Empirical Software Engineering (2021) 26: 118
https://doi.org/10.1007/s10664-021-10010-8

Why and what happened? Aiding bug comprehension with automated category and causal link identification

Cheng Zhou[1,2] · **Bin Li**[1] · **Xiaobing Sun**[1,3] · **Lili Bo**[1,3]

Accepted: 30 June 2021/ Published online: 25 August 2021

Abstract

When a new bug report is assigned to developers, they first need to understand what the bug report expresses (***what***) and why this bug occurs (***why***). To do so, developers usually explore different bug related data sources to investigate whether there are historical bugs with similar symptoms and causes related to the bug at hand. Automatic bug classification with respect to what and why information of bugs would enable developers to narrow down their search of bug resources and improve the bug fixing productivity. To achieve this goal, we propose an approach, *BugClass*, which applies a deep neural network classification approach based on Hierarchical Attention Networks (HAN) to automatically classify the bugs into different what and why categories by exploiting the bug repository and commit repository. Then, we explore the causal link relationship between what and why categories to further improve the accuracy of the bug classification. Experimental results demonstrate that *BugClass* is effective to classify the given bug reports into what and why categories, and can be also effectively used for identifying the why category for new bugs based on the causal link relations.

Keywords Bug comprehension · Bug classification · Causal link · Hierarchical attention network

1 Introduction

As the size of the software continuously increases, platform compatibility, user demand changes and software version evolution bring new functional modules and new conflicts between each other, and result in a large number of bugs (Zhou et al. 2020; Timperley et al. 2018; Zhou et al. 2019; Zhou et al. 2018; Le Goues et al. 2015). Promptly fixing software

Communicated by: Shaowei Wang, Tse-Hsun (Peter) Chen, Sebastian Baltes, Ivano Malavolta, Christoph Treude, and Alexander Serebrenik

This article belongs to the Topical Collection: *Collective Knowledge in Software Engineering*

✉ Xiaobing Sun
xbsun@yzu.edu.cn

Extended author information available on the last page of the article.

bugs to maintain the quality of the system is a prerequisite for ensuring the quality of the software (Wang et al. 2017; Tan et al. 2014). As a common practice, researchers design and propose lots of approaches to explore automatic program repair (Le et al. 2017; van Tonder and Le Goues 2018; Xiong et al. 2017; Wen et al. 2018; Jiang et al. 2018; Xia and Lo 2017; Soto and Le Goues 2018). These approaches could generate positive results, but sometimes suffer from the overfitting problem (Le et al. 2018) and may be only suitable for specific types of bugs (Motwani et al. 2018; Zhou et al. 2020; Sun et al. 2019; Zhong and Mei 2018). Moreover, it is difficult for these techniques to fix bugs based on the given free-style bug reports (Motwani et al. 2018; Zhong and Mei 2018; Chaparro et al. 2017; urphy-Hill et al. 2015).

Kim and Whitehead Jr. (2006) reported that the median time for fixing a single bug is about 200 days, and developers always spend more than half of their time on the bug comprehension activity to confirm what happened to the software (i.e., symptom) and why it happened (i.e., cause) to determine where in the code the bug actually locates before starting modifying the program. Given a new bug report, developers need to first understand its cause and symptom before they get a patch resolution (Bòhme et al. 2017). However, an in-depth study of bugs at different projects, along with a more general survey of developers (Davies and Roper 2014; Bugde et al. 2008), indicates that information in the majority of new bug reports is incomplete and inaccurate. Users or testers typically report the circumstances leading to the exposure of a bug and the likely impact to the users in the new bug report, while information about the cause of a bug is not always obvious until a bug has been verified, fixed and closed.

For clarity, let's consider a real bug report from Mozilla, one of the two projects taken into account in our study. Given the bug report *1067042* in Fig. 1, we can easily identify that the function of the bookmark does not match the design requirements from the title and description, which is the symptom. By further analyzing the comments and attachments (i.e., commit files) in Fig. 2, we notice that the bug is caused by a parameter call error (Lines 290-295 in buggy code), expressing the cause information for this bug. In fact, the bug has been repaired for more than half a year (2014-2015). As revealed in the comments, at first developers thought the suspicious code was at line 277, then at line 315, which may be a logic problem. When the initial patch was submitted, the test failed and the initial patch was overturned. Finally, developers correctly located to line 286 and resubmitted a new patch. For a new bug report, there are few comments or commit files for developers to analyze or deduce the cause information. Therefore, developers can more easily analyze the symptom of the new bug based on the text content in the bug report, but difficult to determine its cause.

A lot of research works have targeted the area of automatic cause classification of software maintenance requests including bug triage, severity prediction, duplicate bug detection, bug location and so on (Podgurski et al. 2003; Huang et al. 2015; Thung et al. 2015; Ocariza et al. 2017; Terdchanakul et al. 2017; Qin and Sun 2018; Nayrolles and Hamou-Lhadj 2018; Catolino et al. 2019; Ni et al. 2020). Bugs are initially classified before their cause is investigated manually. Accurate cause classification can not only help project managers improve software control and allocate their testing resources effectively, but also support developers speed up the comprehension activities with much less human effort. For example, assuming that developers know that the cause of the bug *1067042* belongs to the *Interface* category (see Table 1 for a detailed explanation of the category), they may prefer to locate the buggy code at line 286. However, almost all work attempts to group together bugs with the same or similar causes by combining text content in bug reports and code

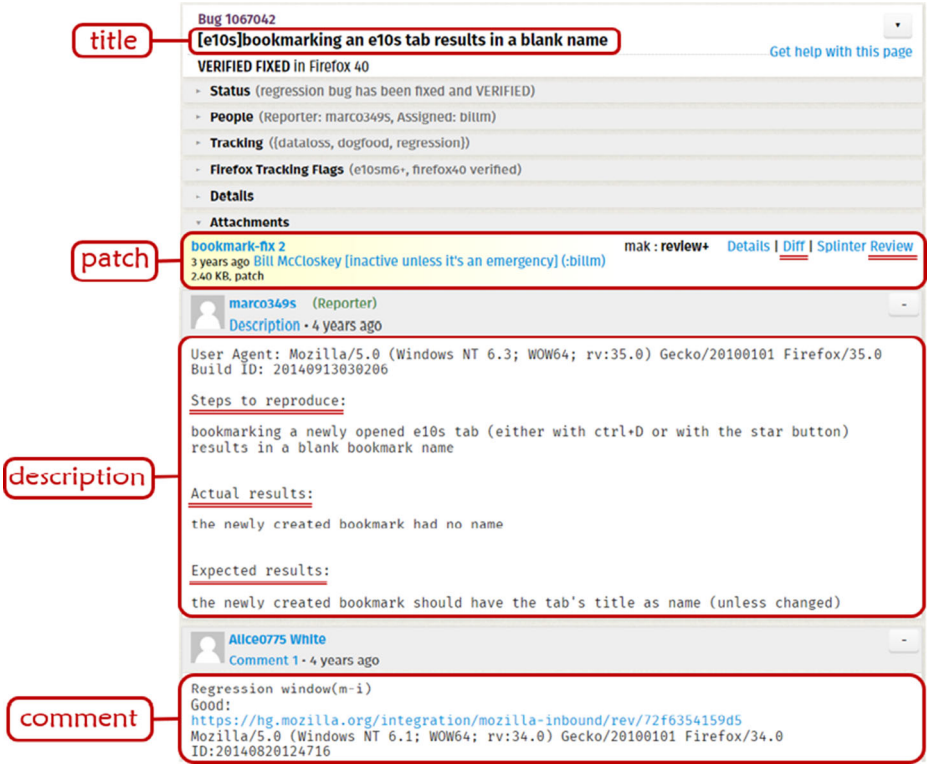


Fig. 1 The bug report of bug 1067042 in Mozilla Bugzilla

features or only by code features. Few works take into account the premise that a new bug report lacks code files. When the code features are removed, the bug cause classification results are much worse.

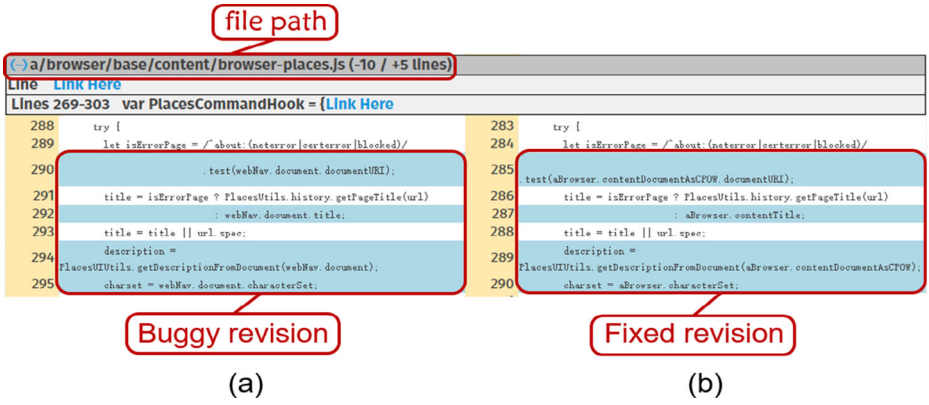


Fig. 2 The commit file of bug 1067042 in Mozilla Bugzilla

At the same time, we notice that researchers always start from a certain perspective to complete the practice of automatic bug classification. On the basis of orthogonal classification of bug reports from three perspectives (i.e., root causes, impacts, and components), Tan et al. found that there are some correlation between attribute categories in different perspectives (Tan et al. 2014). For example, the majority of security bugs are caused by semantic bugs. Inspired by the work of Tan et al., we form a initial conjecture that *there are some stable strong association between cause categories and symptom categories, which can help improve cause classification*. In this article, instead of classifying the cause directly according to the text content of a new bug, we extract the potential associations between the two attributes of cause and symptom from the fixed historical bug data as the prior knowledge, and predict the cause category according to the bug symptom category. To achieve this, we have to overcome the following challenges:

- Although researchers have proposed several ways to classify bugs, existing bug classification approaches fail to meet the comprehensive understanding of both the symptom and the cause for bugs (Hamill 2015). Moreover, there is no good benchmark for us to systematically evaluate the associations between them (Le Goues et al. 2015; Shan et al. 2005).
- The associations between the two attributes of cause and symptom are usually various and complex. It is difficult to analyze and represent the associations between them.
- We need to predict the cause of the bug based on its symptom. However, most of the existing bug classification methods combine artificial feature engineering and machine learning classifiers, which do not fully mine the semantics in the bug report and can not give a satisfactory result of bug symptom classification.

To address the above challenges, we first examine the fixed historical bug data, track their fixes and define a classification criterion for the two bug attributes of cause and symptom based on existing standards. Then, we collect 30000 verified fixed bug reports from the Mozilla project BTS [1] (Bug Tracking System) and 20000 verified fixed bug reports from the Eclipse project BTS [2]. From them, we manually label 1000 and 700 bug reports as the ground truth. On this basis, we extract Abstract Syntax Trees (AST) of diff structures from the commit files as code features, and apply the Hierarchical Attention Network (HAN) (Pappas 2017) to classify the above 50000 historical bugs from two perspectives (i.e., symptom and cause) respectively, forming two attribute datasets. Thirdly, we use the Apriori algorithm to deal with the attribute datasets, mining association rules between fine-grained symptom and cause categories. Fourth, we combine association rules and symptom classification probability to optimize the cause classification probability and predict the new bug cause category. The major contributions include:

- We provide an orthogonal bug classification criterion from two perspectives: cause and symptom, and use deep learning network, combining text and code features, to realize the automatic classification of historical bug.
- We explore the potential causal links between bug symptoms and bug causes, and express them as quantitative association rules adopting the Apriori algorithm.
- We improve the new bug cause classification by combining association rules and known symptom category to optimize the cause classification probability. We evaluate our approach on two open source projects, Mozilla and Eclipse. The results show that our

[1] https://bugzilla.mozilla.org/

[2] https://bugs.eclipse.org/bugs/

approach can effectively improve the accuracy of new bug cause classification, and confirm that there are some stable causal links between the two bug attributes of symptom and cause.

The remainder of this article is structured as follows: Section 2 introduces the background. In Section 3, we show an automatic classification approach for historical bugs and new bugs. In Section 4, we present the empirical study and show the empirical results to evaluate our approach, and followed by the threats to validity in Section 5. We discuss the related work in Section 6. In Section 7, we conclude this paper and discuss the future work.

