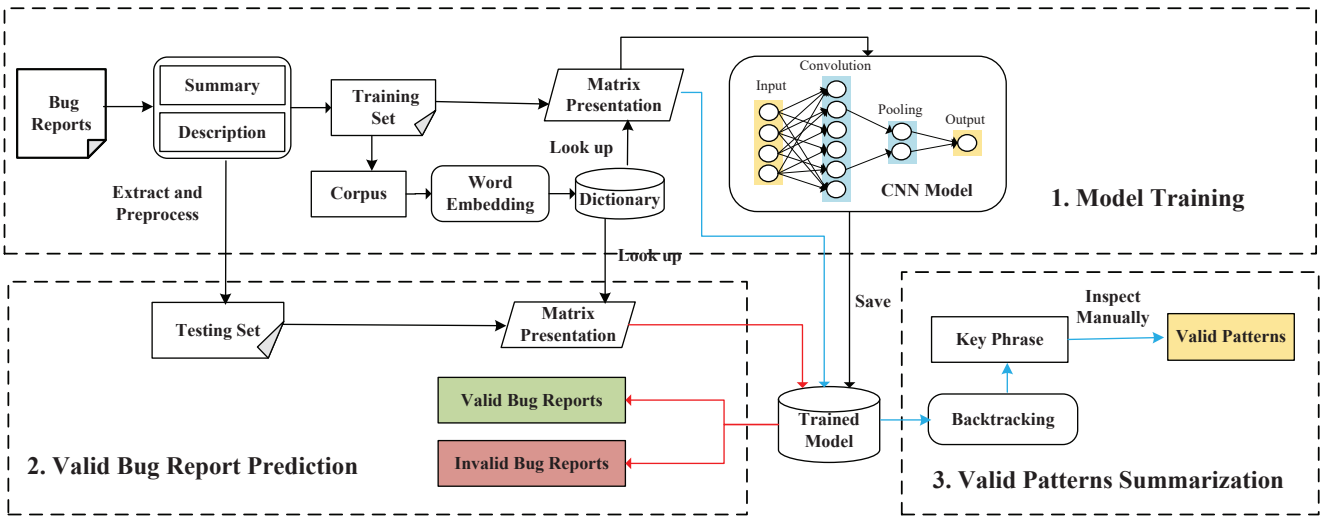


Fig. 2: The framework of our approach.

features into classifiers. However, there are some problems in extracting these features. For example, a newly reported bug report does not contain complete fields to help extract various features. Figure 1 shows an example of a bug report in Eclipse. When we found this bug report, it had been reported for three days. The CC field required by Zanetti et al. was not yet available. In practice, we hope to identify the validity of a bug report as soon as possible after it is reported. And this newly reported bug report had not been commented by others. Thus, Fan et al.'s approach cannot extract the relevant features of the collaborative network. Therefore, the approaches based on feature engineering may not be able to determine the validity of a newly reported bug report in time.

However, as can be seen from Figure 1, a newly reported bug report always includes a summary and a description. Moreover, the experimental results of Fan et al. showed that text-based features are beneficial [4]. Motivated by these observations, we explore the idea of using only textual information to identify valid bug reports. In recent years, many studies have shown that convolutional neural network (CNN) has excellent performance in text classification [7], [8]. We attempt to employ a CNN model to extract the contextual and semantic features from the textual information of bug reports.

Related researches show that high-quality bug reports usually contain some specific descriptions, such as additional attachments and steps to reproduce [9]. Due to the high correlation between the validity of a bug report and its quality, we suspect that a valid bug report may contain some valid patterns that can be used to identify valid bug reports. A model based on deep learning is usually considered as a black box [10]–[12], and its performance is generally excellent but difficult to explain. However, this black box can be opened. In this paper, we use the spatial structure of CNN to locate some key phrases that make bug reports determined to be valid in a layer-by-layer backtracking strategy, and finally summarize some valid bug report patterns. To the best of our knowledge, this is the first study to explore valid bug report patterns.

The main contributions of this paper are as follows:

1) We propose a deep learning-based approach using only textual information to determine and explain the validity of bug reports. We use CNN to capture the contextual and semantic features of bug reports without manually identifying features. We experiment on five large open-source projects containing a total of 540,491 bug reports, and the experimental results show that our approach outperforms the state-of-the-art valid bug reports determination approach.
2) By backtracking CNN, we first extract valid bug report key phrases. We inspect these key phrases manually and then summarize valid bug report patterns from three aspects of **Attachment**, **Environment**, and **Reproduce**. Finally, based on these patterns and some related statistics, we provide reporters with suggestions on describing valid bug reports.

This paper is organized as follows. Section II presents our approach in detail. In section III, we describe the datasets used and the evaluation metrics adopted. Section IV presents the experimental results and analysis. In Section V, we discuss the similarities and differences between good bug report patterns and valid bug report patterns. Section VI presents the related work. Section VII summarizes the main threats of our study. The conclusion is presented in Section VIII.

II. Approach

Figure 2 presents the overall framework of our approach which contains three phases: model training, valid bug report prediction, and valid patterns summarization. In the modeling training phase, all the bug reports we collect are divided into a training set and a testing set in chronological order. These data in the training set are fed into a CNN to learn to determine whether bug reports are valid or invalid. In the valid bug report prediction phase, given a future bug report, the trained model is used to predict the valid/invalid label. The trained model is also used in the valid patterns summarization phase. In this phase, we backtrack the trained model to extract the key phrases from the input bug reports. These key phrases are contributing most

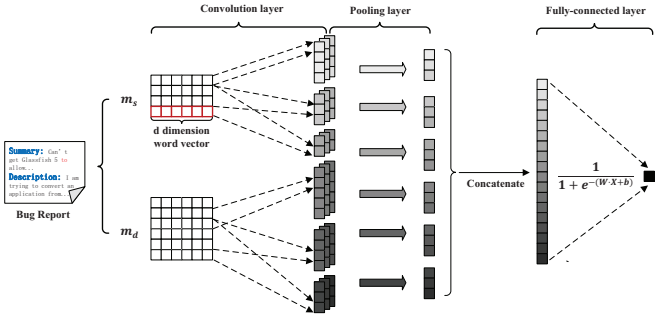


Fig. 3: The architecture of the Convolutional Neural Network.

to determining whether a bug report is valid. By inspecting these key phrases manually, we summarize some valid bug report patterns.

