The Journal of Systems and Software 163 (2020) 110538

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Analyzing bug fix for automatic bug cause classification

Zhen Ni[a], Bin Li[a], Xiaobing Sun[a,b,*], Tianhao Chen[a], Ben Tang[a], Xinchen Shi[a]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 6 October 2019
Revised 13 December 2019
Accepted 27 January 2020
Available online 28 January 2020

Keywords:
Bug analysis
Fix tree
Bug cause classification
TBCNN

ABSTRACT

During the bug fixing process, developers usually need to analyze the source code to induce the bug cause, which is useful for bug understanding and localization. The bug fixes of historical bugs usually reflects the bug causes when fixing them. This paper aims at exploiting the corresponding relationship between bug causes and bug fixes to automatically classify bugs into their cause categories. First, we define the code-related bug classification criterion from the perspective of the cause of bugs. Then, we propose a new model to exploit the knowledge in the bug fix by constructing fix trees from the diff source code at Abstract Syntax Tree (AST) level, and representing each fix tree based on the encoding method of Tree-based Convolutional Neural Network (TBCNN). Finally, the corresponding relationship between bug causes and bug fixes is analyzed by automatically classifying bugs into their cause categories. We collected 2000 real-world bugs from two open source projects Mozilla and Radare2 to evaluate our approach. The experimental results show the existence of observational correlation between the bug fix and the cause of the historical bugs, and the proposed fix tree can effectively express the characteristics of the historical bugs for bug cause classification.

1. Introduction

During software development, an important task for software developers is to fix software bugs.[1] There are a large number of historical fixed bugs in various repositories, such as the defect tracking system and the platform hosting software projects. Thung et al. (2015) proposed that categorizing historical bugs brings obvious benefits for better managing and understanding bugs. With a better understanding, one can then plan the best course of action to minimize the future impact of bugs. In fact, many classification schemes, such as Orthogonal Defect Classification (ODC) (Chillarege et al., 1992) and Common Weakness Enumeration (CWE) (Martin, 2007), have been widely used to manage bugs in various software projects (Zheng et al., 2006).[2] In the case of fixing a new bug, the developer usually first searches for similar bugs to find inspiration for a possible fix. However, many of these similar historical bugs used repeated repair patterns. Therefore, developers often need to spend a lot of time reading the bug fix[3] of historical bugs to filter out the best solution. This greatly reduces the efficiency of bug fixing. Kreutzer et al. (2016) pointed out that bug modification types are associated with special kinds of faulty source code. In other words, the knowledge in the bug fix provides essential insights into the causes of bugs. Furthermore, we manually observed that bugs of a certain cause type correspond to certain AST-level modifications in the bug fix. Specifically, through investigating the bug fixes of the fixed bugs combining with their bug reports and commit messages, we empirically find that certain fix operations and corresponding code elements at AST-level are closely related to certain cause types of bugs. For example, one fixing "adds an 'if' statement and the if conditional expression here" is a check for null or boundary values. The cause type for the bug corresponding to this fix is ignoring the extreme conditions. We then did a preliminary statistical analysis to verify the observed correlation, which will be detailed in Section 4.3. Therefore, if these historical fixed bugs have been classified into their cause categories, and there are specific fix regularities (such as fix patterns) for bugs in each cause type, the efficiency of bug fixing can be improved. Therefore, in this paper we use the bug cause category as the classification system/architecture for the historical fixed bugs, and use the knowledge extracted from bug fix of the historical fixed bugs to automatically identify the causes of these bugs.

As mentioned before, the source code of historical fixed bugs implies a lot of empirical knowledge, such as fix patterns

* Corresponding author at: School of Information Engineering, Yangzhou University, Yangzhou, China.
E-mail addresses: lb@yzu.edu.cn (B. Li), xbsun@yzu.edu.cn (X. Sun).
[1] In this paper, we use the terms "defect", "error", and "bug" as synonyms.
[2] National, vulnerability database. https://nvd.nist.gov/.
[3] In this paper, we use the terms "bug fix", "code patch", "diff source code", and "fixing source code" as synonyms.

https://doi.org/10.1016/j.jss.2020.110538
0164-1212/© 2020 Elsevier Inc. All rights reserved.

(Sun et al., 2019a; 2018; 2017b). Le et al. (2016) proposed that previously-appearing fix patterns can provide useful guidance to an automated repair technique. Zhong and Meng (2017) systematically designed six overlap metrics, and performed an empirical study on 5735 bug fixes to investigate the usefulness of past fixes when composing new fixes. Therefore, mining and representing knowledge in bug fix plays an important role in advancing the field of automatic program repair. While traditional tools for source code analysis in the bug fix are static (pre-runtime) (Xu et al., 2010) or dynamic (runtime) (Luk et al., 2005) analysis on programs, these tools are typically used to detect a few types of bugs based on predetermined rules. For example, Xu et al. (2010) used rules, such as the memory locations and values of the elements of each expression, to precisely track different memory object values in C programs for automatic bug finding. With the widespread availability of open source repositories in recent years, it has become a trend to use data-driven technology to analyze source code. Nguyen et al. (2013) introduced SLAMC, a novel statistical semantic language model for source code. It incorporates semantic information into code tokens and models the patterns of such semantic annotations. Their work combines the local context in semantic n-grams with the global technical functionality into an n-gram topic model, together with pairwise associations of program elements. Based on SLAMC, they developed a new code suggestion method, which is evaluated to have a relatively higher accuracy than the state-of-the-art approach. Xia and Lo (2017) presented an effective approach, SupLocator, to recommend relevant locations that need to be changed for supplementary bug fixes. In particular, SupLocator extracts six change relationship graphs according to various relationships between methods, classes, and packages in the source code. The semantic annotation proposed in (Nguyen et al., 2013) and the nodes in the change relationship graph defined in (Xia and Lo, 2017) inspires us to use the code element types when mining the knowledge in the fixing source code. Zhong and Meng (2018) created delta dependency graphs (i.e., program dependency graphs for code changes) for each bug fix, and identified how bug fixes overlap with each other in terms of the content, code structure, and identifier names of fixes.