2 Background

In this section, we introduce the background concerning bug data sources, bug taxonomies, and the Hierarchical Attention Network (HAN) model to support our approach.

2.1 Bug Data

As more and more bugs have been reported, in order to conveniently manage and trace these bugs, many large software projects are equipped with a dedicated Bug Tracking System (BTS) such as Bugzilla [3] and JIRA [4] to collect and store information and process these bugs. In the Bugzilla for Mozilla, more than 1.8 million bug reports have been accumulated in the first two decades from 2000 to 2021. However, there seems to be no universally accepted definition of the term *bug*, even in so many BTSs. According to Avizienis et al. (2004), a *failure* is "an event that occurs when the delivered service deviates from correct service", a *fault* is "the *cause* of a *failure*". *Bugs* and *defects* are synonyms of *faults*, and *failures* are the "observable *symptom* of software *faults*". Thus, we use the terms *bug* and *defect* interchangeably in this article.

There are many bug data, but the new bug is only the bug report submitted by users or testers. Bug reports are files used to describe software bugs by which users of a system are able to report a bug to the developers. Their contents are important, including the basic information of bugs themselves and status migration during the entire bug fixing process. As Fig. 1 shows, a bug report typically includes these parts: title, description, comment, attachment, and multiple attributes such as type, component and product. Davies and Roper (2014) reported ten most important features developers found most useful when fixing bugs. However, few bug reports have more than five features, and the majority have three or fewer.

Among the information, *Steps to reproduce, Observed behavior* and *Expected behavior* can be found in the description part of most bug reports. In general, developers can determine the symptom of a new bug by reading its text content and reproducing it. Figure 2 shows the commit file of the bug *1067042*, which includes a commit message, file paths, code diff changes, etc. The information can assist in identifying the cause of a bug. However, *patches* (Tarnpradab et al. 2018), *Code examples* (Tan et al. 2014), *Screenshots* and *Stack traces* (Campos and de Almeida Maia 2017) are relatively rare, especially for new bugs. This increases the difficulty for developers to infer the causes of new bugs and hinders the subsequent bug fixing process. With such limited information, we are committed

[3] https://www.bugzilla.org/

[4] https://www.atlassian.com/software/jira

to exploring the causal relations between the two bug attributes of symptoms and causes, which helps to predict the cause of a new bug based on easily known symptom.

2.2 Orthogonal Bug Classification for Symptom and Cause

In this work, we focus on new bug comprehension. The purpose of bug classification is to explore whether there are stable causal relations between the symptoms and causes of historical bugs. Therefore, we need to give two independent taxonomies from two different starting points for the same bug. The Orthogonal Defect Classification (ODC) [5] for software design and code proposed by Chillarege et al. at IBM in 1992 is a widely used defect classification standard in the industry (Chillarege et al. 1992; Huang et al. 2015; Thung et al. 2012; Thung et al. 2015). It believes that information on bugs is available at two specific points in time, three attributes (i.e., Activity, Triggers, Impact) could be collected when the bug is opened and five attributes (i.e., Target, Type, Qualifier, Source, and Age) could be collected when the bug is closed. We have applied ODC's idea of classification by stages, that is, classification of symptom mainly based on the description text of bug report, and classification of cause mainly based on the commit files of bug report. At the same time, we also refer to the IEEE Standard Classification for Software Anomalies (IEEE Std 1044-2009) [6], which provides developers with bug classification scheme from multiple perspectives (539, 2010). Based on our experience with many real-world bug reports and the inspirations from the above two standards , we designed bug taxonomies in two dimensions, **cause** (the fault) and **symptom** (the failure caused by the bug).

2.2.1 Bug Symptom Taxonomy

The standard *Symptom RR500* in the IEEE Std 1044-1993 classifies bugs from the perspective of symptoms, which consists of eight categories (i.e., *Operating system crash, Program hang-up, Program crash, Input problem, Output problem, Perceived total product failure, System error message, Other*). We also consider sub types in the Input problem and Output problem categories. ODC defines thirteen fine-grained *impact* categories for the impact of bugs to the customer. According to the classification statistics of different projects by Huang et al. (2015) and our real annotation, we exclude seven impact categories with too small distribution (less than 1%) because they have no significance for association mining. The six impact categories reserved are *Security, Performance, Usability, Reliability, Requirements and Capability*. We combine the above categories in the form of supersets, and classify bugs into nine symptom categories as shown in Table 1.

2.2.2 Bug Cause Taxonomy

Most of the existing bug taxonomies and practices rely on code, such as Basili and Selby (1987), Humphrey (1995), and Beizer (1990), and Shepperd (1993). The bug report is reported by the user or tester, which describes an objective phenomenon caused by bugs, rarely subjective cause judgment. Therefore, we can mainly deduce the cause from the code review of historical bugs. ODC gives detailed classification for the *defect type* attribute, the IEEE Std 1044-1993 also gives the *Type IV300* from the perspective of cause. Compared

[5]The definitions and taxonomy of ODC v5.2 attributes are accessible at http://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf.
[6]https://standards.ieee.org/standard/1044-1993.html

Table 1 Bug categories from the cause and symptom perspective

Perspective	Bug category	Description	Abbreviation
Cause	Resource	Bugs refer to mistakes made with data, variables, or other resource initialization, manipulation, and release.	RE
	Check	Bugs are validation mistakes or mistakes made when detecting an invalid value.	CH
	Interface	Bugs are mistakes made when interacting with other parts of the software, such as an existing code library, a hardware device, a database, or an operating system.	IN
	Logic	Bugs refer to mistakes made with comparison operations, control flow, and computations and other types of logical mistakes.	LO
	Timing	Bugs that are possible only in multithread applications where concurrently executing threads or processes use shared resources.	TI
	Function	Unlike presented above, bugs cannot be pinpointed to a single, small set of code lines. Function bugs typically refer to situations in which functionality is missing or implemented incorrectly and such bugs often require additional code or larger modifications to the existing solution.	FU
	Support	Support bugs relate to support systems and libraries or their configurations, change management or version control.	SU
	Documentation	Bugs caused by missing, inaccurate, inconsistent or incomplete documents.	DO
	Enhancement	The bugs needs to be fixed through the change in program requirements or code efficiency improvement as it affects significant capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure(s).	EN
Symptom	Security	The systems, programs, and data have been inadvertently or maliciously destruction, alteration, or disclosure.	SE
	Reliability	The software cannot consistently perform its intended function with unplanned interruption, crash, or hang up.	RB
	Input problem	Unreasonable input detection, correct input not accepted or wrong input accepted, incorrect data source, incomplete/missing parameters.	IP
	Output problem	The result output data, incomplete/missing parameters, incomplete/missing result, incorrect data processing.	OP
	Message	System error message, unclear or unreasonable message notification.	ME
	Performance	The speed of the software as perceived by the customer and the customer's end users, that is, functions correctly but runs/responds slowly.	PE
	Usability	The software and publications make product hard to understand and inconvenient for end users, does not affect the function.	US
	Capability	The software does not perform its intended functions, does not meet known requirements, not addressing any of the previous categories.	CA
	Requirements	A customer expectation, with regard to capability, which was not known, understood, or prioritized as a requirement for the current product or release.	RM

with the above bug taxonomies and combined with manual code reviews, bugs are mainly divided into two groups: *functional* bugs (affect the correct functioning of a software system and related to the code) and *non-functional* bugs (not affect the correct functioning of a software system or even not in the code) (Thung et al. 2012). According to the classification proposed by Mäntylä and Lassenius (2009), we first select six categories belong to the functional group including: *Resource, Check, Interface, Logic, Timing* and *Function*. Then, based on the *Type IV300* standard, we determine three categories belong to the non-functional group including: *Enhancement*, *Support* and *Documentation*.

To ensure disjoint classification, setting priorities of categories is of critical importance. Once a bug is labeled as a higher priority category, it will no longer be considered as a lower priority category. The priority of the category is consistent with the top-down order of the category being presented in the table. As shown in Table 1, from the perspective of cause, bugs are divided into nine categories, of which *Resource* has the highest priority, and *Enhancement* has the lowest priority. Similarly, from the perspective of symptom, bugs are also divided into nine categories, of which *Security* has the highest priority, and *Requirements* has the lowest priority.

2.3 Hierarchical Attention Network (HAN)

In our work, the automatic classification of bugs is the basis for bug comprehension. As a text classification task for bug reports, the entire process is divided into two parts: feature engineering and classifier. In the general domain (such as news), traditional approaches for text classification represent documents with sparse lexical features, such as n-grams (Wang and Manning 2012; Joachims 1998). Recently, more neural network architectures have been applied, such as Convolutional Neural Networks (CNN) (Kalchbrenner et al. 2014; Hermann and Blunsom 2014) and Recurrent Neural Network (RNN) (Hochreiter and Schmidhuber 1997), to automatically acquire feature expression capabilities, remove complicated artificial feature engineering.

As shown in Fig. 1, most of the textual content in the bug report is contained in three parts: title, description and comment. On the one hand, there is a lot of redundant user communication information in comments, so that the information distribution density is significantly lower than the other two parts. On the other hand, the cause information that can be probed in the comments is much more. In view of the uneven distribution of information in the bug report, in this article, we adopt a deep neural network classification model called Hierarchical Attention Networks (HAN) based on RNN (Yang et al. 2016) which has shown its effectiveness on other types of user-produced data (Tarnpradab et al. 2018; Gao et al. 2018; He et al. 2019). HAN mines both semantic relationship between words in a sentence as well as between sentences in the whole bug report. We consider a bugset $\{(b_i, l_i)|i = 1, \cdots, N\}$ made of N documents b_i (i.e., text content of the bug report) with labels $l_i \in \{0, 1\}^k$ (i.e., the bug category). Each document is represented by the sequence of d-dimensional embeddings of their words grouped into sentences, $b_i = \{w_{11}, w_{12}, \cdots, w_{KT}\}$, where T is the maximum number of words in a sentence, and K is the maximum number of sentences in a document. The HAN takes as input a document b_i and outputs a document vector v_i, which learns effective sentence representation by attending to important words, and similarly learns document representation by attending to important sentences in the document. The details of HAN are shown in Fig. 3. The word hierarchy is made of an encoder g_w with parameters H_w and an attention model α_w with parameters A_w, while the sentence hierarchy similarly includes an encoder and an attention

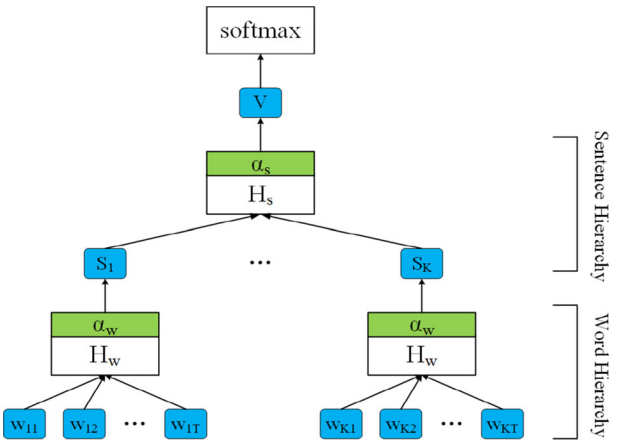


Fig. 3 Hierarchical attention network

model (g_s, H_s and α_s, A_s). The output v_i is used by the classification layer to determine the label l_i .

HAN consists of the following parts: a word sequence encoder, a word-level attention layer, a sentence sequence encoder, a sentence-level attention layer and a softmax layer.

2.3.1 GRU-Based Word Sequence Encoder Layer

The word hierarchy reads an input sentence and outputs a sentence vector. The function g_w encodes the sequence of input word embeddings $\{w_{it}|t = 1, \cdots, T\}$ for each sentence s_i of the document, noted as:

$$h_w^{it} = \{g_w(w_{it})|t = 1, \cdots, K\} \tag{1}$$

HAN commonly uses the bi-directional Gated Recurrent Unit network (Dey and Salem 2017) noted as BiGRU as the encoder. GRU is a variant of Long Short-Term Memory (LSTM), one of the state-of-the-art RNNs. The BiGRU computes the new state as a concatenation of the hidden states for each input word vector obtained from the forward GRU $\overrightarrow{g_w}$, and the backward GRU $\overleftarrow{g_w}$ as:

$$h_w^{it} = [\overrightarrow{g_w}(w_{it}); \overleftarrow{g_w}(w_{it})] \tag{2}$$

The same concatenation is applied for the hidden state representation of a sentence h_s^i.