A. Valid Bug Reports Determination by Convolutional Neural Network

1) Data Extraction and Preprocessing: We first separately extract textual information from summaries, descriptions, and comments added by the reporter himself within fifteen minutes. These comments are usually supplements to bugs [4]. Thus, we add descriptions and comments as new descriptions. Zimmermann et al. [6] analyzed the comments within 15 minutes after the creation of bug reports. They found that this information was important for valid bug report detection. Fan et al. followed this study and considered the comments within 15 minutes, and so did we. The descriptions that appear in the following paper refer to the sum of the original descriptions and the comments within fifteen minutes. Then, we perform typical preprocessing steps, including tokenization, stop word removal, and case conversion. To maintain the readability of these texts, we do not perform word stemming.

2) Word Embedding: The preprocessed words must be embedded into vectors before fed into the CNN model. The skip-gram model of word2vec is adopted to transform words in summaries and descriptions into vectors. The output of word2vec is a dictionary containing word vectors of all words in the vocabulary V, and we denote it as $W \in \mathbb{R}^{d \times |V|}$, where d is the dimension of word vectors and $|V|$ is the capacity of V. A text with n words can be represented as a $n \times d$ matrix $m = v_1 \oplus v_2 \oplus \cdots \oplus v_n$, where v_i can be looked up in W and $\oplus$ denotes vector concatenation. Matrix m can be fed as a input to the CNN model.

3) Model Construction: We use CNN to obtain the classification results in this paper. Figure 3 presents the overall workflow of our CNN model.

Convolution Layer. The inputs of the convolution layer are two matrices, i.e., the matrix of summary m_s and the matrix of description m_d. In Figure 3, the summary is represented as a 4×6 matrix and the description is represented as a 6×6 matrix. Then, two independent CNNs are used to extract the features of the two inputs, respectively. The filters f of these two CNNs are $h \times w$ matrices. One row of the input matrix represents a word, so the width of the filter must be equal to the width of the input matrix. To get different features of inputs, we set different filters by varying the height of the filters. Different heights h of filters indicate that the convolution layer can extract various h-gram features of the inputs. As shown in Figure 3, for the input matrix of summary, we set filters of size $1 \times d$, $2 \times d$, and $3 \times d$ to capture 1-gram, 2-gram, and 3-gram features, respectively. The convolution layer processing description is similar to the convolution layer of processing summary, except that its filters extract 2-gram, 3-gram, and 4-gram features of description. The convolution layer that processes the summary contains filters that extract 1-gram features, while the convolution layer that processes description does not. Since the summary is usually much shorter than the description and one word in the summary may also contain crucial information.

Pooling Layer. As seen in Figure 3, we get a lot of feature maps by convolution. These feature maps can be viewed as column feature vectors of different lengths. Pooling is used to reduce the number of features, alleviate the phenomenon of overfitting, and finally get vectors of the same length. In this paper, we use max-pooling to get the most important feature captured by each filter, that is, the maximum value in each feature map. Then, the values selected from the feature maps corresponding to the same size of filters are concatenated to form new feature vectors. In Figure 3, we end up with six three-dimensional feature vectors in the pooling layer.

Fully-connected Layer. We concatenate the feature vectors obtained from the pooling layer to form a vector $X = [x_0, x_1, ..., x_k]$, which is the input of the fully-connected layer. We expect the model to output a value $y_{predict}$ that indicates how likely a bug report is to be valid. Larger values tend to indicate that a bug report is more likely to be valid. This value ranges from 0 to 1, thus we use a linear classifier and sigmoid activation function. Assuming the weight vector $W = [w_0, w_1, ..., w_k]$ and the bias is b, then $y_{predict}$ can be calculated as follows:

$$y_{predict} = \frac{1}{1 + e^{-(\sum_{i=1}^{k} w_i x_i + b)}}. \tag{1}$$

In this paper, we use binary classification cross-entropy as the loss function:

$$\begin{aligned} loss = -(\sum_{i=1}^{n} & y_{real_i} log(y_{predict_i}) + \\ & (1 - y_{real_i}) log(1 - y_{predict_i})) \end{aligned} \tag{2}$$

where n denotes the total number of bug reports.

4) Validity Prediction: We feed a new bug report into the trained model, and get a value that indicates how likely this bug report is to be valid. When this value is greater than or equal to the threshold, this bug report is classified as valid, otherwise it is invalid. In this paper, we set the threshold as 0.5 (according to the characteristic of sigmoid function).

B. Valid Bug Report Patterns Summarization by Backtracking the Trained CNN Model

1) Key Phrase Extraction: We hope to find effective patterns from bug reports, which can provide useful suggestions

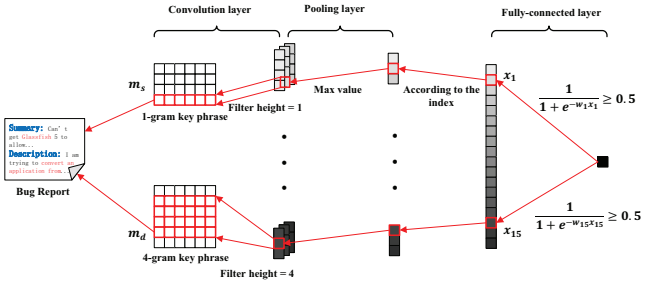


Fig. 4: Backtracking the trained CNN model to obtain key phrases.

for bug reporters. In this paper, the spatial structure of CNN is used to locate phrases for determining valid bug reports. As shown in Figure 4, any features in the fully-connected layer can backtrack to a h-gram phrase in the inputs. The whole process of finding such phrases through backtracking can be broken down into the following steps:

1) We feed a bug report to the trained model and get the parameters in the fully-connected layer, including the feature vectors $X = [x_0, x_1, ..., x_k]$ and the weight vectors $W = [w_0, w_1, ..., w_k]$. For $x_i (1 \leq i \leq k)$ in X, we use a sigmoid function to calculate the probability p of x_i to be valid: $p = \frac{1}{1+e^{-w_i x_i}}$. If p is greater than or equal to 0.5 (according to the characteristic of sigmoid function), we consider that the phrase corresponding to x_i is a key phrase for this bug report to be determined as valid, and call x_i a prominent feature. In Figure 4, features circled by the red frame in the fully-connected layer are prominent features.
2) The feature vector X is obtained by sequentially concatenating the outputs of the pooling layer. Therefore, when the index i of the prominent feature x_i is known, we can find which pooling layer it corresponds to. As seen in Figure 4, prominent feature x_1 corresponds to the second feature of the first pooling layer.
3) Because we use max-pooling in the pooling layer, each feature in the pooling layer is the maximum value of the output of its corresponding convolution layer. Then, for the feature in the pooling layer, we look up the maximum value in its corresponding convolution layer and get the index j of this maximum value.
4) The index j obtained in step 3) can be view as the start index of the key phrase in the input. According to the value of h in the h-gram extracted by the different filters, the entire phrase can be determined as $v_{j:j+h-1}$, where v_j indicates the j_{th} word vector in the input matrix. Finally, we look up the word vectors corresponding to these words in the dictionary to get a readable key phrase. At the same time, we save the probability p calculated in step 1), and use this probability to indicate how likely the key phrase is determined to be valid.