From the above work, we see that categorizing the historical fixed bugs helps developers understand the fixes of the similar historical bugs and modeling the bug fix of historical bugs helps developers generate effective fixes. According to our empirical observation, the fix of a bug has a corresponding relationship with the bug cause type. Therefore, in this paper, we focus on constructing a representation method for bug fixes, and exploiting the corresponding relationship between bug causes and bug fixes to automatically classify bugs into categories. Specifically, our goal is to construct a representation model for the bug fix of the historical bugs and generate a cause classification model for the historical fixed software bugs. To achieve this goal, we need to solve the following challenges.

(1) To the best of our knowledge, there is not a study which work on leveraging knowledge in the fixing source code to identify the cause category of the historical fixed bugs. It is difficult to mine the connection between fixing source code and the cause of the bug.
(2) Although there have been many studies on mining and analyzing patterns in source code of bugs in recent years (Zhong and Mei, 2018; Raychev et al., 2014; Ray et al., 2016; Bhoopchand et al., 2016; Campos and de Almeida Maia, 2017; Liu et al., 2017). There are few on building a representation model for the bug fix by incorporating lexical information (code tokens), concept information (code element types) and fix structures in the fixing source code. In addition, code terms, code element types and structures are diverse and complex in source code (Chang et al., 2018), making it difficult to model them.

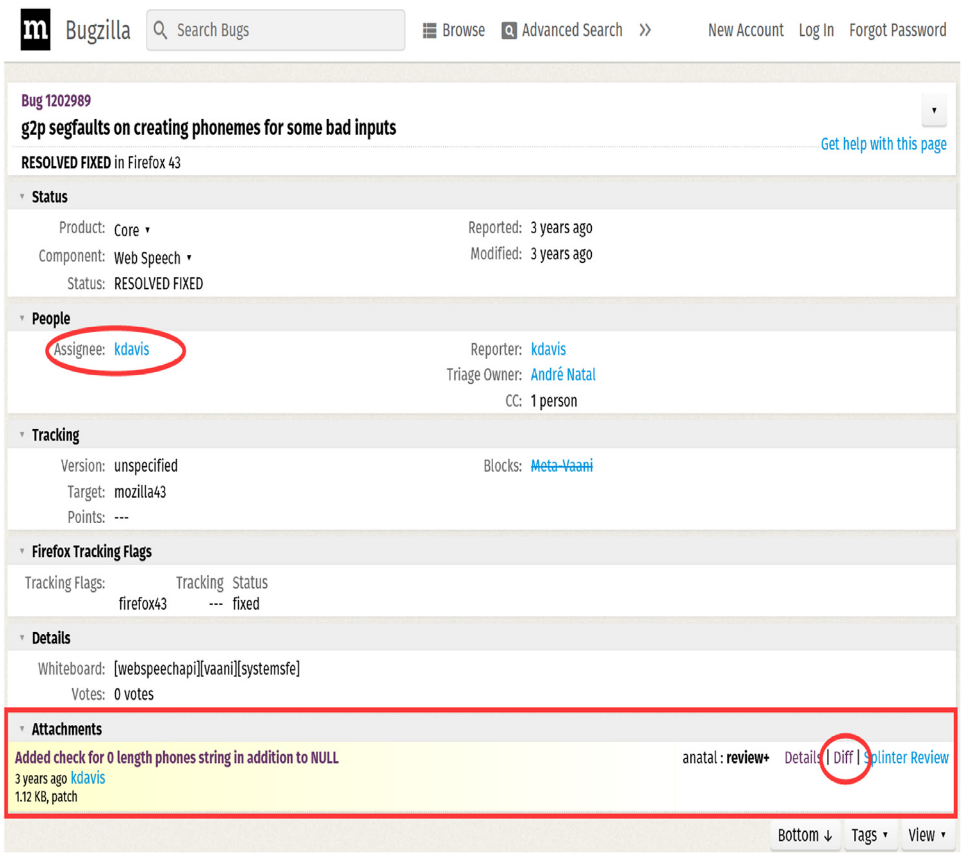
In order to solve the above challenges, combining the Orthogonal Defect Classification (ODC) method (Chillarege et al., 1992) and the defect classification standards (Ieee, 1994) proposed by IEEE, we extracted the categories related to source code and used them to construct a bug classification criterion. In this empirical analysis process, 400 bugs with bug fix in C language were sampled from the open source projects Mozilla[4] and Radare2[5], 1600 bugs with bug fix in Java language were sampled from open source products Android Firefox and Rhino in Mozilla. We read the description of these bug reports and their commit messages from their assignees. According to the classification criterion mentioned before, the bugs were classified into their respective cause types by manual labeling and cross-validation. In particular, we conducted a preliminary statistical analysis to verify the existence of the correlation between bug causes and the AST-level modifications in the bug fixes. Before training, we defined the fix tree for the fixing source code, and we extracted fix trees from the diff source code by leveraging an AST-based code diff tool (Falleri et al., 2014). Specifically, we extract the lexical code tokens, concept types of the code elements and fixing structures to generate the fix tree at the AST level. Therefore, the knowledge consisting of code elements, concept types and structures of the fixing source code were extracted. We convert the above knowledge into vector representations based on the Tree-based Constitutional Neural Network (TBCNN) (Mou et al., 2016). Then, we propose an approach to automatically classify the historical fixed bugs into the corresponding bug cause types based on the fix tree. Finally, we conducted an empirical study to validate the correspondence between the fix tree and the classification criterion. The results show that it is effective to automatically identify the cause of bugs through the proposed fix tree. The main contributions of this paper are as follows:

(1) We manually investigated the bug reports, commit messages and bug fixes of a dataset consisting of 2000 historical fixed bugs. Specifically, we first combine existing bug categorization criteria (Chillarege et al., 1992), (Ieee, 1994) to define the software bug classification criterion from the bug cause perspective. Table 1 shows the categories we use and the observational fix operations. Using the classification criterion, we did a manual labeling for each bug in this dataset by investigating the bug reports and commit messages. Then, we did a preliminary statistical analysis on 1089 bugs in this dataset to verify the existence of the correlation between bug cause types and the AST-level modifications in the bug fix.
(2) We construct a tree structure (i.e. fix tree) to express the fixing source code of each bug by combining the lexical code tokens, the concept types of code elements and the fixing structures. Then, we use the tree-based encoding method, Tree-based Constitutional Neural Network (TBCNN) (Mou et al., 2016), to represent the fix trees for the bug fixes of these historical fixed bugs.
(3) We propose an approach based on the representation of fix trees that can automatically classify the bug causes of the historical fixed bugs.
(4) We evaluate our approach on a dataset consisting of 2000 bugs from two open source projects, and the results show that the fix tree is effective to classify historical fixed bugs into corresponding bug causes.

[4] https://bugzilla.mozilla.org/.
[5] https://github.com/radare/radare2.

Table 1
Bug categories related to the cause of bugs.

Cause category	Sub-category	Description and corresponding fix	Abbreviation
01 Function	0101 Functional error	The function cannot be implemented normally. The general scope of fixing is in the overall structure, such as the main function or the fixing spans the code file.	FuncErr
	0102 Algorithm error	Errors in the set of steps used to solve a particular problem or calculation, including errors in calculations, error implementations of algorithms. Fixing usually appears in the function/method definition, which contains some statements with logical structure.	AlgrErr
02 Interface	0201 Interface error between modules	Interaction issues between components or with other systems. Fixing usually appears in include/import statements and function/method call statements.	IntrfBtwErr
	0202 Interface error within the module	Interaction issues within components, including mismatched calls and incorrect opening, reading, writing, or closing of files and databases. Fixing usually appears in function/method call statements, or parameter lists.	IntrfInErr
03 Logic	0301 Incorrect Branch	The branch statement is incorrect. Fixing usually appears in the branch block of if-else, switch statement.	BrchErr
	0302 Ignore extreme conditions	Extreme situations are not considered. Lack of special case handling or boundary values, null checks. Fixing usually appears in the addition of the if statement, and the if conditional expression checks for extreme cases.	IgnrCond
	0303 Redundant logic	Extra logical statements. Some logic statements will be deleted in the fixing.	RedtLogc
	0304 Conditional test error	The logic test conditions are incorrect. Fixing usually appears in conditional expressions that fix logical statements,such as if, while statement.	CondErr
	0305 Incorrect loop	The loop logic is incorrect. Fixing usually appears in the loop body of for, while, do-while and other loop statements.	LoopErr
	0306 Logical order error	The logical order is wrong, including jump error. Fixing usually appears in the changing the order of the statements or jump statement.	LogcErr
04 Computation	0401 Operator error	The wrong operator was used. Fixing usually appears in modification of operators.	OprtErr
	0402 Incorrect operand	The operand in the operational expression is incorrect. Operands are generally variables or literals.Fixing usually appears in modification of operands.	OprnErr
	0403 Insufficient precision	The precision of the data is not sufficient.Fixing usually appears in an expression, where the numeric constant value is not set correctly or the placeholder precision setting is incorrect.	PrcsnErr
05 Assignment	0501 Initialization error	Data initialization error. Fixing usually appears in the declaration of a variable, with an assignment expression.	InitErr
	0502 Access error	The access to the field is incorrect. Fixing usually appears in variable access statements or modification of the data address.	AccErr
	0503 Inconsistent subroutine parameters	Such as the type of argument used in the method call is incorrect. Fixing typically appear in the list of arguments for method calls.	IncnParam
	0504 Incorrect data range or type	The limited data range or type is incorrect. Fixing usually appears in modification of qualifiers or type constants.	TypRngErr
	0505 Input or Output data error	The input or output data is incorrect. Fixing usually appears in expressions that contain an input or output stream call.	InOutErr
	0506 Data verification error	Inspection issues of abnormal data. Fixing usually appears in the call of exception handling, such as try-catch, throw statements.	VerfErr
	0507 Incorrect variable name or constant name	The reason for adding this item is that the modification of the variable or constant name may be missed during the porting process. Fixing usually appears in modification of names.	NamErr
06 Others	0601 Others	All the bugs not in the above categories	Other