2.3.2 Word-Level Attention Layer

Not all words contribute equally to the representation of the sentence meaning. Similarly, the sentences in the document are not equally important. HAN introduces an attention mechanism at each hierarchy (noted as α_w and α_s) to extract such words that are important to the meaning of the sentence, then aggregate the representation of those informative words to form a sentence vector s_i. W_w and b_w are model parameters.

$$u_{it} = tanh(W_w h_w^{(it)} + b_w) \tag{3}$$

$$\alpha_w^{it} = \frac{exp(u_{it}^\top u_w)}{\sum_t exp(u_{it}^\top u_w)} \tag{4}$$

$$s_i = \frac{1}{T}\sum_{t=1}^{T}\alpha_w^{(it)}h_w^{(it)} \tag{5}$$

2.3.3 GRU-Based Sentence Sequence Encoder Layer

The sentence hierarchy encodes the sequence of input sentences $\{s_i|i = 1, \cdots, K\}$ and outputs the document vector d. Similarly, the BiGRU concatenates the hidden states for each input sentence vector obtained from the forward GRU $\overrightarrow{g_s}$, and the backward GRU $\overleftarrow{g_s}$ to represent each sentence as:

$$h_s^i = [\overrightarrow{g_s}(s_i); \overleftarrow{g_s}(s_i)] \tag{6}$$

2.3.4 Sentence-Level Attention Layer

Similarly, in the sentence hierarchy, we extract sentences that are important to the meaning of the whole document, then aggregate the representation of those informative sentences to form the last document vector v as follows, W_s and b_s are model parameters :

$$u_i = tanh(W_s h_s^i + b_s) \tag{7}$$

$$\alpha_s^i = \frac{exp(u_i^\top u_s)}{\sum_i exp(u_i^\top u_s)} \tag{8}$$

$$v = \frac{1}{K}\sum_{i=1}^{K}\alpha_s^i h_s^i \tag{9}$$

2.3.5 The Output Layer

The document vector v is a high-level representation of the document and fed to a softmax layer for classification, with a loss L based on the negative log likelihood of the correct labels (Yang et al. 2016).

$$p = softmax(W_c v + b_c) \tag{10}$$

$$L = -\sum_d logp_{d_j} \tag{11}$$

W_c and b_c are model parameters.

3 Approach

In the face of a new bug report with little information, if developers can get the new bug cause category with accurate judgment, the difficulty of bug comprehensive will be alleviated. In this section, we present our automatic bug report classification approach called *BugClass* combining deep learning and association rules. As illustrated in Fig. 4, the classification framework consists of three phases: 1) classification of historical bug reports from two perspectives of causes and symptoms separately to build the multi-attribute database; 2) mining rational association rules between bug symptoms and bug causes from the attribute classification information of historical bug reports; 3) prediction of the new bug report cause category optimized by association rules and symptom category probability.

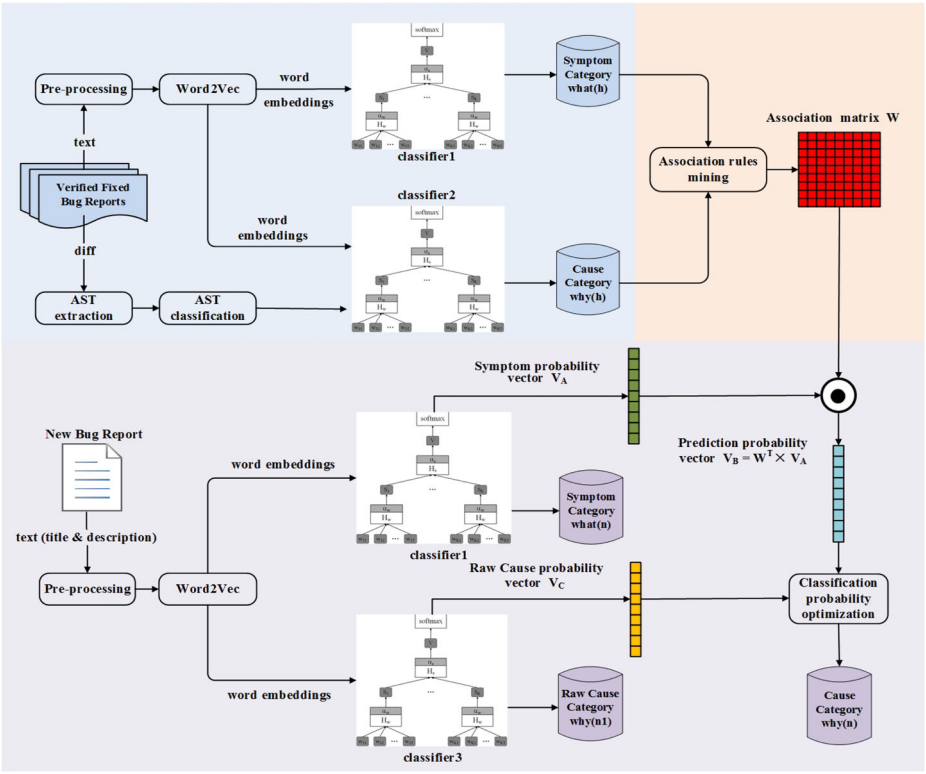


Fig. 4 Our approach *Bugclass* to classify bug reports from the symptom and cause perspectives

3.1 Classification of Historical Bug Reports for the Multi-Attribute Database

In our approach, basic HAN model is used to identify both the symptom category (a.k.a. $what_h$) and cause category (a.k.a. why_h) of historical bug reports separately. We use HAN for classification as HAN: 1) mines both semantic relationship between words in a sentence as well as between sentences in bug reports; 2) effectively extracts more important words and sentences from bug reports. HAN has shown its effectiveness on other types of user-produced data (Pappas 2017; Gao et al. 2018). The classified historical bug reports constitute a multi-attribute database, which lays a bugset for the next step of rational association rules mining.

3.1.1 HAN-Based Historical Bug Symptom Classification

We use the HAN model as classifier to process the text content in the historical bug reports and identify the bug symptom category, which includes three steps including text pre-processing, word embedding and HAN model learning.

Text Pre-processing We extract text content (i.e., title, description and comments) from each bug report to form a bug document, and remove links, XML tags, all numbers and punctuation marks as they often have weak correlation with the meaning of the bug reports.

Then, we pre-process bug documents with the Natural Language Toolkit (NLTK) [7]. Specifically, we perform word tokenization, remove stop Words, employ the Snowball stemmer to transform the words into their root forms (e.g., "results" and "resulting" are reduced to "result"), and employ the Truecase to restore uppercase to the most probable state (e.g., "Page Title" is reduced to "page title") in order to unify similar words into a common representation.

Word Embedding After tokenization, each bug document is divided into a series of word tokens. By translating these word tokens into low-dimensional word vectors, word embedding can capture the semantic and syntactic information of words from a large amount of unclassified bug data. Compared to random initialization, we pre-train the word embeddings on the larger bugsets (seen in Table 3) by the Word2Vec tool [8] using the skip-gram model, as it trains faster, takes up less memory and disk space (Hu et al. 2018). In this way, we convert each bug document into a tensor of size $B_{K\times T\times e}$, where K is the number of sentences per document, T is the number of words per sentence, and e is the word embedding size. These tensors become the input for the HAN model.

HAN Model Learning Because the corpus constructed by manual annotation is relatively small, we select 20% of the bug reports as the test set. Then, the 5-fold cross-validation method is used for the remaining 80% data in the corpus. We use sequential optimization with gradient boosted trees to find the best hyper-parameters for our HAN (Gao et al. 2018) model. This optimization method uses a gradient boosted tree-based regression model to predict the model performance at unexplored hyper-parameter settings. We use this optimization method because tree-based optimization has been shown to converge faster than traditional Bayesian optimization (Gao et al. 2018). The following hyper-parameters need to be tuned: (1) size of word embeddings (100-500); (2) type of RNN unit used (GRU or LSTM); (3) number of hidden GRU or LSTM cells used in each bidirectional RNN layer (50-200); (4) size of hidden layer in attention mechanism (50-300); (5) dropout on final document embedding (0.5 or none).

After learning, we get the bug symptom automatic classifier (a.k.a. *classifier1*) and classify the historical bug reports in bugsets, representing the symptom category label as l_a.

3.1.2 HAN-Based Historical Bug Cause Classification

There are rich information for the fixed bug information in the BTS. Most of the historical bug reports have detailed text descriptions and completed patch files. The developers can repair the historical bugs effectively only if they correctly judge the causes of the bugs. Therefore, according to this fact, we have reason to think that the analysis of the patch file that has been verified fixed is conducive to the backward deduction of the cause of the bug. Hence, we combine the two different sources (i.e., text and code) by introducing the diff code structure as a feature to assist in identifying the cause categories of historical bug reports. There are four steps including text pre-processing, word embedding, code feature extraction and HAN model learning. The first two steps are to deal with the text content in the bug report, the fourth step is the training process of HAN model. They are the same as the processing procedure in the previous Section 3.1.1. The third

[7] http://www.nltk.org/

[8] http://https//code.google.com/p/word2vec/

step is code feature extraction, which is divided into two parts: AST extraction and AST classification.

AST Extraction As shown in Fig. 2, the buggy revision and the fixed revision are displayed at the same time in the commit file. Developers can directly view the differences between them (a.k.a. diff). A simple way to classify diff is to just view the source code as plain text , and the structural information will be omitted. The Abstract Syntax Tree (AST) is a tree representation of the abstract syntax structure of source code. Each node in the AST represents a code structure. We use GumTreeDiff [9], a code differencing tool proposed by Falleri et al. (2014) to convert diff to ASTs.

AST Classification Since the diff structure can be represented as an AST T, differences between trees can be used to compare the information of key nodes to analyze the type of code structure changes.

According to the cause categories shown in Table 1, we define six diff structure types as code features: *resource, check, interface, logic, timing, function*. We analyze the diff structures in the BTS, summarize fix patterns for each of the first five diff structure types and construct the diff pattern set which is represented as C. Unlike those presented above, the type *function* refers to larger bugs in which functionality is missing or implemented incorrectly and such bugs often require additional code or larger modifications to the existing solution. Therefor, the type *function* cannot be pinpointed to a single, small set of fix patterns.

We match the code change mode for historical bugs by the following formula.

$$match\left(x,e\right)=\begin{cases}1\ if\ l\left(x\right)=l\left(e\right)\\ \quad and\ sim_{2g}\left(v\left(x\right),v\left(e\right)\right)\geq f\\ \\ 0\ otherwise\end{cases} \tag{12}$$

x and e are nodes in AST. $l\left(x\right)$ denotes the label of x, and $v\left(x\right)$ denotes the value of x. f is the threshold for the string similarity (in default, f=0.6).

The matching procedure (Chawathe et al. 1996) mentioned in the algorithm consists in identifying the appropriate node x $(x \in T)$ and the similar type pattern element e $(e \in C)$. Two nodes match if their strings are similar according to a given string similarity measure $sim(x, e)$. We use the method proposed by Adamson and Boreham to calculate the similarity of two strings by setting their n-grams as a relationship (Adamson and Boreham 1974). We use the n-gram similarity measure as it is more fault-tolerant, and does not rely on the longest common subsequence. Instead, it mainly focuses on common characters and secondarily on word order. N-grams are the sequence of n items in a given text. The n-gram similarity measure is the ratio of twice the number of shared n-grams and the total numbers of n-grams in two strings s_a and s_b:

$$sim_{ng}\left(s_a,s_b\right)=\frac{2\times\left|n-grams\left(s_a\right)\bigcap n-grams\left(s_b\right)\right|}{n-grams\left(s_a\right)\bigcup n-grams\left(s_b\right)} \tag{13}$$

The algorithm to mine the diff structure is presented in Algorithm 1. The input of the algorithm is the grammatical structure tree T of the diff file analyzed based on the AST. The output of the algorithm is a collection of type fix patterns for the current diff results, indicating the type to which the current code structure belongs. The algorithm first calculates

[9] https://github.com/GumTreeDiff/gumtree

the degree of matching of all nodes with the characteristics of the corresponding type (Lines 2-6). The elements in C_{tmp} are ranked in a descending order according to node similarity (Line 9), and the best match result is added to the final pattern set C_{final} for T (Lines 8-10).