2) Manual Inspecting: For all the bug reports in the training set, we extract the key phrase as described above. For each key phrase, we record such triples during the calculation: $(key\,phrase, count, p_{avg})$, where $count$ denotes the number of bug reports in which this key phrase is found, and p_{avg} denotes the average probability that the phrase is determined to be valid.

For all the key phrases, we sort them in descending order according to their p_{avg}. Then we divide them into several parts based on $count$: 50-200, 200-400, 400-600, 600-800, 800-1000, and >1000. For the first 200 key phrases of each part, we inspect them manually and then summarize some valid bug report patterns.

III. Study Setup

A. Research Questions

First, we answer a research question (**RQ1**) to discuss the performance of our approach. Second, we develop two research questions (**RQ2 and RQ3**) to discuss the patterns summarized in our approach.

RQ1: *Can we determine the validity of a bug report more effectively?*

To evaluate the performance of our approach, we compare our approach with the state-of-the-art approach proposed by Fan et al. [4].

RQ2: *What patterns can we summarize from valid bug reports determination?*

In this research question, we summarize some patterns by backtracking the trained CNN model and manually inspecting key phrases. When a bug report has these patterns, it is more likely to be considered as a valid bug report.

RQ3: *What can we learn from the distribution of the summarized patterns in each project?*

To verify the validity of the patterns proposed in RQ2, for each valid bug report in each dataset, we check whether it contains these patterns. Then we give the statistical results and analyze the distributions of these patterns.

B. Datasets

We collect all fields of bug reports and the full history of bug report updates stored in the Bugzilla system of Eclipse, Netbeans, Mozilla, Firefox, and Thunderbird. Table I shows the statistics of our datasets.

We divided the training and testing set in a time-aware setting for simulating the usage of our approach in the practical bug handling process (i.e., predicting future data based on historical data). In particular, we sort the bug reports in our dataset in chronological order and then divide them into ten folds. The prior nine folds are the training set, and the last fold is the testing set.

We can see from Table I that the class distributions in different projects are different, and they are imbalanced to some extend. For example, in Eclipse, the valid bug reports account for 78%, while in Thunderbird, the valid bug reports only account for 21%.

C. Implement Details

1) Text Preprocessing: For the summary and description extracted from the original bug reports, we only tokenize and remove stop words. In order to make the phrases obtained by

TABLE I: Statistics of our datasets.

Project	Time Period	# Bug Report	# Valid	# Invalid
Eclipse	2010.1-2014.12	119832	93946(78%)	25886(22%)
Net Beans	2010.1-2014.12	53642	32981(61%)	20661(39%)
Mozilla	2013.1-2014.12	222361	154212(69%)	68149(31%)
Firefox	2002.4-2014.12	112830	20794(18%)	92036(82%)
Thunderbird	2000.1-2014.12	31826	6661(21%)	25165(79%)
Total		540491	308594(57%)	231897(43%)

backtracking the trained CNN model readable, we do not stem them. Python's NLTK package is used to tokenize, remove stop words, and converse case.

2) Word Embedding: We use word2vec to convert the words to vectors. The word dimension is set to 200. The window size is 5. We filter words that appear less than five times. Skip-gram is selected as the training algorithm for word2vec.

3) CNN Model:

Input Layer: To unify the length of the input, we adopt a truncation strategy, which preserves the first n words of the input. If there are more than n words in a piece of text, the extra parts are discarded, and if there are less than n words, the zero vectors are used to make up to n words. We set n to 10 for summary and set n to 50 for description.

Convolution Layer: The model has two separate convolution layers for dealing with the summary and description (recall Figure 3). For the convolution layer involved in processing summaries, we set three types of kernels whose lengths are 1, 2, 3, respectively, and the width is 200. For the convolution layer involved in processing descriptions, we set three types of kernels whose lengths are 2, 3, 4, respectively, and the width is 200. For each type of kernel, we set their number to 128. What's more, the stride is set to 1, there is no padding, and the activation function is rectified linear unit (RELU).

Pooling Layer: In the pooling layer, global max-pooling is adopted, and there is also no padding.

Fully-connected Layer: In this layer, the output dimension is 1, and the activation function is sigmoid. We use adam as the optimizer and set the learning rate to 0.0001. An early stop strategy is adopted to stop the training. When the loss no longer drops within four epochs, the model stops training.

D. Evaluation Metrics

For each bug report in the testing set, it has four possible prediction results: a bug report that is predicted to be valid and it is truly valid (true positive, TP); a bug report that is predicted to be valid and it is truly invalid (false positive, FP); a bug report that is predicted to be invalid and it is truly invalid (true negative, TN); and a bug report that is predicted to be invalid and it is truly valid (false negative, FN). Based on these four prediction results, we calculate valid and invalid precision, recall and F1-score to evaluate the performance of our approach and baseline. In addition, we also calculate AUC as another very important evaluation metric.

Valid Precision: it is the proportion of bug reports that are correctly predicted as valid to all bug reports that are predicted as valid. Valid Precision P(v) is calculated as $\frac{TP}{TP+FP}$.

Valid Recall: it is the proportion of bug reports that are correctly predicted as valid to all bug reports that are truly valid. Valid Recall R(v) is calculated as $\frac{TP}{TP+FN}$.

Invalid Precision: it is the proportion of bug reports that are correctly predicted as invalid to all bug reports that are predicted as invalid. Invalid Precision P(inv) is calculated as $\frac{TN}{TN+FN}$.

Invalid Recall: it is the proportion of bug reports that are correctly predicted as invalid to all bug reports that are truly invalid. Invalid Recall R(inv) is calculated as $\frac{TN}{FP+TN}$.

F1-score: it is a comprehensive evaluation of recall and precision, and it is the harmonic mean of them. It evaluates if an increase in precision (or recall) outweighs a reduction in recall (or precision), respectively and provides a balanced view of precision and recall. Valid F1-score F1(v) is calculated as $\frac{2\times P(v)\times R(v)}{P(v)+R(v)}$, while Invalid F1-score F1(inv) is calculated as $\frac{2\times P(inv)\times R(inv)}{P(inv)+R(inv)}$.