Bugzilla Search Bugs Browse Advanced Search New Account Log In Forgot Password

Bug 1202989
g2p segfaults on creating phonemes for some bad inputs
RESOLVED FIXED in Firefox 43

Get help with this page

Status
Product: Core
Component: Web Speech
Status: RESOLVED FIXED
Reported: 3 years ago
Modified: 3 years ago

People
Assignee: kdavis
Reporter: kdavis
Triage Owner: André Natal
CC: 1 person

Tracking
Version: unspecified
Target: mozilla43
Points: ---
Blocks: ~~Meta-Vaani~~

Firefox Tracking Flags
Tracking Flags: Tracking Status
firefox43 --- fixed

Details
Whiteboard: [webspeechapi][vaani][systemsfe]
Votes: 0 votes

Attachments
Added check for 0 length phones string in addition to NULL
3 years ago kdavis
1.12 KB, patch
anatal : **review+** Details | Diff | Splinter Review

Bottom ↓ Tags View

Fig. 1. The bug report #1202989 in Mozilla.

The rest of this paper is organized as follows: Section 2 introduces the background. Section 3 shows the overall methodology. In Section 4, we present the empirical study and results of the study, while Section 5 discusses the possible threats to validity. Section 6 discusses the related work. Finally, Section 7 concludes the paper and outlines our future research agenda.

2. Background

2.1. Bug fix data

To improve the quality of software products, developers, testers, and users are often allowed to report bugs they encounter in the bug tracking system (BTS), such as Bugzilla[6], Mantis[7], BugNet[8], Jira[9], etc. In addition to the description information for each bug, the fixing information from the assignee (the person assigned to fix the bug) is also attached to the committed message in the corresponding bug report when the bug was fixed (Sun et al., 2017a; 2018), as shown in Fig. 1. These commits usually contain the modified source code, which is included in the "diff" hyper link in Fig. 1. Fig. 2 is a screen shot of diff source code of a bug (BUGID:1202989[10]) in Bugzilla. The well-known software project hosting platform Github[11] also has its issue tracking module. The issues whose status are closed and tagged with "bug" are historical fixed bugs. The bug reports of Mozilla we used come from the fixed and verified bugs hosted on Bugzilla. We get the bug reports of Radare2 from the closed bugs hosted on Github. In order to construct fix trees of the corresponding bugs, we get all the complete source code before and after modification from Github.

As mentioned before, we find that the knowledge in the bug fix has a corresponding relationship with the cause classification of bugs at AST level. So we analyze fixed code at AST level in this paper. In general, AST refers to the tree-like abstract syntax structure of the source code written in a programming language. Fig. 3 illustrates the AST of the statement "$if(a > b)\{a = a - b;\}$". Each node of the tree represents its code element type in this code.

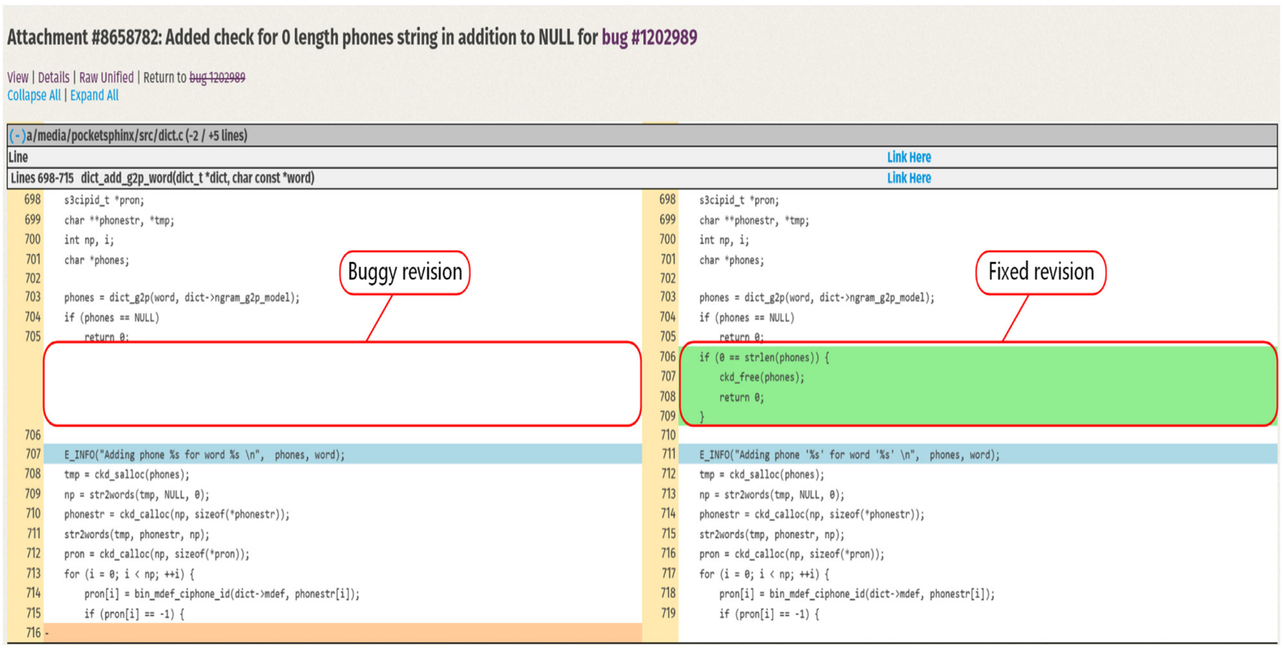
In this paper, we leverage GumTreeDiff (Falleri et al., 2014) to extract fix related concept types of code elements and code structures to construct the fix tree of each bug fix which implies the knowledge in the bug fix. The tool GumTreeDiff[12] is open source, allowing for replication, and it can parse AST for fixed code in various programming languages.