Algorithm 1: The algorithm for matching the structure type of fixed code.

Input: Trees T, Fix pattern set C
Output: Final type pattern set C_{final}
1 $C_{final} \leftarrow \emptyset, C_{tmp} \leftarrow \emptyset$;
2 **for** *each node* $x \in T$ *and* $e \in C$ **do**
3 **if** *match(x,e)* **then**
4 $C_{tmp} \leftarrow C_{tmp} \bigcup \left(x, e, sim_{2g}\left(v\left(x\right), v\left(e\right)\right)\right)$;
5 **end**
6 **end**
7 Sort the elements in C_{tmp} in a descending order according to the node similarity;
8 **for** *each node-similarity* $\left(x, e, sim_{2g}\left(v\left(x\right), v\left(e\right)\right)\right) \in C_{tmp}$ **do**
9 $C_{final} \leftarrow C_{final} \bigcup (x, e)$;
10 **end**
11 **return** C_{final}

Naturally, we can determine the type of the AST structure based on the type of the best match result in C_{final}. In particular, when there are multiple best matching fix patterns in C_{final}, and the types of fix patterns are not consistent, that is, there are multiple diffs of different types in one bug, we consider this situation as a *function* type.

From the changed code structure, we can analyze some of the causes of bugs. For example, part (a) in Fig. 2 is the partial buggy version of the bug *1067042*, and part (b) is the corresponding fixed version. It can be seen that there is a call error occurred and matched to the *interface* structure type . So for bug *1067042*, it can be viewed as a bug caused by the *Interface* category. Finally, we take the type of diff structure as a feature and reload it into HAN for processing, and the cause category as shown in Fig. 4 can be identified, representing as l_b. In order to distinguish, we call the learned historical bug cause classifier as *classifier2*.

3.2 Mining Association Rules of Bug Categories

For new bug reports, there is little information and no commit files. Compared to the symptom, it is more difficult to identify the cause of the bug based solely on the textual content in the bug report. It is difficult to identify the cause of a new bug report by using neural network alone. We choose to combine the association rules with neural network, mining the association in bug category data as a prior knowledge to tune the confidence of cause recognition results.

3.2.1 Association Rules Between Symptoms and Causes

As one of the most important branch of data mining, association rule mining (Agrawal and Srikant 1994) identifies the associations and frequent patterns among a set of items in a given database. Association rules can be used to find the association of each attribute of the target in a large number of data, so as to assist in the decision-making of the target attribute type. They are widely used in multiclass and multilabel algorithm (Bian et al. 2018;

Sharma et al. 2018; Azmi et al. 2019). This article mainly focuses on the classification of two dimensional attributes of bug reports, namely, bug causes and bug symptoms. Therefore, the relevant definitions of the applied association rules are as follows:

Definition 1 We use the trained *classifier1* and *classifier2* in Section 3.1 to classify historical bug reports in the larger bugset shown in Table 3, and get the symptom category (i.e., l_a) and cause category (i.e., l_b) of each historical bug report respectively. In our approach, we mine association rules from a larger bugset rather than simply from a manually annotated corpus for two reasons: (1) Although the annotated corpus is accurate, the amount of data is too small to fully reflect the main causal link between bug attributes in a huge bug database; (2) In the follow-up experiments, we use the annotated corpus as the ground-truth to train and test the classifier. If only the association rules mined from the corpus are used as a prior knowledge, the test results are not objective and accurate.

Given the bug multi-attribute database DB,

$$I = \{l_{a1}, l_{a2}, \cdots, l_{a9}, l_{b1}, l_{b2}, \cdots, l_{b9}\}$$

is the set of all bug attribute categories as shown in Table 1. T is a subset of I, representing a set of two attributes of a bug. Each bug T has a unique ID, that is, the ID of each bug report in the BTS. Because each bug is categorized in two perspectives, and the attribute categories in each perspective are disjoint, we only need to consider *frequent 2-itemset*. Association rules in dataset DB are expressed as implication $A_i \Rightarrow B_j$, and $A_i \in I$, $B_j \in I$, $A_i \cap B_j = \varnothing$. The symptom category is the former item A_i, while the cause category is the latter item B_j, and the latter item is recommended on the basis of the former item. For example:

$$A_i = \{l_{a3}(Input\ problem)\} \quad B_j = \{l_{b4}(Logic)\}$$

The association rule can be expressed as:

$$\{l_{a3}(Input\ problem)\} \Rightarrow \{l_{b4}(Logic)\}$$

Definition 2 To study the causal links between symptom and cause categories, we use three statistical metrics called *support*, *confidence* and *lift* (Han et al. 2011). They are defined as follows:

$$support(A_i \Rightarrow B_j) = P(A_i B_j) = \frac{|A_i \cap B_j|}{|A_i \cup B_j|} \tag{14}$$

$$confidence(A_i \Rightarrow B_j) = \frac{P(A_i B_j)}{P(A_i)} \tag{15}$$

$$lift(A_i \Rightarrow B_j) = \frac{P(A_i B_j)}{P(A_i) * P(B_j)} \tag{16}$$

where $P(A_i)$ is the probability that a bug belongs to the category A_i and $P(B_j)$ is the probability that a bug belongs to the category B_j. *Support* $P(A_i B_j)$ is the probability that a bug belongs to both category A_i and B_j. *Confidence* is an indication of how often the rule has been found to be true, while *lift* is used to judge the independence and correlation of events.

Take the bug *1067042* in Fig. 1 as an example , there are in total 1000 sampled bugs in the Mozilla corpus, 289 of which are Capability bugs, 79 of which are Interface bugs, and 38 of which are Interface bugs that cause failure impacting Capability, we can calculate the values of *support*, *confidence* and *lift* as 0.038, 0.1315, and 1.66. We show the *lift* results on the Mozilla corpus in Table 2. We only keep the *lift* value greater than 1, which means there

Table 2 Correlation between bug causes and bug symptoms in Mozilla (The horizontal axis is the abbreviation of cause category, and the vertical axis is the abbreviation of symptom category)

category	RE	CH	IN	LO	TI	FU	SU	DO	EN
SE	0	0	0	0	0	1.53	2.64	0	0
RB	0	**8.08**	0	0	0	0	2.61	0	0
IP	0	0	0.94	0	0	0	0	**5.79**	1.02
OP	**4.19**	0	0	1.56	0	0	0	0	0
ME	0	0	0	0	2.05	1.47	0	0	2.3
PE	0	0	1.94	0	**8.71**	1.27	0	0.	0
US	1.37	0	0	0	0	0	0	0	**3.5**
CA	1.11	0	1.66	1.94	0	0	1.04	0	0
RM	0	0	0	0	0	2.18	0	0	1.56

Values in bold denote the highest values

is a positive correlation. Values greater than 3 are shown in bold. In RQ3 in Section 4.4, we further explore the influence of the dataset DB size on the results of association rule mining and the overall approach.

3.2.2 Mining Association Rules with Apriori Algorithm

We use the Apriori algorithm (Agrawal and Srikant 1994; Liu et al. 2018) to calculate the above statistical metrics, formulate association rules, and exclude some rules with low probability of occurrence. Apriori algorithm has two important threshold parameters, *minsupp* and *minconf*, which represent the minimum value of support and confidence respectively. The process of mining association rules is divided into three steps:

Step 1: The database DB is scanned globally, and *frequent 1-itemset* $L1$ is extracted by *minsupp*.

Step 2: By connecting each *frequent 1-itemset* $L1$, pruning to obtain *frequent 2-itemset* $L2$.

Step 3: Calculate confidence and lift, filter strong association rules with confidence value greater than *minconf* and lift value greater than 1.

According to the above strong association rules, the causal link of different cause categories on the symptom categories can be obtained. We convert the filtered association rules into a matrix $W_{9\times9}$, with rows representing nine symptom categories and columns representing nine cause categories. The element value in the matrix is the confidence value of the corresponding association rule. If there is no strong association rules between A_i and B_j, the corresponding element $a_{i,j}$ is 0.

3.3 New Bug Cause Category Prediction Based on Association Rules

There is no commit file in the new bug report, so we can only analyze the symptom and cause of a new bug through the text content (i.e., the title and description), which is mainly divided into three steps as shown in Fig. 4: 1) classify the symptoms of new bug reports (a.k.a. $what_n$) based on HAN model; 2) classify the causes of new bug reports (a.k.a. why_{n1})

based on HAN model; 3) fine-tune the raw cause category (a.k.a. why_{n1}) based on association rules and known symptom category (a.k.a. $what_n$) for an optimized cause category (a.k.a. why_n).

3.3.1 HAN-Based Symptom Classification for New Bugs

Since there are few comments in a new bug report, we just take the title and description from it as input, and adopt the trained *classifier1* in Section 3.1.1 to classify the new bug symptom. After learning and softmax operation, we use the function *classifier.predict_Proba()* to get the symptom probability vector V_A. Each element in V_A represents the prediction probability value of the bug in the corresponding symptom category $P(y = A_i|x)$. The label with the maximum probability value will be output as the symptom category of the input new bug report.

we assign the association matrix $W_{9\times9}$ to the output symptom probability vector V_A, making it to be the prediction probability vector V_B for cause categories.

$$V_B = W^\top V_A \tag{17}$$

Each element in V_B represents the prediction probability of the bug in the corresponding cause category $P(y = B_j|x)$.

3.3.2 HAN-Based Cause Classification for New Bugs

We still use HAN model as the underlying classifier to carry out a preliminary classification of new bugs. The correctness of commit files cannot be guaranteed until the new bug has been successfully fixed. So we give up the commit files, just like analyzing symptoms, only extract the title and description of a new bug report as cause classification input. After learning and softmax operation, we get the raw bug cause category probability vector V_{Craw}. Each element in V_{Craw} represents the raw prediction probability value of the bug in the corresponding cause category $P(y = C_i|x)$. Different from historical bugs (i.e., *classifier2*), we call the preliminary learned new bug cause classifier as *classifier3*.

3.3.3 Cause Classification Optimized by Association Rules

According to the experimental results of RQ1 discussed in Section 4.4 , we can confirm that if the label with the maximum probability value in V_{Craw} is directly output as the cause category of the new bug report, the classification accuracy is relatively low. We try to mine association rules from two-dimensional attribute data of bugs as prior knowledge to fine tune the confidence of raw cause recognition results.

In details, the new cause classification probability is optimized as:

$$V_{Cnew} = V_{Craw} \times (1 - \alpha_C) + V_B \times \alpha_C \tag{18}$$

α_C is the weight parameter to adjust the degree of optimization, which controls whether to modify the raw classification result to association rule result. The larger the α_C is , the more likely the bug cause attribute will be modified by association rules. However, if α_C is too large, not only the influence of association rules will decrease, but also the results in some special cases may be modified by mistake, resulting in low accuracy. Through the experiment, we find that when α_C exceeds 0.5, the test results will not change. So we set it to 0.5. Although there will be false modification, the advantage of association rules will be reflected when the correct example of modification exceeds the false example.

In *Bugclass*, the causal relations between causes and symptoms are mined from the historical bug categories, which are introduced into the new bug automatic classification system to obtain a more reliable cause category. With it, developers can directly access the historical bugs with the same attribute categories for reference, reducing the difficulty of information retrieval. On the other hand, they have a credible repair direction, which improves the efficiency of bug location.

4 Empirical Study

In this section, we use the bug data from two projects (i.e., Mozilla and Eclipse) illustrated in Table 3 to conduct our empirical study.

4.1 Research Questions

To show the effectiveness of our approach, we mainly focus on the following research questions:

RQ1: How effective does HAN-based classification approach perform compared with the state-of-the-art bug classification techniques?

Our approach uses the HAN model as the underlying learning algorithm to classify bugs from the cause and symptom perspectives. We try to mine association rules as prior knowledge from the attribute classification of historical bugs, and predict the cause category of a new bug according to its symptom category. Therefore, we first investigate whether the HAN model can improve the classification effectiveness compared with state-of-the-art bug classification techniques.

RQ2: How effective is our proposed approach in inferring the cause category of a new bug with association rules?

In practice, it is more difficult to determine the cause of a new bug than the symptom. We use the association rules between symptoms and causes to optimize cause classification for new bugs. So we need to investigate whether such association rules are effective to predict the cause category for a new bug.