AUC: it is the Area Under the Curve (AUC) of Receiver Operator Characteristic (ROC). The AUC is a robust evaluation metric when the class imbalance problem occurs, and it is the main evaluation metric of our approach. The AUC is computed by plotting the ROC curve. By computing the true positive rate (TPR) and false positive rate (FPR) across all thresholds, the ROC curve is obtained. In our approach, the model first outputs a likelihood score for a bug report to be valid. Then this likelihood score is compared with a set threshold (a value ranging from 0 to 1). If the likelihood score is higher than the set threshold, the bug report is determined as valid. Otherwise, it is determined as invalid.

Baseline Method: the baseline of our approach is proposed by Fan et al. [4]. It is the state-of-the-art approach, which has outperformed the approach proposed by Zanetti et al. on the same dataset with ours. Therefore, we believe that comparing with the state-of-the-art approach is sufficient. In order to determine whether a bug report is valid, Fan et al. extract 33 features of the bug report. These 33 features can be grouped into five dimensions: reporter experience, collaboration network, completeness, readability, and text. Based on these features, Fan et al. use the random forest to build a classifier to determine whether a bug report is valid. The baseline can be referred to [4].

IV. Experimental Results

A. RQ1: Can we determine the validity of a bug report more effectively?

Motivation. The key task of our work is to propose a more effective approach for determining valid bug reports. Our baseline relies on a large number of fields in bug reports to

TABLE II: The results of our approach compared with the baseline.

Project	Approach	AUC	Valid			Invalid		
			F1(v)	P(v)	R(v)	F1(inv)	P(inv)	R(inv)
Eclipse	Baseline	0.71	**0.89**	0.84	0.95	0.40	0.60	0.30
	Ours	**0.88**	0.86	0.79	0.95	**0.65**	0.88	0.60
Net Beans	Baseline	0.64	0.79	0.72	0.87	0.33	0.47	0.26
	Ours	**0.71**	0.79	0.74	0.84	**0.53**	0.60	0.46
Mozilla	Baseline	0.80	0.86	0.86	0.85	0.59	0.58	0.60
	Ours	**0.87**	**0.88**	0.84	0.92	**0.63**	0.74	0.55
Firefox	Baseline	**0.93**	**0.75**	0.77	0.74	**0.89**	0.89	0.90
	Ours	0.86	0.72	0.66	0.78	0.72	0.87	0.79
Thunderbird	Baseline	0.82	0.36	0.67	0.25	0.92	0.91	0.98
	Ours	**0.91**	**0.75**	0.76	0.74	0.92	0.91	0.92
Average	Baseline	0.78	0.73	0.77	0.73	0.63	0.69	0.61
	Ours	**0.85**	**0.80**	0.76	0.85	**0.69**	0.80	0.66

extract features. However, some features may not work or be incalculable, and we also found a summary and a description are always available for each bug report. Therefore, we extract summaries and descriptions from bug reports, and then use a CNN as a classifier to determine valid bug reports. To evaluate the performance of our approach and the baseline, we calculate the AUC, Precision, Recall, and F1-score for valid and invalid bug reports across five open-source projects. Table II shows the AUC, Precision, Recall, and F1-score for valid and invalid bug reports and the average values for the five projects.

Results. On Eclipse, our approach improves the baseline by 23.94%, 62.5% in terms of AUC, F1(inv), respectively. On Netbeans, our approach improves the baseline by 10.94%, 60.61% in terms of AUC, F1(inv), respectively. On Mozilla, our approach improves the baseline by 8.75%, 2.32%, 6.78% in terms of AUC, F1(v), F1(inv), respectively. On Thunderbird, our approach improves the baseline by 10.98%, 108.33% in terms of AUC, F1(v), respectively. On Firefox, the performance of the baseline is better than our approach. But on average, across the five projects, our approach achieves 0.85, 0.80, 0.69 in terms of AUC, F1(v), F1(inv), respectively. Compared with the baseline, our approach improves 8.97%, 9.59%, 9.52% in terms of AUC, F1(v), F1(inv), respectively.

Compared to the 33 features of Fan's model, we just use summaries and descriptions as inputs. Although using less information, our approach achieves better results. It shows that summaries and descriptions have great potential to help determine the validity of bug reports. Our method does not outperform the baseline on Firefox. From a feature point of view, only the text of the bug report was used in our approach. For Firefox, the approach of Fan et al. showed very good classification performance, which meant that there were good discriminative features that are not from the text. In their experiments, it was also shown that on Firefox, features from Reporter Experience and Collaboration Network performed well. However, it is not easy to extract these features from each project. Although our approach does not achieve optimal performance on all projects, the data is easy to obtain and process.

Only using the textual information, our approach **achieves** 0.85, 0.80, and 0.69 and **improves** the state-of-the-art approach by 8.97%, 9.59%, 9.52% in terms of AUC, F1(v), and F1(inv) respectively.

B. RQ2: What patterns can we summarize from valid bug reports determination?

Motivation. In order to provide reporters with suggestions on how to describe a valid bug report, we expect to find valid patterns of bug reports from the historical bug reports. Deep learning models are usually considered as black box models, but we can observe that CNN has a spatial structure. Using this spatial structure, we can backtrack from back to front to get some key phrases. These phrases are the basis for CNN to determine a valid bug report. After checking a large number of phrases, we summarize some valid bug report patterns. To the best of our knowledge, this is the first study to investigate the patterns that affect the validity of bug reports.

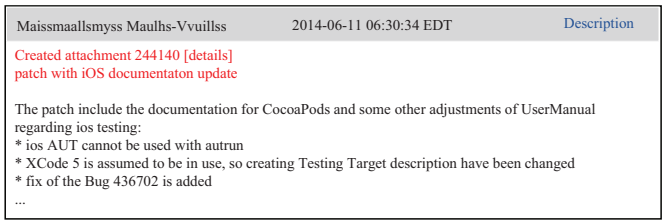
Results. Our inputs consist of two parts: summary and description. The summary is usually a short sentence that contains a small amount of information, but focuses on key points. The description is usually a long text, which contains precious information, but the content is complicated. Therefore, the patterns we summarize are explained from these two aspects, respectively.

Table III presents the patterns summarized from the descriptions. In general, for all projects, their valid patterns are classified into three categories, including **Attachment**, **Environment**, and **Reproduce**. The **Attachment** category contains specific patterns such as *Patch*, *Screenshot*, *Log*, *Test case*, *Code*, and *Stack trace*. With these attachments, fixers can fix bugs more efficiently. Figure 5 shows a bug report with attachments. Usually, if the reporter attaches an attachment to the bug report, he will start his description with *"Created attachment"*, and then write the type and role of the attachment. For example, in the bug report shown in Figure 5, in this way, the reporter expresses that I submit an attachment, and it is a patch for IOS documentation updating.