2.2. Bug classification

Classifying software bugs can improve bug inspection and guide software bug localization and fix.

Among the bug classification methods, the Orthogonal Defect Classification (ODC) method is widely used (Chillarege et al., 1992), which is introduced by IBM. The ODC method classifies defects based on the semantics of associated defects, and combines the distribution of defects with the software development process and the maturity of the software. The ODC method is not only a description of the defect type, but also provides an analysis of the

[6] https://www.bugzilla.org/.
[7] https://www.mantisbt.org/.
[8] http://bugnet.switchking.se/.
[9] https://www.atlassian.com/software/jira.
[10] https://bugzilla.mozilla.org/attachment.cgi?id=8658782&action=diff.
[11] https://github.com/.
[12] https://github.com/GumTreeDiff.



Attachment #8658782: Added check for 0 length phones string in addition to NULL for bug #1202989

View | Details | Raw Unified | Return to ~~bug 1202989~~
Collapse All | Expand All

(-)a/media/pocketsphinx/src/dict.c (-2 / +5 lines)

Line — Link Here

Lines 698-715 dict_add_g2p_word(dict_t *dict, char const *word) — Link Here

```
s3cipid_t *pron;
char **phonestr, *tmp;
int np, i;
char *phones;

phones = dict_g2p(word, dict->ngram_g2p_model);
if (phones == NULL)
    return 0;

E_INFO("Adding phone %s for word %s \n",  phones, word);
tmp = ckd_salloc(phones);
np = str2words(tmp, NULL, 0);
phonestr = ckd_calloc(np, sizeof(*phonestr));
str2words(tmp, phonestr, np);
pron = ckd_calloc(np, sizeof(*pron));
for (i = 0; i < np; ++i) {
    pron[i] = bin_mdef_ciphone_id(dict->mdef, phonestr[i]);
    if (pron[i] == -1) {
-
```

```
s3cipid_t *pron;
char **phonestr, *tmp;
int np, i;
char *phones;

phones = dict_g2p(word, dict->ngram_g2p_model);
if (phones == NULL)
    return 0;
if (0 == strlen(phones)) {
    ckd_free(phones);
    return 0;
}

E_INFO("Adding phone '%s' for word '%s' \n",  phones, word);
tmp = ckd_salloc(phones);
np = str2words(tmp, NULL, 0);
phonestr = ckd_calloc(np, sizeof(*phonestr));
str2words(tmp, phonestr, np);
pron = ckd_calloc(np, sizeof(*pron));
for (i = 0; i < np; ++i) {
    pron[i] = bin_mdef_ciphone_id(dict->mdef, phonestr[i]);
    if (pron[i] == -1) {
```

Fig. 2. The diff source code of #1202989 in Mozilla.

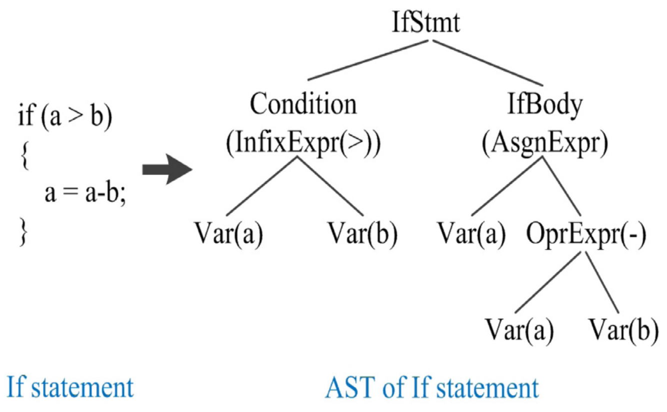


Fig. 3. The *if* statement and its AST.

cause of the defect. The ODC analyzes the defects in detail, and is suitable for the localization, elimination, analysis of defect causes and defect prevention. In the ODC, the defect contains eight orthogonal defect attribute types: Activity, Triggers, Impact, Target, Type, Qualifier, Source, and Age. Since the bug fix of the historical bugs discussed in this paper is at the software source code level, and the software bug occurrence here is also limited to the source code, this paper mainly focuses on the attributes related to source code and their values in the ODC method. A key point of this paper is to exploit the relation between the bug fix and the cause of bugs. The cause classification criterion presented in this paper is mainly based the ODC method. So we first select the category based on its defect type attribute related to source code. The following describes the attribute values of the code-related defect types in the ODC.