RQ3: How is the performance of our proposed approach affected with varying amount of historical bugs used to mine association rules?

We mine association rules from the symptom and cause classification of historical bugs. However, as a ground truth, the scale of human annotation corpus is limited. Therefore, we need to observe the impact of the amount of historical bugs on association rules.

4.2 Datasets

4.2.1 Tagging of Bug Category

Given a bug report, we can classify the bug into the above defined bug categories from both the cause and symptom perspectives. For example, the bug *1067042* in Fig. 1 belongs to the *Capability* category according to the symptom perspective and *Interface* category according to the cause perspective. To ensure correct classification, we only sample bug reports for resolution "fixed" and status "verified", "resolved" and "closed" because these bug reports consist of meaningful information for the experiment.

As shown in Table 3, we randomly collect 30,000 fixed bug reports from the Mozilla BTS and 20,000 fixed bug reports from the Eclipse BTS to be two individual bugsets according

Table 3 The distribution of our datasets

Project	Bugset	Corpus	Setences in Corpus	Tokens in Corpus
Mozilla (historical bug)	30,000	1,000	53,452	807,534
Mozilla (new bug)	-	1,000	6,791	50,029
Eclipse (historical bug)	20,000	700	34,307	581,574
Eclipse (new bug)	-	700	4,025	27,608

to components. There are many components in a project. The number of bug reports in a component can be very large, up to tens of thousands, also can be very small, even only a few dozen, with a huge gap between each other. If the sampling is completely random, there will be a "loss of a component" situation. Then, by stratified sampling according to the components, 1,000 verified fixed bug reports have been obtained from the Mozilla bugset and 700 verified fixed bug reports have been obtained from the Eclipse bugset. Bugs on the same component tend to belong to the same or several fixed symptom categories. We use stratified sampling according to the components to avoid category imbalance caused by random sampling.

We manually assign bug categories to these 1,700 bugs to form two classification corpora and use them as ground truths. The annotation process is performed by 4 annotators who are either teachers or graduate students in our lab with 4+ years of programming experience. The whole process lasts for one month, which is divided into two stages: training and labeling. In Stage 1, we give all annotators an 1-hour tutorial regarding both of the two bug classification standards. Before annotation, we have labeled 300 bug reports in advance, covering most of the components of Mozilla and Eclipse. Then these annotation examples are equally divided into four parts for annotators to practice. Each annotator is required to pre-label at least 75 bugs with an accuracy of 90%. In the meantime, it's inevitable that in some cases, the annotator disagrees with the categories of annotation examples provided. For these controversial cases (only 11), we discuss with all annotators, reach consensus and add some more detailed definitions to each category.

After a week of training, the annotators have mastered the tagging method and are familiar with the two software projects to be tagged. Then it entered the labeling stage which lasts for 3 weeks. The stage 2 is further divided into three sub stages. In Stage 2.1, each annotator is assigned with 350 bug reports. During this manual annotation process, we ask them to report to us when there are deficiencies, and when certain bugs are hard to be labeled using the bug categories we develop. After annotation, we use the feedback from our annotators to improve our software bug categories, and clean up the annotated data. Then, we let annotators cross validate the data, i.e., the same set of bugs from Stage 2.1 is examined by a different annotator in Stage 2.2. In Stage 2.3, a final sweep to all the annotated data is made by four annotators to improve the consistency of our annotation projects. If the results are inconsistent, they discuss and reach a consensus.

We use Cohen's Kappa coefficient (Vieira et al. 2010) to measure the level of agreement among annotators. Cohen's Kappa coefficient is widely used as a rating of interrater reliability and it represents the extent to which the corpus constructed in our study correctly represents the bug category to be assign. In the annotation, each bug report has been checked at least twice, and we assign annotators according to components, so we no longer calculate the agreement level between annotators, but instead calculate the agreement level between

the last two check results. The value of the Cohen's Kappa coefficient is computed by the following equation:

$$kappa(K) = \frac{P_O - P_E}{1 - P_E} = \frac{0.925 - 0.457}{1 - 0.457} = 0.861 \tag{19}$$

The value of kappa coefficient falling between 0.81 and 1 demonstrates an almost perfect agreement between check results.

4.2.2 Datasets Details

In this article, we first mine the association relations between the two bug attributes (i.e., cause and symptom) from the historical bugset, and then predict the cause category of a new bug according to the association rules.

Figures 5 and 6 summarize the distribution of different bug categories in the Mozilla bugset and Eclipse bugset. The horizontal axis shows the proportion of different symptom categories to the total number of bugs. The vertical axis shows the proportion of different cause categories in a specific symptom category. We notice that, for the symptom categories, the largest proportion is *Capability* (Abb:*CA*) category (28.9% in Mozilla and 28% in Eclipse), that is, the function realization is inconsistent with the design requirements. We analyze all bugs belonged to the *Capability* category. Among them, the main causes are different between the two projects. In the Mozilla project, the main cause is *Logic* (Abb:*LO*, proportion: 28.37%), while in the Eclipse project, the main cause is *Interface* (Abb:*IN*, proportion: 39.29%). This difference is due to the different implementation features of Mozilla and Eclipse. One of Mozilla's main open source projects is Firefox, which is a browser. Therefore, most of the bugs are related to GUI of the browser, for example, some GUI functions were not well implemented. The amount of code modified is not very large, mainly due to some loops, selection statements or logical sequence errors. However, Eclipse is an open source, Java-based, extensible development platform. Therefore, most bugs are caused by calls between services or between plugins, and developers need to modify the parameter settings of interface functions.

In the experiment, we use the manually annotated corpus as the ground truth to train and test all the three classifiers (i.e., *classifier1*, *classifier2* and *classifier3*) in our approach. We can only extract the title and description from a historical bug report to simulate a

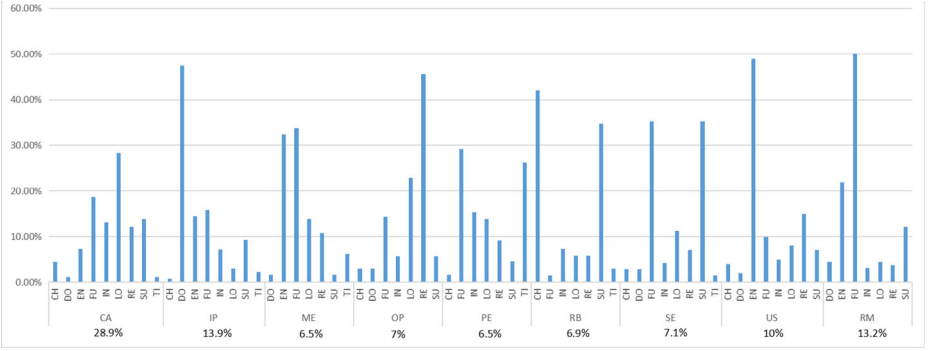


Fig. 5 Distribution of different bug categories from both the cause and symptom perspectives in Mozilla

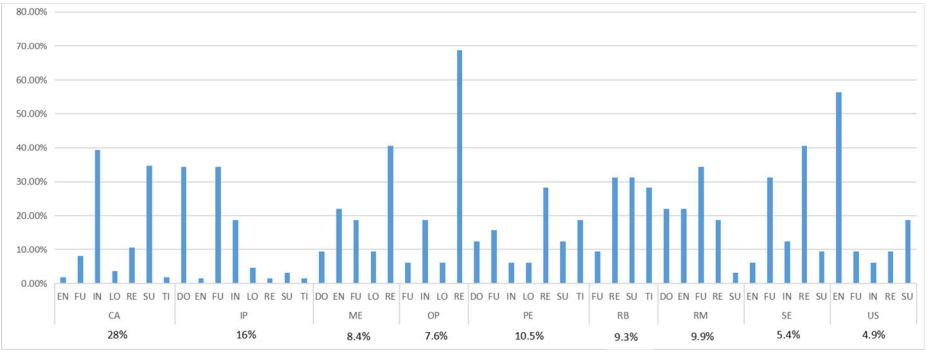


Fig. 6 Distribution of different bug categories from both the cause and symptom perspectives in Eclipse

new bug report. As shown in Table 3, in order to distinguish the datasets used in different experimental sessions, we refer to the extracted corpus (with only the title and description information) as *new bug* corpus, and the original corpus is called *historical bug* corpus.

4.3 Experiment Setup

In our experiments, we use a HAN classifier as the underlying learning algorithm for classifier training and set up three independent classifiers. (1) *classifier1*: we directly use HAN model to classify the text content of a bug report and get the symptom category (i.e., $what_h$ and $what_n$). The model is trained on the symptom corpus (*historical bug*) , and tested on the symptom corpus (*historical bug* and *new bug*) respectively; (2)*classifier2*: we use HAN model, combined with the text and code content of a historical bug report, to classify the historical bug and get its cause category (i.e., why_h). Both training and testing are based on the cause corpus (*historical bug*); (3) *classifier3*: at first, we use HAN model to classify the new bug report (only the title and description) and get the raw cause category of the new bug (i.e., why_{n1}). Both training and testing are based on the cause corpus (*new bug*). Then, we use the association matrix to fine tune the raw cause category probability obtained from the previous test result to get the optimized cause category (i.e., why_n).

4.3.1 Parameter Settings

In the experiment, the training process of these three classification models is similar. HAN runs for about 30 epochs and we select the best model that has best results on the validation set as the final model. The model is then evaluated on the test set by calculating the average accuracy. We zero-pad documents up to a maximum of 30 words per sentence and 100 sentences per document. The parameters are shown as follows:

- The stochastic gradient descent (SGD) with Adam until convergence was used to train the parameters. SGD is an iterative method for optimizing a differentiable objective functions (Ge et al. 2015).
- Our approach used 100 hidden BiGRU cells in each encoder layer. The size of hidden layer in attention mechanism was set to 100, and 100-dimensional word embeddings at every level.
- The learning rate was set to 0.1 and the learning rate was decayed using the rate 0.99.
- We used dropout with 0.5 to prevent over-fitting.

– The optimization weight α_C for cause probability was set to 0.5.

4.3.2 Evaluation Metrics

The performance evaluation metrics for text categorization are mainly *recall* (R), *precision* (P), *F-score*, and *macro-average* and *micro-average* for evaluating accuracy (Pappas 2017). Since bug classification in our approach is a multi-class classification task, we need to combine F-score of all categories for consideration. The micro-F score and macro-F score are widely used for evaluating multi-class classification tasks. For macro-F score, each category has equal weight; for micro-F score, each instance has equal weight. The micro-F score can better reflect the classification performance for given instances in the corpus while macro-F score is better for given categories in the corpus. Because the number of bugs in each category in our corpus is unbalanced, micro-F score is used as the classifier training evaluation metric in the epoch.

$$P^{micro} = \frac{\sum_{C_i \in C} TP_i}{\sum_{C_i \in C} (TP_i + FP_i)} \tag{20}$$

$$R^{micro} = \frac{\sum_{C_i \in C} TP_i}{\sum_{C_i \in C} (TP_i + FN_i)} \tag{21}$$

$$F^{micro} = \frac{2P^{micro}R^{micro}}{P^{micro} + R^{micro}} \tag{22}$$

$$F^{macro} = \frac{1}{|C|} \sum_{C_i \in C} F(C_i) \tag{23}$$

TP_i represents that a sample i originally belongs to positive class and is divided into positive classes by the classification technique; FP_i represents that a sample i originally belongs to negative class but is divided into positive classes; and FN_i represents that a sample i originally belongs to positive class but is divided into negative classes.

4.3.3 Baseline

There are many bug classification methods. Although the specific classification standards are different, they are all based on various multi-class classification models. We use the following state-of-the-art classification models as baselines in our study:

– *SVM* [10], Support Vector Machine (SVM) has been shown to be effective in many past studies in mining software repository. Compared with other machine learning algorithms, it has been proved to perform well in bug classification tasks based on ODC classification scheme (Huang et al. 2015; Thung et al. 2012; Thung et al. 2015).
– *NB*, Naive Bayes (NB) is also a well-known machine learning classification algorithm, which is widely used in many bug report analysis tasks, including ODC-based bug classification tasks (Huang et al. 2015; Thung et al. 2012; Thung et al. 2015).
– *CNN*, as a deep learning algorithm, CNN has been widely used in the field of text classification. We use the basic CNN architecture proposed by Kim (Kim 2014).
– *LSTM*, Long Short Term Memory networks (LSTM) is a typical recurrent neural network (RNN). Compared with traditional RNN, LSTM is good at handling long term dependency and has been used for bug report classification (Ye et al. 2018; Qin and Sun 2018).