The patterns of **Reproduce** can help fixers understand the full picture of a bug when they fix it. Figure 6 shows a bug report with these patterns. The reporters remind the readers with *"Steps to reproduce:"*. And then, they itemize how to reproduce this bug. There are some reporters who describe

TABLE III: The valid bug report patterns extracted from descriptions.

Project	Categories	Valid Patterns
Eclipse	Attachment	Created attachment, Patch, Screenshot, Error log, Test case, Code, Stack trace
	Reproduce	Reproducible: always, Steps to reproduce, Actual/Expected result
	Environment	Java Version, JVM Properties, SVN Client, Build Identifier
Netbeans	Attachment	Stack trace, IDE log, NPS snapshot, Server log
	Reproduce	Steps to reproduce
	Environment	Build, VM, OS
Mozilla	Attachment	Screen/Screenshot, Patch, Stack
	Reproduce	Repro steps/Steps to reproduce/Reproduction steps, Repro frequency/Reproduction frequency, Actual/Expected (result)
	Environment	Environmental Variables, User Agent
Thunderbird	Attachment	Patch, Fix, Screen/screenshot
	Reproduce	Reproducible: always, Steps to reproduce, Actual/Expected results
	Environment	User Agent, Build Identifier
Firefox	Attachment	Patch, Fix, Screen/screenshot
	Reproduce	Reproducible: always, Steps to reproduce, Actual/Expected results
	Environment	User Agent, Build Identifier

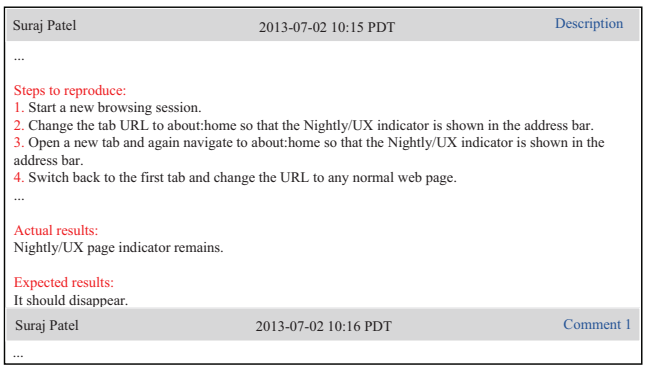

Maissmaallsmyss Maulhs-Vvuillss 2014-06-11 06:30:34 EDT Description

Created attachment 244140 [details]
patch with iOS documentaton update

The patch include the documentation for CocoaPods and some other adjustments of UserManual regarding ios testing:
* ios AUT cannot be used with autrun
* XCode 5 is assumed to be in use, so creating Testing Target description have been changed
* fix of the Bug 436702 is added
...

Fig. 5: An example of a bug report which contains attachment. Bug ID: 437114. Project: Eclipse.

"Reproduce steps:" or other phrases with the same meaning. A considerable number of reporters write *"Actual result(s):"* and *"Expected result(s):"*. The pattern *Actual result(s)* represents the software behavior seen by the reporters, and the pattern *Expected result(s)* represents the software behavior expected by the reporters. These patterns are also used to help the fixers understand the bugs, so they are divided into the category **Reproduce**. What's more, some reporters omit *"result(s)"* and write *"Actual/Expected:"* directly.


Suraj Patel 2013-07-02 10:15 PDT Description

...

Steps to reproduce:
1. Start a new browsing session.
2. Change the tab URL to about:home so that the Nightly/UX indicator is shown in the address bar.
3. Open a new tab and again navigate to about:home so that the Nightly/UX indicator is shown in the address bar.
4. Switch back to the first tab and change the URL to any normal web page.
...

Actual results:
Nightly/UX page indicator remains.

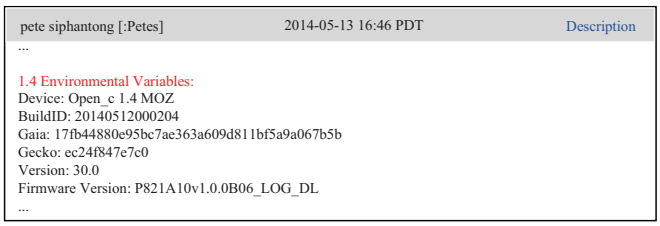
Expected results:
It should disappear.

Suraj Patel 2013-07-02 10:16 PDT Comment 1

...

Fig. 6: An example of a bug report which contains reproduce. Bug ID: 889428. Project: Mozilla.

Patterns in **Environment** remind readers that the following contents indicate the software or hardware environmental information of the project when a bug occurs. Figure 7 shows a bug report with patterns representing environmental information. In this bug report, the reporters first use *"Environmental Variables:"* to indicate that the following content is the project-related environmental variables and then describe the content in detail. There are many versions of open-source projects which are deployed on various software and hardware environments. When a bug occurs, providing its environmental information to the fixers can help them quickly find the causes and solutions. Different projects require various environmental information. For integrated development environments such as Eclipse, reporters usually indicate the configuration of the Java virtual machine. For browsers such as Firefox and Thunderbird, the user agent and browser version are usually provided by reporters.


pete siphantong [:Petes] 2013-05-13 16:46 PDT Description

...

1.4 Environmental Variables:
Device: Open_c 1.4 MOZ
BuildID: 20140512000204
Gaia: 17fb44880e95bc7ae363a609d811bf5a9a067b5b
Gecko: ec24f847e7c0
Version: 30.0
Firmware Version: P821A10v1.0.0B06_LOG_DL
...

Fig. 7: An example of a bug report which contains environment. Bug ID: 1009939. Project: Mozilla.

The summary is different from the description. It is usually a short description of the bug. Therefore, in a summary, there are almost no patterns related to completeness. In the summary, reporters usually use some words or phrases to summarize the content of this bug. For example, in Eclipse, the summary of bug report 364813 is *"[Sequence Diagram] - Part decompositions and combined fragments"*, the summary of bug report 407414 is *"[compiler][null] Incorrect warning on a primitive type being null"*, and the summary of the bug report is *"NPE in faces config editor"*. We find that in the summary of valid bug reports, reporters usually describe the unit where the bug occurs or the basic type of the bug. At the same time, the reporters use brackets to surround some keywords so that readers can more accurately understand the bugs.

For each project, we **summarize** some **patterns** to help determine bug reports as valid from three aspects of **Attachment, Reproduce**, and **Environment**.

C. RQ3: What can we learn from the distribution of the summarized patterns in each project?

Motivation. In RQ2, we state the valid bug report patterns summarized by manually inspecting key phrases which are obtained by backtracking the trained CNN model. In this research question, we present some statistics to analyze the distributions of these patterns. We first set up some regular expressions according to the types and related phrases of valid bug report patterns. Then for each valid bug report, we use it to match regular expressions. Finally, for each valid bug report pattern, we get the number of valid bug reports that contain it.