- Assignment: incorrect assignment, or no assignment.
- Checking: missing or incorrect data, or incorrect loop and conditional statements.
- Interface: An interaction that occurs between software components, modules, drivers, or errors in call statements, parameter lists.
- Algorithm: Errors that can be fixed by modifying algorithms or data structures without major design-level changes.
- Function: Involving the execution error of the software function, the code modification amount is large.

From the perspective of code analysis, the defect types provided by ODC are still too coarse. We further combine the IEEE classification method (Ieee, 1994) to refine the bug category in more detail. The IEEE standard classification for anomalies (Ieee, 1994) provides a comprehensive classification of software anomalies. This standard describes the processing of software anomalies discovered at various stages of the software lifecycle. The classification process is divided into four sequential steps interspersed with 3 administrative activities. The four sequential steps are: recognition, investigation, action and disposition. The classification code consists of 2 letters and 3 numbers. A decimal can be added to represent a new classification. For example, RR324, IV321.1, where RR is the recognition step and IV is the investigation step. The investigation step divides the anomalies into logic problem, computation problem, interface/timing problem, data handling problem, data problem, documentation problem, document quality problem and enhancement problem. There are eight categories, and they are divided into different numbers of sub-categories. The classification is in-depth and it accurately indicates the types of anomalies. This classification method provides a unified method for detailed classification of anomalies found in software and documents, and provides data items related to anomalies to help identify anomalies and track abnormalities. The IEEE standard classification for anomalies has high authority and can be tailored for actual software projects with high flexibility and wide application. Since the investigation step covers the classification of the actual cause of the anomalies, in this paper we extracted the parts (i.e. IV300/390/400 in the IEEE standard classification for anomalies) related to the software source code as the sub-types of the defect types of ODC. The bug categories related to the cause of bugs used in this paper are illustrated in Table 1.

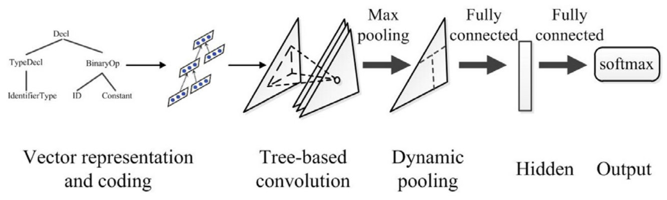


Fig. 4. Framework of tree-based convolutional neural network (TBCNN).

2.3. Tree-based convolutional neural network

In recent years, deep learning is applied to programming language processing (Lam et al., 2017; Mou et al., 2016; Zaremba and Sutskever, 2014; Piech et al., 2015; White et al., 2015; Huo et al., 2016; Huo and Li, 2017). Similarly, source code contain rich structural information. Mou et al. (2016) proposed a tree-based convolutional neural network (TBCNN) for programming language processing, in which the convolution kernel is designed on the AST of the program to capture structural features. In addition, TBCNN is a general architecture for programming language processing. In this paper, in order to characterize the knowledge in the bug fix, we employed the TBCNN model to represent the fix trees of the bug fix and extract the features of the fix trees.

The tree-based convolutional neural network (TBCNN) is a CNN-based model on processing source code, which is suitable for program classification problems. The architecture of TBCNN is shown in Fig. 4. The model consists of five main layers: vector representation and coding, tree-based convolution, dynamic pooling, a fully-connected hidden layer and an output layer. First, the model takes as input the entire AST of a program, program vector representations are learned from the AST in the coding layer. The aim of this layer is to embed AST symbols in a continuous vector space, since similar symbols have similar feature vectors. For example, the symbols 'While' and 'For' are similar because both of them are loop statements. But they are different from 'ID' which probably represents some data. After coding, each symbol in ASTs is represented as a distributed, real-valued vector. Then, a set of fixed-depth subtree feature detectors, called the tree-based convolution kernel, is designed to sliding over the entire AST to extract structural information of a program. Next, a dynamic pooling layer is applied to gather information over different parts of the tree. Further, a three-way pooling where features are pooled to three parts according to their positions in the AST. After pooling, the features are fully connected to a hidden layer and finally fed to the standard output layer (softmax) for supervised classification.We also use the standard output layer as one of the classifiers for supervised learning.

3. Approach

The general framework of our approach is illustrated in Fig. 5. First, we need to preprocess the commits of historical bugs to extract the bug fixes. Then, we construct the fix trees for these extracted bug fixes and represent the fix trees based on the encoding method of TBCNN. Finally, we use the classifier to automatically classify the bug fixes into the corresponding bug cause category.

3.1. Construction

In this paper, we propose to represent the bug fix of each bug in the form of a fix tree by combining the lexical code tokens, the concept types of code elements and the fixing structures in the source code. We propose to use a tree structure for each bug fix to preserve the natural syntactic structure of the source code (i.e. Abstract Syntax Tree), and to intuitively represent the fix action to each node of the tree. A fix tree includes fix-related code element terms (also called code tokens), concept types of code element terms, and fix actions and their relations.