[10] http://svmlight.joachims.org/svmmulticlass.html

4.4 Empirical Results

4.4.1 RQ1: HAN vs. other Classification Models

An excellent underlying classifier is essential whether for the mining of association rules or the optimization of cause classification. In our approach, we used the HAN model as the underlying classifier to perform classification for new bugs and historical bugs. We applied five classifiers to classify the same dataset to investigate the performance of HAN model.

For traditional machine learning classifiers, we tested NB and SVM model with word-level TF-IDF vectors. In addition, we also compared the HAN model with two deep learning classifiers (i.e., CNN and LSTM), applying the same word embeddings pre-trained on bugsets. To maintain consistency, all models were trained using the 5-fold cross-validation method.

The comparative results among different classification models are shown in Tables 4 and 5. We notice that the HAN for bug classification outperforms all other models on both Mozilla and Eclipse. On one hand, when HAN is used for historical bugs on Mozilla, compared with the other four models, the micro-F score of symptom classification (i.e., $what_h$) increases by 9.7% and the macro-F score increases by 10.8%, the micro-F score of cause classification (i.e., why_h) increased by 7%, and the macro-F score increases by 15.1%. When HAN is used for historical bugs on Eclipse, the micro-F score of symptom classification (i.e., $what_h$) increases by 14.8% and the macro-F score increases by 14%, the micro-F score of cause classification (i.e., why_h) increased by 15.4%, and the macro-F score increases by 7.4%. We improve the accuracy of historical bug classification as much as possible, to stabilize the cornerstone of causal link mining between symptom and cause categories.

On the other hand, when HAN is used for new bugs, we can also see significant improvements over the other four models, the micro-F score of symptom classification (i.e., $what_n$) on Mozilla increases by 8.5% and the macro-F score increases by 9.5%, the micro-F score of symptom classification (i.e., $what_n$) on Eclipse increases by 9.4% and the macro-F score increases by 6.1%. However, compared with $what_n$, when HAN is directly used for new bug cause classification (i.e., why_n) , we see moderately significant improvements. The micro-F score of why_n on Mozilla increases by 1.6% and the macro-F score increases by 9.6%, the micro-F score of $what_n$ on Eclipse increases by 6.4% and the macro-F score increases by 3.4%.

We further compared the symptom classification performance with the cause performance, and find that there is a same trend in all models. With respect to symptom, the micro-F score and macro-F score of HAN model decreases slightly when the classification

Table 4 Performance of different classification models on Mozilla

Classifier	$what_h$		why_h		$what_n$		why_n[a]	
	micro F	macro F	micro F	macro F	micro F	macro F	micro F	macro F
NB	0.516	0.337	0.465	0.249	0.521	0.321	0.395	0.210
SVM	0.553	0.264	0.439	0.216	0.567	0.313	0.367	0.242
CNN	0.672	**0.486**	**0.657**	0.328	**0.681**	0.469	**0.592**	**0.359**
LSTM	**0.683**	0.440	0.631	**0.374**	0.674	**0.479**	0.567	0.341
HAN	**0.780**	**0.594**	**0.727**	**0.525**	**0.766**	**0.574**	**0.608**	**0.455**

[a]In RQ1, the cause category of a new bug is obtained by classifiers without optimization of association rules

Values in bold denote the highest values

Table 5 Performance of different classification models on Eclipse

Classifier	$what_h$		why_h		$what_n$		why_n	
	micro F	macro F	micro F	macro F	micro F	macro F	micro F	macro F
NB	0.525	0.350	0.483	0.267	0.536	0.383	0.380	0.222
SVM	0.536	0.355	0.516	0.382	0.529	0.425	0.384	0.286
CNN	0.629	**0.448**	0.617	0.417	0.616	0.450	0.518	**0.363**
LSTM	**0.649**	0.441	**0.622**	**0.458**	**0.635**	**0.496**	**0.521**	0.349
HAN	**0.797**	**0.588**	**0.776**	**0.532**	**0.729**	**0.557**	**0.585**	**0.397**

Values in bold denote the highest values

objects changed from historical bugs to new bugs. The other four models even show a small improvement. However, with respect to cause, the micro-F score and macro-F score of all models show a significant decline when varying from historical bugs to new bugs. However, with respect to cause, from historical bugs to new bugs, the micro-F score and macro-F score of each model show a significant decline.

So from the above experimental results, we can conclude that HAN is more effective to classify the symptom and cause for bug data than other state-of-the-art classification models. At the same time, the experimental results also confirm the needs of our work. Only based on the limited information of the new bug report, it is more difficult to analyze the cause of the new bug than the symptom.

4.4.2 RQ2: HAN vs. HAN with Association Rules

According to the discussion of RQ1, there are some obstacles in using traditional approaches to classify the causes of new bugs. We propose a modified HAN classifier to analyze the cause category (i.e., why_n) based on the known symptom category (i.e., $what_n$) and the association rules mining among the above two attributes from historical bugs. We set up a pair of contrast experiments on the two cause classification approaches (HAN with or without association rules) to see whether the causal link is valid.

For the Apriori algorithm applied in this article, there are two parameters (i.e., *minsupp* and *minconf*) defined in Section 3.2.2 need to be adjusted when mining association rules. Some experiments show that when the minsupp is greater than 0.1, the number of association rules obtained without reducing minconf is too small. Considering that there are 18 category labels in the corpus, and the *label cardinality* (average number of related category labels per bug in our work) is only 2, we reduce the values of minsupp (ranging from 0.05 to 0.1) and minconf (ranging from 0.5 to 0.8). When minsupp is set to 0.06 and minconf is set to about 0.6, the number of rules selected is relatively large (23 rules on Mozilla and 20 rules on Eclipse). We evaluate the classification performance applying these two values as shown in Table 6. When HAN-AR is used on Mozilla, the micro-F score of cause classification increases by 6.4% and the macro-F score increases by 5.2%. When HAN-AR is used on Eclipse, the micro-F score increased by 4.9%, and the macro-F score increases by 5.4%. Comparing the two datasets, we can see that the HAN-AR approach is better than HAN in two evaluation metrics.

Table 6 Comparison results of HAN and HAN with association rules (HAN-AR) for new bug cause classification (why_n)

Classifier	Mozilla		Eclipse	
	micro F	macro F	micro F	macro F
HAN	0.608	0.455	0.585	0.397
HAN-AR	**0.672**	**0.507**	**0.634**	**0.451**

Values in bold denote the highest values

In our approach, we combine the confidence of association rules with the pre-trained symptom category probabilities to optimize the cause classification. However, the quality of bug reports is uneven. The length of bug reports with different states varies greatly, from a few words to a few pages. Therefore, we must consider the influence of the content length of bug report on the performance of symptom classification. As shown in Table 3, in the historical bug corpus, on average, there are about 50 sentences and 800 words in each document, which are long texts. In the new bug corpus, on average, there are about 5 sentences and 50 words in each document, which are short texts. HAN reads only one sentence at a time instead of the entire document. In order not to break the integrity of the sentence, we selected the first n sentences of each document to form a series of test sets to observe the trend under different volume of information. Figure 7 shows the classification curve of two classifiers (i.e., HAN and HAN-AR) on two datasets. When we reduce the number of input sentences, the classification performance of HAN decreases significantly. The classification performance of HAN-AR is relatively smooth, especially when the number of sentences is about 10, that is, when it is equivalent to the content of new bug reports, it is far beyond the performance of HAN.

So from our study, we can conclude that there is indeed a measurable causal link between the cause and symptom categories of bugs, which is useful to improve the cause classification for new bugs.

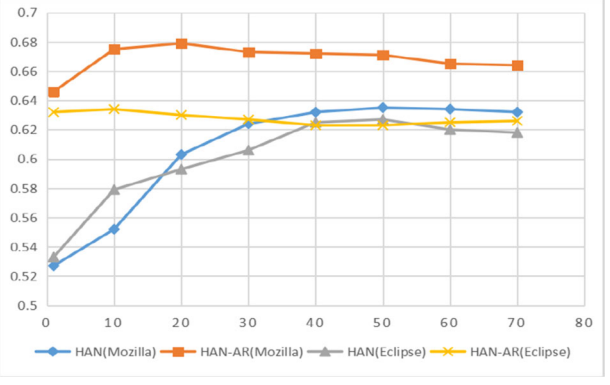
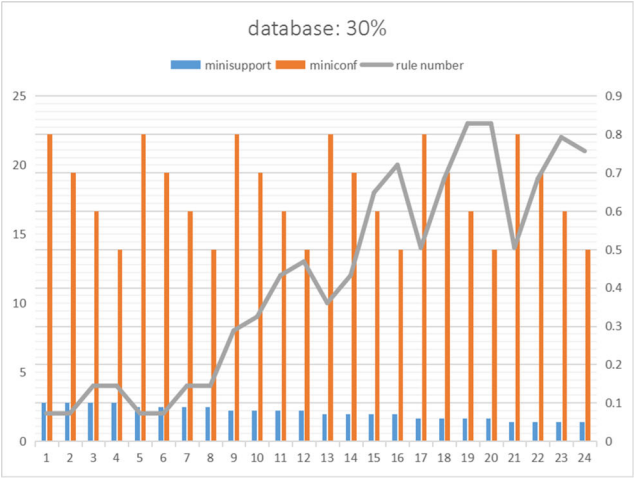


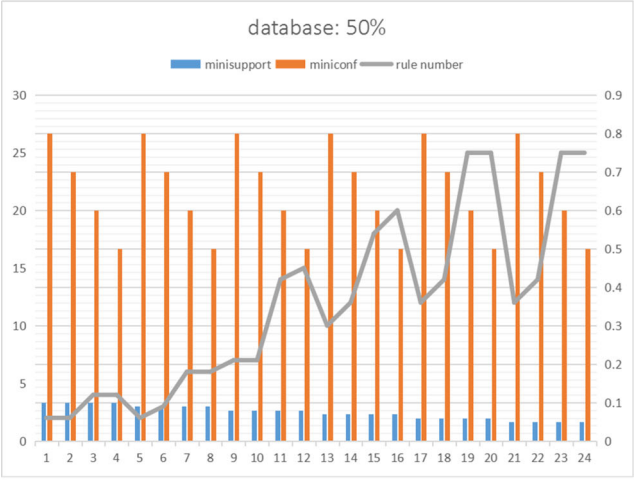
Fig. 7 Performance of HAN-AR for different lengths of bug reports

4.4.3 RQ3: *Bugclass* with Association Rules Mining from Different Amount of Historical Bugs

For RQ2, in order to be more comprehensive and objective, we mine association rules from larger bugsets rather than corpora to optimize cause classification. In order to further explore the impact of attribute database size on the approach, we collect attribute sub-databases of different scales from the bugset by stratified sampling according to components. Bugs on the same component tend to belong to the same or several fixed symptom categories. We use stratified sampling instead of random sampling to ensure an experimental condition of association rules mining as far as possible, that is, the distribution of attribute categories will not change with the varying amount of attribute database. At the same time, all the sub-databases do not contain the bug reports in the corpus. The corpus is used as training set and



(a)



(b)

Fig. 8 Number of association rules mined in different scale databases in Mozilla

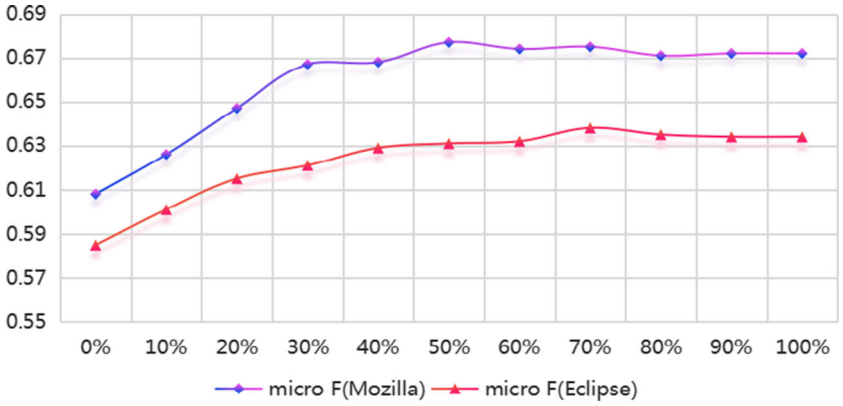


Fig. 9 Performance of HAN-AR with association rules from varying amount of attribute databases (0% means HAN without association rules. 10%-90% represent 9 sub-databases sampled from the bugset with different proportions. 100% represents the complete bugset)

test set, if the intersection of test set and attribute sub-database is too large, the accuracy of test will be affected.