Results. Table IV shows the statistical results. Since the category **Attachment** contains many specific items, we gather not only the statistics for pattern *Attachment*, but also the statistics for its specific items. Besides, we classify *Result* and *Steps to Reproduce* patterns to **Reproduce**. Usually, in the bug reports, *"Actual/Expected result:"* is written independently of *"Steps to reproduce:"*. Therefore, we also gather the statistics of the two separately. In Table IV, the bold items represent specific patterns summarized for each project (recall Table III).

According to the overall situation of the statistical results, for ease of explanation, we roughly define the inclusion rate greater than 20% as high and inclusion rate greater than 5% and less than 20% as medium. For Eclipse, the pattern with high inclusion rate is *Environment*, and those with medium inclusion rate are *Attachment*, *Code*, *Patch*, *Result*, *Stack Trace*, and *Steps to Reproduce*. For Netbeans, the patterns with high inclusion rates are *Attachment*, *Environment* and *Stack Trace*, and those with medium inclusion rates are *Result*, *Screen*, and *Steps to Reproduce*. For Mozilla, the pattern with high inclusion rate is *Attachment*, and those with medium inclusion rates are *Patch*, *Result*, *Screen*, and *Steps to Reproduce*. For Thunderbird, the patterns with high inclusion rates are *Attachment*, *Environment*, *Result* and *Steps to Reproduce*, and those with medium inclusion rates are *Patch* and *Screen*. For Firefox, the patterns with high inclusion rates are *Attachment*, *Environment*, *Result* and *Steps to Reproduce*, and the those with medium inclusion rates are *Patch* and *Screen*.

Based on these statistics horizontally and vertically, we have the following interesting findings:

(1) For each project, the inclusion rates of the patterns we summarized in RQ2 (i.e., bold items in Table IV) are much higher than those of patterns that were not summarized. It shows that inspecting the key phrase manually to summarize valid bug report patterns is effective.

(2) For all the projects, the inclusion rate of *Attachment* is high. Furthermore, for *Log*, *Code*, and *Stack Trace* patterns, the inclusion rates in Eclipse and Netbeans are significantly higher than those in Mozilla, Thunderbird, and Firefox. This is related to their project type. Eclipse and Netbeans are both integrated development environments, and the other three projects are browsers. Of course, in the bug reports of the integrated development environment, there are more patterns related to code, log, and stack trace. Therefore, our **first suggestion** is: when submitting a bug report, reporters should submit as many attachments as possible related to the bug according to the type of current project.

(3) For most projects, the inclusion rate for pattern *Environment* is high. Since there are usually many versions for an open-source project, and the environments in which users use a project are different. This may cause trouble for bug fixers. Therefore, our **second suggestion** is: when submitting a bug report, reporters should submit as much information about the environment of the project as possible, such as the version of the project and the version of the operating system.

(4) For all projects, *Result* and *Steps to Reproduce* patterns are highly included, especially for Thunderbird and Firefox. This shows that when a bug fixer fixes a bug, he/she usually needs to reproduce the bug and fully understand the process of this bug occurring to find a solution. Therefore, our **third suggestion** is: when submitting a bug report, reporters should recall and record the process of the bug occurring as detailedly as possible.

> In different projects, the distributions of valid bug report patterns are different. But for each project, its distribution is basically consistent with the patterns we summarized in RQ2. Based on the statistical results, we give reporters suggestions on describing valid bug reports in the three aspects of **Attachment, Environment, and Reproduce**.

V. Discussion

A good bug report refers to a high-quality bug report, which usually contains adequate and correct information [9]. A valid bug report refers to a bug report which points to a real bug and contains a complete description and can be reproduced [4]. Both good bug reports and valid bug reports are related to the quality of bugs and the information contained in them. In this section, we discuss the **similarities** and **differences** between good bug report patterns investigated by Zimmermann et al. and the valid bug report patterns proposed in this paper.

We first focus on the **similarities**. Zimmermann et al. [9] did a survey. They sent questionnaires (some factors expected to affect the quality of the bug report are listed) to developers, aiming to understand what information in the bug reports is most commonly used and what information is most important to developers when they fix bugs. Zimmermann et al. investigated the quality of bug reports from the perspective of developers. Their survey results showed that the previously used items were *steps to reproduce*, *observed and expected behavior*, *stack traces*, and *test cases*. In addition, their survey results showed that the most important items were *steps to reproduce*, *test cases*, *observed behavior*, and *screenshots* for GUI errors. In this paper, we summarize valid bug report patterns from the perspective of data (i.e., bug reports). It is interesting that the survey results from developers and patterns from data (recall Table III) overlap to a large extent. The feature engineering-based approach [4], [6] and questionnaire have one thing in common: they both need to list factors that

TABLE IV: Statistics of valid bug reports which contain valid bug report patterns. Exclusion indicates the number of bug reports which do not contain a certain pattern, while inclusion indicates the number of bug reports which contain a certain pattern. The bold items indicate the information of patterns we have summarized in Table III for each project.

Project	Pattern / Distribution	Attachment	Code	Log	Patch	Test Case	Screen	Stack Trace	Step to Reproduce	Result	Environment
Eclipse	Exclusion	**76322**	**89244**	**92531**	**86846**	**93066**	**90854**	**85072**	**76686**	**86612**	**74406**
	Inclusion	**17624**	**4702**	**1415**	**7100**	**880**	**3092**	**8874**	**17260**	**7334**	**19540**
	Inclusion Rate	**18.76%**	**5.01%**	**1.51%**	**7.56%**	**0.94%**	**3.29%**	**9.45%**	**18.37%**	**7.81%**	**20.80%**
Netbeans	Exclusion	**17461**	31412	**32113**	32546	32814	**29648**	**23335**	**29362**	30715	**17946**
	Inclusion	**15520**	1569	**868**	435	167	**3333**	**9646**	**3619**	2266	**15035**
	Inclusion Rate	**47.06%**	4.76%	**2.63%**	1.32%	0.51%	**10.11%**	**29.25%**	**10.97%**	6.87%	**45.59%**
Mozilla	Exclusion	**115096**	153065	154507	**143421**	152102	**145428**	153443	**134359**	**127699**	**148291**
	Inclusion	**39116**	1147	155	**10791**	2110	**8784**	769	**19853**	**26513**	**5921**
	Inclusion Rate	**25.37%**	0.74%	0.10%	**7.00%**	1.37%	**5.70%**	0.50%	**12.87%**	**17.19%**	**3.84%**
Thunderbird	Exclusion	**15481**	20683	20784	**18734**	20638	**19271**	20767	**12530**	**13924**	**15073**
	Inclusion	**5313**	111	10	**2060**	156	**1523**	27	**8264**	**6870**	**5721**
	Inclusion Rate	**25.55%**	0.53%	0.05%	**9.91%**	0.75%	**7.32%**	0.13%	**39.74%**	**33.04%**	**27.51%**
Firefox	Exclusion	**4579**	6642	6654	**5538**	6652	**6194**	6645	**4023**	**4344**	**4621**
	Inclusion	**2082**	19	7	**1126**	9	**467**	16	**2638**	**2317**	**2040**
	Inclusion Rate	**31.26%**	0.29%	0.11%	**16.90%**	0.14%	**7.01%**	0.24%	**39.60%**	**34.78%**	**30.63%**