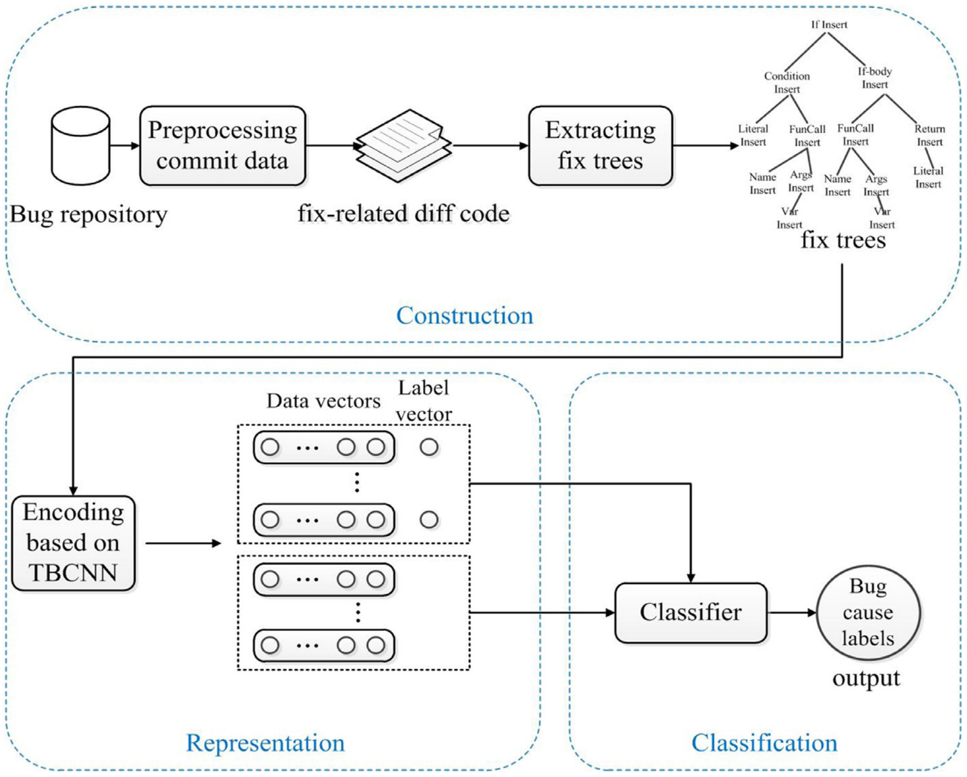


Fig. 5. Framework of the approach.

The terms of the source element consist of identifiers (such as variable name, method name, class name), literals, and operators. Code element terms imply lexical semantics (i.e.lexical features) of the bug fix. The concept type of a code element refers to the functionality of a code element in the source code, such as type declarations, function or method calls, loop statements, assignment expressions, etc. For example, the concept type of 'ckd_free' in Fig. 2 is "Function Call (Abbreviation: FuncCall)". A fix action refers to the change operation on a code element, such as changing the condition of the 'if' statement. The relations between fix actions are the relative positions on the fix tree. For example, the action of changing the condition of the 'if' statement is the child of the action of changing this 'if' statement. Similarly, the fix actions and their relations contain the features of the fixing pattern and structure semantics, respectively.

The code element terms are the lexical tokens related to the fix in the code, where we remove the keywords in the corresponding programming language and the general symbols except the computation operators.

Let's take the diff source code in Fig. 2 as an example, which shows the syntactic difference between the buggy version and the fixed version of a code segment. The differencing tool (Falleri et al., 2014) can help extract the changes generated by the fix.

We define the fix tree based on the AST structure. Each node of a fix tree corresponds to the AST node of the modified file. An edge of a fix tree denotes the corresponding parent-child relationship between the AST nodes of the modified file. The position order of the nodes in the same layer of a fix tree is the same as that of the corresponding AST tree. The schematic fix tree of the bug in Fig. 2 is shown in Fig. 6, which represents the addition of an 'if' statement containing a jump. It is necessary to specify that while extracting the fix action, we also extract the corresponding code element terms and their concept types, which we have omitted in Fig. 6 to simplify the structure of the fix tree. As can be seen from Fig. 6, the nodes on the tree are micro fix patterns, such as: *f1 = (If insert, *), f2 = (Condtn Insert, fix1), f3 = (If-body insert, fix1), f4 = (Literal insert, f2), f5 = (FunCall insert, f2), f6 = (FunCall insert, f3)*, and *fix7 = (Return insert, f3)*, etc. The formal representation of each node is $fix_i = (et_i, pt_i)$, where et_i is the fixing of current node, pt_i is the fixing of its parent. It can be seen that fix_1 has a parent-child relationship with fix_2 and fix_3, respectively. The same situation occurs in the following nodes. That is, they are nested on the AST structure. Similarly, in the case of a node is deleted, the deleted AST node is reserved in the original location, and the name of the fix action is 'delete'. If a node is moved, the node before the move is reserved in the original location and its name of the fix action is 'move before', the moved node is inserted into the target location and its name of fix action is 'move after'. If a node is modified, the node before the modification is reserved in the original location and its name of fix action is'modify before', The modified node is inserted between the original node and its next sibling AST node, and its name of fix action is'modify after'. Considering the space limitation, the examples of other cases are not illustrated in this paper. But we have posted a complete schematic and its description of an example covering all cases on the website mentioned in Section 3.2. It should be noted that there may be a lot of scattered code blocks in a diff source file that are fixed or the fixing of a bug may involve multiple diff source files. In this case, we use a root node (represented by 'compilable unit root' and 'file root' for two cases respectively) as their common parent.