We first observe the change of the number of association rules with the increasing size of database. It can be seen from Fig. 8 that: when minsupp is fixed, the number of association rules decreases with the increase of minconf; when minconf is fixed, the number of association rules decreases with the increase of minsup. Given the same minsupp and minconf, the numbers of association rules mined in different scale databases are basically the same.

We continue to compare the cause classification performance with different association rules. As shown in Fig. 9, for example, when minsupp takes 0.06 and minconf takes 0.6, HAN-AR can get the maximum micro-F score (0.628) on the Mozilla 10% database. When minsupp is 0.07 and minconf is 0.6, HAN-AR can get the maximum micro-F score (0.677) on the Mozilla 30% database. When minsupp takes 0.06 and minconf takes 0.7, HAN-AR can get the maximum micro-F score (0.669) on the Mozilla 50% database. Observing the overall trend, although the curve grows a little fast when the database is relatively small (less than 20%), it becomes stable and fluctuates slightly when the database exceeds 35%. So from our study, we can conclude that the association between bug symptom and cause is stable, and it is less affected by the size of database to be mined.

4.5 Use Case Discussion

In this section, We explore whether identifying the symptom and cause categories of new bugs is beneficial to bug fixing work. Take duplicate bug reports detection task as an example shown in Fig. 10, we combine two bug categories (i.e., $what_n$ and why_n) as additional features with the state-of-the-art approach (i.e., *DBR-CNN* (Xie et al. 2018)) to study the effectiveness of *Bugclass*. *DBR-CNN* can well modify the traditional CNN with some bug domain-specific features (e.g., component, create time, and priority), especially for small datasets.

BugRepo [11] maintains a collection of bug reports that are publicly available for research purposes. Considering that the corpus for *Bugclass* is built in Mozilla, we

[11] https://github.com/LogPAI/bugrepo

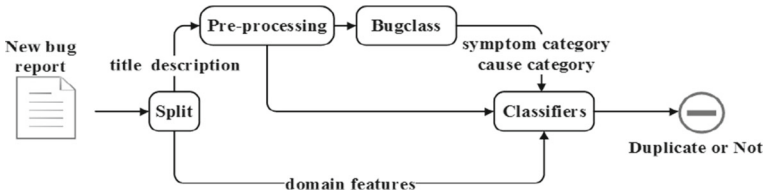


Fig. 10 Duplicate bug reports detection with cause category and symptom category as features

select three of the projects belonging to Mozilla as datasets from BugRepo, which include Mozilla Core, Firefox, and Thunderbird. To verify whether the additional category feature-enhanced model can improve the performance of original model, we conduct the comparison on the three datasets, with results depicted in Fig. 11. We can notice that introducing bug categories as features consistently outperform the original *DBR-CNN* model on all the datasets. When we consider the two bug categories separately, introducing the symptom category as a feature is slightly better than the cause category. In addition, we observe that F1-measure presents largest when considering bug category from both the two dimensions, achieves 0.91, 0.892, and 0.914 for Mozilla Core, Firefox, and Thunderbird, respectively, increasing original DBR-CNN by 4.3%, 3.7%, and 4.4%. This shows that for new bugs, our approach *Bugclass* can not only effectively identify the symptoms of bugs, but also effectively predict the causes of the bugs.

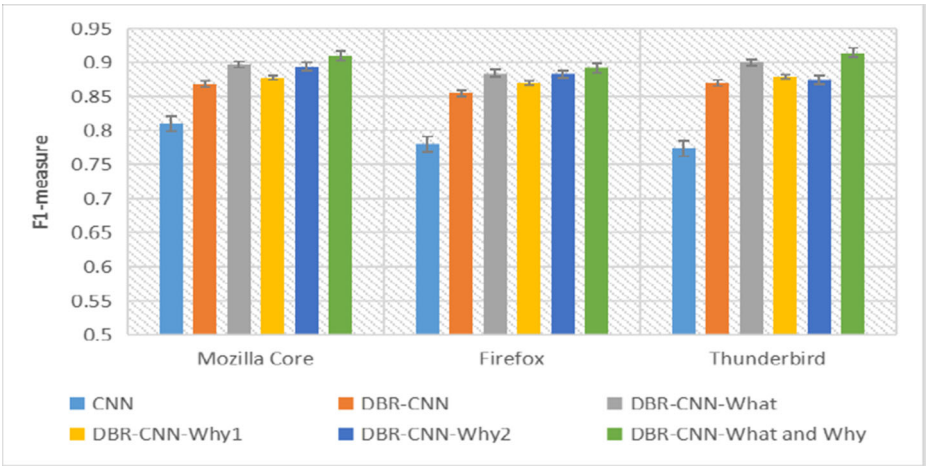


Fig. 11 Performance comparison with or without bug categories as features (Why1 is the raw cause category, and Why2 is the cause category optimized by association rules)

5 Threats to Validity

We identified the following threats that would affect the results of our study:

Construct threats: In our work, our findings are based on the micro-F score metric and the macro-F score metric, and other evaluation measures may yield different results. However, these metrics are widely used to evaluate classification techniques (Pappas 2017).

Internal threats: (1) The problem of unbalanced samples in text categorization: in the distribution of *symptom* and *cause* category samples, there are also categories with less than 5%. Although we have added a small number of samples of the categories with a small proportion, it is impossible to make major adjustments considering the actual distribution characteristics of bugs; (2) Manual classification is subjective and there may be a small number of labeling errors; (3) We only annotated 1,700 bugs, and the small size of training set and validation set would affect the accuracy of the classification.

External threats: We examined the characteristics of bugs from open source projects, including Eclipse and Mozilla. Both of them are application software, which cannot represent the characteristics of other types of software such as operating system software.

6 Related Work

In this article, deep learning and association rule mining technologies are combined to optimize the bug cause classification with the aim of supporting and possibly accelerating the bug comprehension activities. Therefore, in this section, we mainly discuss the related research and application of bug classification technology and bug association rule mining technology.

6.1 Bug Classification

Classification aided bug repair has always been the focus of bug research. On the one hand, researchers have conducted qualitative and quantitative analysis of historical bugs and submitted many empirical studies. Jr. et al. performed an empirical study of 502 bug reports from 19 bug repositories, to understand the root causes and impact of client-side JavaScript faults and how the results can impact JavaScript programmers, testers and tool developers (Ocariza et al. 2017). Zhang et al. studied deep learning applications built on top of TensorFlow, examine the root causes and symptoms of these bugs, and also proposed a number of challenges for their detection and localization (Zhang et al. 2018). Catolino et al. analyzed 1,280 bug reports of 119 popular projects to comprehend nine main root cause of bugs in general, and argued that bugs should be assigned according to their root cause type (Catolino et al.2019).

On the other hand, many studies keep on surveying new techniques to better perform automatic bug classification to approximate the true bug classification with much less human effort. Podgurski et al. used the cluster analysis algorithm to group failures together automatically and confirmed that failures with the same cause tend to form fairly cohesive clusters (Podgurski et al. 2003). Su et al. proposed to use the decision tree C4.5 algorithm to classify the invalid bug types (Su et al. 2017). Patil et al. computed the semantic similarity between the bug type labels, and proposed to use Explicit Semantic Analysis (ESA) to

carry out concept-based classification of software bug reports (Patil 2017). Terdchanakul et al. built classification models with logistic regression and random forest using features from N-gram IDF and topic modeling to identify bugs and nonbugs (Terdchanakul et al. 2017). Shu et al. further improved the accuracy of security bug report classification through parameter tuning of machine learning learners and data pre-processor (Shu et al. 2019). Zafar et al. constructed a hand-labeled dataset from real GitHub commits according to the number of fixed bugs per file and the number of fixed files per bug, and fine-tuned BERT (Bidirectional Encoder Representations from Transformers) for classification (Zafar et al. 2019) . Ni et al. studied the main cause types in commit files, and applied the TBCNN model to realize automatically classification of diff ASTs (Ni et al. 2020) .

6.2 Bug Analysis with Association Rule Mining

Much effort has been devoted to study the correlation between bugs to help developers detect and repair software bugs. Researchers leverage data mining algorithms to extract relational association rules from real large projects. Some work infers frequent patterns from source code and regards such patterns as the implicit rules that developers should follow in coding. Any violation of these rules will be detected as potential bugs. The inferred patterns can be either positive or negative. For example, PR-Miner (Li and Zhou 2005) and AntMiner (Liang et al. 2016) extract positive association rules that enforce paired appearances of program behaviors. Instead, NAR-Miner (Bian et al. 2018) extract negative association rules between program elements (i.e., function calls or condition checks) which are mostly neglected.

Some work try to capture various kinds of relationships between bug attributes, so as to improve bug analysis tasks. Song et al. used the Apriori algorithm to mine the association rules between different cause classes of bugs (Song et al. 2006). Similarly, Wang et al. proposed five rules to identify correlated crash types automatically to locate and rank buggy files (Wang et al. 2016). Sharma et al. applied the association rules as features of clustering method to predict bug assignee based on the bug's attributes (i.e., severity, priority, component and operating system) (Sharma et al. 2018). Shao et al. presented a novel software defect prediction model based on correlation weighted class association rule mining (CWCAR) which can better handle class imbalance (Shao et al. 2020). Tan et al. proposed work closest to ours. They manually study bugs in three dimensions (i.e., root causes, impacts and components), and further study the correlation between categories in different dimensions, and the trend of different types of bugs (Tan et al. 2014).

7 Conclusion and Future Work

Bug fixing is an important activity in software maintenance. Before fixing bugs, developers need to understand the bug, such as what happened to the software and why the bug happened in the software. In this article, we first designed a new bug classification criterion from two dimensions - *symptom* and *cause*. Then, we proposed an approach, *BugClass*, which applies a deep neural network classification approach based on the HAN model to automatically classify bugs into different symptom and cause categories. Finally, we explored the causal links among categories to further improve the accuracy of the bug classification model. Experimental results demonstrate that *BugClass* outperform traditional classification approaches for analyzing causes of new bugs.

In future work, we plan to further improve the accuracy of cause classification in *BugClass* by introducing fine-grained code diff features. We also plan to conduct a survey to get feedback from real developers on the findings of this study. On this basis, we can extend *BugClass* to dig deeper into the connection between the *why*, *what* and *how* information for bugs to help developers better understand bugs and fix bugs efficiently.

Acknowledgements This work is supported by the National Natural Science Foundation of China (No.61872312, No.61972335, No.62002309); the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project (No.YZU201803, No. YZU201902); the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053), the Jiangsu "333" Project; the Natural Science Foundation of the Jiangsu Higher Education Institutions of China (No. 20KJB520016); the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No.KFKT2020B15, No.KFKT2020B16) and Yangzhou University Top-level Talents Support Program (2019) .