may affect the quality or validity of the bug report in advance. Our approach does not need to set the expected factors. We use CNN to automatically extract the features from text and find the factors that really affect the validity by backtracking.

Next, we discuss the **differences**. As shown in Table III, we have summarized some patterns about environmental information for each project, and as shown in Table IV, many valid bug reports contain this environmental information. However, this is inconsistent with the results of Zimmermann et al [4]. The feedback they received indicates that fields such as version, operating system, product, and hardware are less important than expected. Nonetheless, they suggested to treat this result with caution. In the experiment, we have found that a considerable number of the bug reports that have been determined to be valid contain this information. Therefore, when reporters can provide this environmental information, we still recommend to submit them together. They are not irrelevant information, and may still be useful in understanding, reproducing, or triaging bugs.

Compared with the questionnaire, the patterns we summarize are more specific because they are obtained from the bug reports. There are some descriptions that exactly exist in the bug report. Some useful details may not be reflected in the options in the questionnaire. For example, as shown in Table III, we summarize a pattern *"Reproducible: Always"*. There is a case where the reporters may have written the steps to reproduce, but when the fixer follows the process to reproduce the bug, it fails. If the reporter has specified that this bug can always be reproduced, bug triagers can more quickly determine its validity before assigning the bug to the appropriate fixer. The feedback provided by experienced developers is undoubtedly valuable. We summarize some interesting patterns from the perspective of data, supporting and supplementing the developer's experience.

VI. Related Work

In this section, we describe the related studies on valid bug report detection and bug report quality.

A. Valid Bug Reports Determination

In the literature, few approaches have been proposed for valid bug report detection. To the best of our knowledge, Zanetti et al. [6] were the first to study valid bug report detection. They used collaborative networks on open source projects to detect valid bug reports. First, they built a collaborative network based on ASSIGN and CC relations of bug reports. Then they extracted nine features from this network. Finally, these features are applied to the model to determine whether a bug report is valid or not. Fan et al. [4] considered more features. They extracted features not only from the collaborative network, but also from the other four dimensions, including reporter experience, completeness, readability, and text. From these five dimensions, they extracted a total of 33 features. They built models using random forest and support vector machine, and then, applied the extracted 33 features to the models. Experimental results showed that their method significantly improved the performance compared with the method of Zanetti et al.

A common limitation of these two approaches is that they are highly dependent on handcrafted features. However, for a newly reported bug report, some fields may not be available yet. Thus, some features that depend on these fields can not be extracted. As a result, the validity of this new bug report can not be determined in time. In addition, the performance of these two approaches is limited by these handcrafted features. Fan et al. found that although all features were useful for detecting valid bug reports, features extracted from the texts were particularly important. Based on this finding, we extract the texts of bug reports, i.e., summaries and descriptions, and then use the neural network to determine the validity of bug reports.

B. Bug Report Quality

The quality of bug reports affects the efficiency of bug management [13]–[18], bug location [19]–[23], and bug fix [24]–[27]. There have been many studies on bug report quality.

Bettenburg et al. [28] were aware of the quality of the information in the bug reports. They conducted a survey among developers to determine the information of bug reports that developers widely used and the problems developers frequently encounter. They also built a prototype of a tool to determine the quality of bug reports by scanning the contents of bug reports. But their investigation was conducted only in Eclipse's developers. Zimmermann et al. [9] added the Apache

and Mozilla projects to the survey. They also proposed a model called CUEZILLA, which could automatically analyze the quality of bug reports and could classify bug reports into five levels varying from very bad to very good. Their research was carried out in the form of questionnaires, which is from the perspective of developers. In this paper, we explore factors that affect the validity of bug reports based on historical bug reports (i.e., the perspective of data). In addition, we have experimented on more projects and more bug reports.

Several approaches have been proposed for predicting the quality of bug reports. Hooimeijer et al. [29] proposed a descriptive model to determine the quality of bug reports and predict whether bug reports are triaged in a given time. Bhattacharya et al. [30] focused on the fix time of bug reports. They used univariate and multivariate regression to show how previous models fail, and then built a more accurate model for predicting bug fix time of bugs. Linstead et al. [31] used latent dirichlet allocation (LDA) to predict the quality of bug reports, which modeled the bug report content in an unsupervised way. Their experimental results showed that applying statistical text mining algorithms had great potential for predicting the quality of bug reports. Aversano et al. [32] extracted some factors that might affect the quality of bug reports. In particular, they used the reporter's reputation.

There are also some studies focused on improving the quality of bug reports. Weimer et al. [33] stated that the patches in bug reports were important. Thus, they proposed an algorithm that could automatically generate patches in bug reports. In addition, they found that reports with patches were fixed three times faster than reports without patches. Schroter et al. conducted an empirical study in the developers of Eclipse about the use of stack traces in bug reports. Their study provided evidence of the importance of stack traces in bug reports. In addition, they emphasized the need to introduce stack traces to the bug reports. Lotufo et al. [34] investigated the use of game mechanics in Stack Overflow. They found that this method of motivating contributors could improve the quality of contributions. Therefore, they introduced game mechanics to the bug tracking system. Zhang et al. [35] found that more than 70% of bug reports in Eclipse contained no more than 100 words. Therefore, they proposed a sentence ranking algorithm to enrich the content of bug reports. Their experimental results showed that enhanced bug reports could improve the performance of automated fixer recommendation. Chaparro et al. [36] found that steps to reproduce with low quality could make bug fixing difficult. Therefore, they proposed an approach called Euler, which could automatically identify steps to reproduce with low quality and feedback to reporters, and then reporters could modify it to improve the quality of bug reports.

VII. Threats to Validity

In this section, we investigate some potential facts that may threaten the validity of our approach.