References

2010 IEEE standard classification for software anomalies. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) pp 1–23

Adamson GW, Boreham J (1974) The use of an association measure based on character structure to identify semantically related pairs of words and document titles. Information Storage and Retrieval 10(7-8):253–260

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Bocca JB, Jarke M, Zaniolo C (eds) VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago de Chile, Chile, Morgan Kaufmann, pp 487-499

Avizienis A, Laprie J, Randell B, Landwehr CE (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Sec Comput 1(1):11–33

Azmi M, Runger GC, Berrado A (2019) Interpretable regularized class association rules algorithm for classification in a categorical data space. Inf Sci 483:313–331

Basili VR, Selby RW (1987) Comparing the effectiveness of software testing strategies. IEEE Trans Software Eng 13(12):1278–1296

Beizer B (1990) Software testing techniques. 2 edn, Van Nostrand Reinhold

Bian P, Liang B, Shi W, Huang J, Cai Y (2018) Nar-miner: discovering negative association rules from code for bug detection. In: Leavens GT, Garcia A, Pasareanu CS (eds) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, ACM, pp 411-422

Bòhme M, Soremekun EO, Chattopadhyay S, Ugherughe E, Zeller A (2017) 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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp 117–128

Bugde S, Nagappan N, Rajamani SK, Ramalingam G (2008) Global software servicing: Observational experiences at microsoft. In: 3rd IEEE International Conference on Global Software Engineering, ICGSE 2008, Bangalore, India, 17-20 August, vol 2008, pp 182-191

Campos EC, de Almeida Maia M (2017) Common bug-fix patterns: A large-scale observational study. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017, pp 404–413

Catolino G, Palomba F, Zaidman A, Ferrucci F (2019) Not all bugs are the same: Understanding, characterizing, and classifying bug types. J Syst Softw 152:165–181

Chaparro O, Lu J, Zampetti F, Moreno L, Penta MD, Marcus A, Bavota G, Ng V (2017) Detecting missing information in bug descriptions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp 396-407

Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J (1996) Change detection in hierarchically structured information. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996, pp 493–504

Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong M (1992) Orthogonal defect classification - A concept for in-process measurements. IEEE Trans Software Eng 18(11):943–956

Davies S, Roper M (2014) What's in a bug report? In: 2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014, pp 26:1–26:10

Dey R, Salem FM (2017) Gate-variants of gated recurrent unit (GRU) neural networks. arXiv:1701.05923

Falleri J, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pp 313-324

Gao S, Young MT, Qiu JX, Yoon H, Christian JB, Fearn PA, Tourassi GD, Ramanthan A (2018) Hierarchical attention networks for information extraction from cancer pathology reports. JAMIA 25(3):321–330

Ge R, Huang F, Jin C, Yuan Y (2015) Escaping from saddle points - online stochastic gradient for tensor decomposition. In: Proceedings of The 28th Conference on Learning Theory, COLT 2015, Paris, France, July, 3-6, 2015, pp 797-842

Hamill M (2015) Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system. Soft Quality J 23(2):229–265

Han J, Kamber M, Pei J (2011) Data Mining: Concepts and Techniques, 3rd edn. Morgan Kaufmann

He J, Wang B, Fu M, Yang T, Zhao X (2019) Hierarchical attention and knowledge matching networks with information enhancement for end-to-end task-oriented dialog systems. IEEE Access 7:18871–18883

Hermann KM, Blunsom P (2014) Multilingual models for compositional distributed semantics. In: Proceedings of the 52nd annual meeting of the association for computational linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, vol 1. Long Papers, pp 58–68

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27–28, 2018, pp 200-210

Huang L, Ng V, Persing I, Chen M, Li Z, Geng R, Tian J (2015) Autoodc: Automated generation of orthogonal defect classifications. Autom Softw Eng 22(1):3–46

Humphrey WS (1995) A discipline for software engineering. Series in software engineering. Addison-Wesley

Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (2018) Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018, pp 298-309

Joachims T (1998) Text categorization with support vector machines: Learning with many relevant features. In: Machine Learning: ECML-98, 10th European Conference on Machine Learning, Chemnitz, Germany, April 21-23, 1998, Proceedings, pp 137–142

Kalchbrenner N, Grefenstette E, Blunsom P (2014) A convolutional neural network for modelling sentences. In: Proceedings of the 52nd annual meeting of the association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 1: Long Papers, pp 655–665

Kim S, Whitehead Jr. EJW (2006) How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006, pp 173-174

Kim Y (2014) Convolutional neural networks for sentence classification. In: Moschitti A, Pang B, Daelemans W, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing EMNLP2014 October 25-29 (eds). ACL, pp 1746–1751

Le XD, Chu D, Lo D, Le Goues C, Visser W (2017) S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp 593-604

Le XD, Thung F, Lo D, Le Goues C (2018) Overfitting in semantics-based automated program repair. Empir Softw Eng 23(5):3007–3033

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu PT, Forrest S, Weimer W (2015) The manybugs and introclass benchmarks for automated repair of C programs . IEEE Trans Software Eng 41(12):1236–1256

Li Z, Zhou Y (2005) Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Wermelinger M, Gall HC (eds) Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, ACM, pp 306–315

Liang B, Bian P, Zhang Y, Shi W, You W, Cai Y (2016) Antminer: mining more bugs by reducing noise interference. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. ACM, pp 333-344

Liu L, Yu S, Wei X, Ning Z (2018) An improved apriori-based algorithm for friends recommendation in microblog. Int J Commun Sys 31(2)

Mäntylä M, Lassenius C (2009) What types of defects are really discovered in code reviews? IEEE Trans Software Eng 35(3):430–448

Motwani M, Sankaranarayanan S, Just R, Brun Y (2018) Do automated program repair techniques repair hard and important bugs? Empir Softw Eng 23(5):2901–2947

urphy-Hill ER, Zimmermann T, Bird C, Nagappan N (2015) The design space of bug fixes and how developers navigate it. IEEE Trans Software Eng 41(1):65–81
Nayrolles M, Hamou-Lhadj A (2018) Towards a classification of bugs to facilitate software maintainability tasks. In: Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies, SQUADE@ICSE 2018, Gothenburg, Sweden, May 28, 2018, pp 25-32
Ni Z, Li B, Sun X, Chen T, Tang B, Shi X (2020) Analyzing bug fix for automatic bug cause classification. J Syst Softw 163:110538
Ocariza FS, Bajaj K, Pattabiraman K, Mesbah A (2017) A study of causes and consequences of client-side javascript bugs. IEEE Trans Software Eng 43(2):128–144
Pappas N (2017) Popescu-Belis A Multilingual hierarchical attention networks for document classification. arXiv:1707.00896
Patil S (2017) Concept-based classification of software defect reports. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, pp 182-186
Podgurski A, Leon D, Francis P, Masri W, Minch M, Sun J, Wang B (2003) Automated support for classifying software failure reports. In: Proceedings of the 25th International Conference on Software Engineering, May 3-10. Portland, Oregon USA, 465–477, vol 2003
Qin H, Sun X (2018) Classifying bug reports into bugs and non-bugs using LSTM Proceedings of the Tenth Asia-Pacific Symposium on Internetware, Internetware 2018, Beijing, China, September 16-16, 2018. ACM, pp 20:1–20:4
Shan L, Li Z, Feng Q, Lin T, Zhou P, Zhou Y (2005) Bugbench: Benchmarks for evaluating bug detection tools. Workshop on the Evaluation of Software Defect Detection Tools
Shao Y, Liu B, Wang S, Li G (2020) Software defect prediction based on correlation weighted class association rule mining. Knowl Based Syst 196:105742
Sharma M, Tandon A, Kumari M, Singh VB (2018) Reduction of redundant rules in association rule mining-based bug assignment. arXiv:1807.08906
Shepperd MJ (1993) Practical software metrics for project management and process improvement: R grady prentice-hall (1992) $30.95 282 pp ISBN 0 13 720384 5. Inf Softw Technol 35(11-12):701
Shu R, Xia T, Williams L, Menzies T (2019) Better security bug report classification via hyperparameter optimization. arXiv:1905.06872
Song Q, Shepperd MJ, Cartwright M, Mair C (2006) Software defect association mining and defect correction effort prediction. IEEE Trans Software Eng 32(2):69–82
Soto M, Le Goues C (2018) Using a probabilistic model to predict bug fixes. In: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, pp 221-231
Su Y, Luarn P, Lee Y, Yen S (2017) Creating an invalid defect classification model using text mining on server development. J Syst Softw 125:197–206
Sun X, Peng X, Zhang K, Liu Y, Cai Y (2019) How security bugs are fixed and what can be improved: an empirical study with mozilla. Sci China Inf Sci 62(1):19102:1–19102:3
Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empir Softw Eng 19(6):1665–1705
Tarnpradab S, Liu F, Hua KA (2018) Toward extractive summarization of online forum discussions via hierarchical attention networks. arXiv:1805.10390
Terdchanakul P, Hata H, Phannachitta P, Matsumoto K (2017) Bug or not? bug report classification using n- gram IDF. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17–22, 2017, pp 534-538
Thung F, Lo D, Jiang L (2012) Automatic defect categorization. In: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012, pp 205-214
Thung F, Le XD, Lo D (2015) Active semi-supervised defect categorization. In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015, pp 60-70
Timperley CS, Stepney S, Le Goues C (2018) Bugzoo: a platform for studying software bugs. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp 446-447
van Tonder R, Le Goues C (2018) Static automated program repair for heap properties. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June, 03, 2018, pp 151–162
Vieira SM, Kaymak U, Sousa JMC (2010) Cohen's kappa coefficient as a performance measure for feature selection. In: FUZZ-IEEE 2010, IEEE International Conference on Fuzzy Systems, Barcelona, Spain, 18-23 July, 2010 Proceedings. IEEE, pp 1-8

Wang L, Sun X, Wang J, Duan Y, Li B (2017) Construct bug knowledge graph for bug resolution: poster. In: Proceedings of the 39th international conference on software engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion, pp 189–191

Wang S, Khomh F, Zou Y (2016) Improving bug management using correlations in crash reports. Empir Softw Eng 21(2):337–367

Wang SI, Manning CD (2012) Baselines and bigrams: Simple, good sentiment and topic classification. In: The 50th annual meeting of the association for computational linguistics, proceedings of the conference, July 8-14, 2012, Jeju Island, Korea - Volume 2: Short Papers, pp 90–94

Wen M, Chen J, Wu R, Hao D, Cheung S (2018) Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp 1-11

Xia X, Lo D (2017) An effective change recommendation approach for supplementary bug fixes. Autom Softw Eng 24(2):455–498

Xie Q, Wen Z, Zhu J, Gao C, Zheng Z (2018) Detecting duplicate bug reports with convolutional neural networks. In: 25th Asia-Pacific software engineering conference, APSEC 2018, Nara, Japan, December 4-7, 2018, pp 416-425

Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L (2017) Precise condition synthesis for program repair. In: Proceedings of the 39th international conference on software engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp 416-426

Yang Z, Yang D, Dyer C, He X, Smola AJ, Hovy EH (2016) Hierarchical attention networks for document classification. In: NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016, pp 1480–1489

Ye X, Fang F, Wu J, Bunescu RC, Liu C (2018) Bug report classification using LSTM architecture for more accurate software defect locating. In: 17th IEEE International conference on machine learning and applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018, pp 1438–1445

Zafar S, Malik MZ, Walia GS (2019) Towards standardizing and improving classification of bug-fix commits. In: 219 ACM/IEEE international symposium on empirical software engineering and measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19–20, 2019. IEEE, pp 1–6

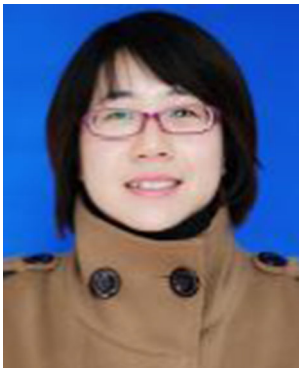
Zhang Y, Chen Y, Cheung S, Xiong Y, Zhang L (2018) An empirical study on tensorflow program bugs. In: Tip F, Bodden E (eds) Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. ACM, pp 129-140

Zhong H, Mei H (2018) Mining repair model for exception-related bug. J Syst Softw 141:16–31

Zhou C, Li B, Sun X, Guo H (2018) Recognizing software bug-specific named entity in software bug repository. In: Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018, pp 108-119

Zhou C, Li B, Sun X (2020) Improving software bug-specific named entity recognition with deep neural network. J Syst Softw 165:110572

Zhou T, Sun X, Xia X, Li B, Chen X (2019) Improving defect prediction with deep forest. Inf Softw Technol 114:204–216



Cheng Zhou is a PhD student in School of Information Engineering, Yangzhou University. Her current research interests include intelligent bug fixing and software data analysis.

Bin Li is a professor in School of Information Engineering, Yangzhou University in China. His current research interests include software engineering, artificial intelligence, etc.

Xiaobing Sun is a professor in School of Information Engineering, Yangzhou University in China. His current research interests include intelligent software engineering, software data analytics.

Lili Bo is a lecturer in School of Information Engineering, Yangzhou University. Her current research interests include software testing, software security.

Affiliations

Cheng Zhou[1,2] · Bin Li[1] · Xiaobing Sun[1,3] · Lili Bo[1,3]

Cheng Zhou
canorcheng@foxmail.com; 153362746@qq.com

Bin Li
lb@yzu.edu.cn

Lili Bo
lilibo@yzu.edu.cn; 007245@yzu.edu.cn

1 School of Information Engineering, Yangzhou University, Yangzhou, China

2 Taizhou University, Taizhou, China

3 Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China