Internal Validity. The threat to internal validity is the potential errors in our experimental implementation. In order to reduce errors in the code, we double-check our code ourselves, and ask an experienced deep learning developer to recheck our code again until no errors were found. Another threat to internal validity is the baseline replication. To mitigate this threat, we use the source code provided by the author.

External Validity. The threats to external validity relate to the generalizability of our experimental results. We have evaluated our approach on five public open-source projects. The whole datasets contain 540491 bug reports and have different distributions.

Construct Validity. The threat to construct validity is the suitability of our evaluation measure. In this paper, we use AUC and F1-score as the main evaluation measure. Moreover, we calculate F1-scores for valid bug reports and invalid bug reports, respectively (i.e., F1(v) and F1(inv)). These measures have been adopted in our baseline [4]. In this way, we have reduced the threat to construct validity by a lot.

VIII. Conclusion

In this paper, we propose a deep learning-based approach for determining and explaining valid bug reports. We extract the texts (i.e., summaries and descriptions) from bug reports, and use a CNN model to extract the contextual and semantic features. We evaluate the performance of our approach on five public open-source projects, including Eclipse, Netbeans, Mozilla, Thunderbird, and Firefox. Our approach achieves 0.85, 0.80, 0.69, which improves the approach proposed by Fan et al. [4] by 8.97%, 9.59%, 9.52% in terms of average AUC, F1(v), and F1(inv) respectively. In order to explain the performance of our approach and give reporters some suggestions on how to submit valid bug reports, we backtrack the CNN model to obtain some phrases that make the model determine bug reports as valid. Then, we manually inspect these phrases and summarize some valid bug report patterns by category (**Attachment**, **Environment**, and **Reproduce**) for each project. For each pattern, we also count the valid bug reports that include it. In summary, our conclusion is as follows:

1) Our approach improves the baseline proposed by Fan et al. [4] in the case of using only textual information of bug reports.
2) Through backtracking the trained CNN model and manual inspecting, we summarize some patterns that facilitate the valid bug report determination by category.
3) We make statistics for each pattern, and the results prove that the patterns we summarize are indeed useful.
4) Based on valid bug report patterns and statistics, we provide reporters with some suggestions on submitting valid bug reports.

Acknowledgment

This work was supported in part by the National Key Research and Development Project (No.2018YFB2101200), the Fundamental Research Funds for the Central Universities (No.2019CDYGYB014 and No.2020CDJQY-A021).

REFERENCES

[1] N. Serrano and I. Ciordia, "Bugzilla, itracker, and other bug trackers," *IEEE software*, vol. 22, no. 2, pp. 11–13, 2005.

[2] Y. C. Cavalcanti, E. S. de Almeida, C. E. A. da Cunha, D. Lucredio, and S. R. de Lemos Meira, "An initial study on the bug report duplication problem," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 264–267.

[3] M. Yan, X. Zhang, D. Yang, L. Xu, and J. D. Kymer, "A component recommender for bug reports using discriminative probability latent semantic analysis," *Information and Software Technology (IST)*, vol. 73, pp. 37–51, 2016.

[4] Y. Fan, X. Xia, D. Lo, and A. E. Hassan, "Chaff from the wheat: Characterizing and determining valid bug reports," *IEEE transactions on software engineering (TSE)*, 2018.

[5] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 35–39.

[6] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1032–1041.

[7] H. T. Le, C. Cerisara, and A. Denis, "Do convolutional networks need to be deep for text classification?" in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[8] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in Neural Information Processing Systems*, 2015, pp. 649–657.

[9] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 5, pp. 618–643, 2010.

[10] R. Shwartz-Ziv and N. Tishby, "Opening the black box of deep neural networks via information," *arXiv preprint arXiv:1703.00810*, 2017.

[11] W. Samek, T. Wiegand, and K.-R. Müller, "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models," *arXiv preprint arXiv:1708.08296*, 2017.

[12] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: Interpretable deep learning by propagating activation differences," *arXiv preprint arXiv:1605.01713*, vol. 4, 2016.

[13] 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 Symposium on the Foundations of Software Engineering (FSE)*, 2009, pp. 111–120.

[14] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 884–887.

[15] 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 (WCRE)*. IEEE, 2012, pp. 215–224.

[16] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 2–11.

[17] J. Deshmukh, S. Podder, S. Sengupta, N. Dubash *et al.*, "Towards accurate duplicate bug retrieval using deep learning techniques," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 115–124.

[18] A. Budhiraja, K. Dutta, M. Shrivastava, and R. Reddy, "Towards word embeddings for improved duplicate bug report retrieval in software repositories," in *Proceedings of the 2018 ACM SIGIR International Conference on Theory of Information Retrieval*, 2018, pp. 167–170.

[19] N. Ali, A. Sabane, Y.-G. Gueheneuc, and G. Antoniol, "Improving bug location using binary class relationships," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2012, pp. 174–183.

[20] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon, "D&c: A divide-and-conquer approach to ir-based bug localization," *arXiv preprint arXiv:1902.02703*, 2019.

[21] Z. Shi, J. Keung, K. E. Bennin, and X. Zhang, "Comparing learning to rank techniques in hybrid bug localization," *Applied Soft Computing*, vol. 62, pp. 636–648, 2018.

[22] C. Oo and H. M. Oo, "Spectrum-based bug localization of real-world java bugs," in *International Conference on Software Engineering Research, Management and Applications*. Springer, 2019, pp. 75–89.

[23] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.

[24] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering (EMSE)*, vol. 14, no. 3, pp. 286–315, 2009.

[25] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2009, pp. 121–130.

[26] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 117–128.

[27] E. C. Campos and M. d. A. Maia, "Discovering common bug-fix patterns: A large-scale observational study," *Journal of Software: Evolution and Process (JSEP)*, vol. 31, no. 7, p. e2173, 2019.

[28] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, 2007, pp. 21–25.

[29] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering (ICASE)*, 2007, pp. 34–43.

[30] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 207–210.

[31] E. Linstead and P. Baldi, "Mining the coherence of gnome bug reports with statistical topic models," in *2009 6th IEEE International Working Conference on Mining Software Repositories (MSR)*. IEEE, 2009, pp. 99–102.

[32] L. Aversano and E. Tedeschi, "Bug report quality evaluation considering the effect of submitter reputation." in *ICSOFT-EA*, 2016, pp. 194–201.

[33] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, 2006, pp. 181–190.

[34] R. Lotufo, L. Passos, and K. Czarnecki, "Towards improving bug tracking systems with game mechanisms," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 2–11.

[35] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug report enrichment with application of automated fixer recommendation," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 230–240.

[36] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2019, pp. 86–96.