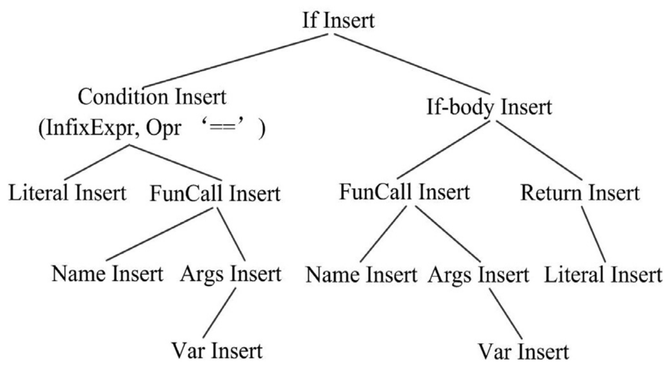


Fig. 6. The fix tree of Bug#1202989.

As can be seen from the "Description and corresponding fixing" column in Table 1, the fix actions and related code elements have a corresponding relationship with the bug cause categories.

With the definition of the fix tree, we need to construct fix trees for the fixing source code of real-world bugs. We built a tool to construct fix trees by applying the tool, GumtreeDiff[13], which deduce diff pairs of source code at AST level (Falleri et al., 2014). GumtreeDiff can process code files written in various programming languages, including C and Java files used in this paper. We extract fix trees at AST level. That is, each node on the extracted tree represents a fix at the corresponding location. Each fix contains all the specific knowledge in the bug fix, such as code element terms, concept types of code element terms and fix actions. The relations between fix actions have been implied in the structure of the fix tree.

In addition, we refine the fix trees by removing irrelevant code element terms. In this process, we leverage regular expressions to remove useless code tokens to prune each tree. In detail, we remove the keywords of the corresponding programming language during the process of extracting code element terms, such as 'void', 'if', 'while'. Because these keywords represent the type information of code tokens to a certain extent, and the type information is already included in the extraction of the concept type of every code element. At the same time, the general symbols except the operators for computation are also removed, such as '{' '}' ';' '//' '*'. In order not to affect the construction of the tree structure, we remove the comments in the bug fix, which are described in natural language and their syntax structure is different from the source code.

3.2. Representation

To automatically identify the cause type of a bug, we have to extract bug features related to the cause type. According to our previous manual observation, the knowledge in the fixing source code has a correspondence with the cause of the bug. Therefore, after constructing fix trees at AST level, we need to represent these fix trees to express their features. We represent the fix tree of each bug fix by applying the coding method of TBCNN (Mou et al., 2016), which is an unsupervised approach to learn program vector representations by a coding criterion. This method is based on vector representations (also known as embeddings), which can capture the underlying semantic meanings of symbols, such as AST nodes (Bengio et al., 2013). For example, the symbols 'While' and 'For' are similar because they have the same concept type, i.e., loop statement related to control flow.

During the coding process, the children nodes are used to "code" their parent via a single neural layer, during which both vector representations and coding weights are learned. In the scenario based on tree structure, leaf nodes are just the direct vector representations. For a non-leaf node p, its representation is linearly combined with its vectors and the coding from its children.

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

Formally, let $c_1, c_2, \ldots, c_n$ be the children of node p and the combined vector of p is denoted as follows:

$$\mathbf{p} = W_{comb1} \cdot vec(p) + W_{comb2} \cdot tanh\left(\sum l_i W_{code,i} \cdot vec(c_i) + b_{code}\right) \tag{1}$$

where $W_{comb1}, W_{comb2} \in \mathbb{R}^{N_f \times N_f}$ are the parameters for combination. The $vec(p)$ and $vec(c_i)$ are computed in Eq. (2). The vector of each node on a fix tree is a linear combination of the direct vectors of the related code element term, concept type of this element and the fix action. Formally, the vector of a node f is denoted as follows:

$$vec(f) = W_{f_e} \cdot vec(f_e) + W_{f_t} \cdot vec(f_t) + W_{f_a} \cdot vec(f_a) \tag{2}$$

where $W_{f_e}, W_{f_t}, W_{f_a} \in \mathbb{R}^{N_f \times N_f}$ are the weights for combination. All the parameters are initialized as diagonal metrics and then fine-tuned during supervised training. Therefore, the $vec(f_e)$ of a keyword node is a zero vector, its $vec(f_t)$ and $vec(f_a)$ are the vector representations of its corresponding type name and the name of its fix action. The $vec(f_e)$, $vec(f_t)$ and $vec(f_a)$ of a custom method node are the vector representations of its corresponding method name, type name and the name of its fix action.

After coding, the fixing nodes are embedded in a continuous vector space where similar nodes are mapped to closer points. Each fixing node in the fix tree is represented as a distributed, real-valued vector $x \in \mathbb{R}^{N^f}$. Then, we employ the tree-based convolutional layer of TBCNN to extract the features of the fix trees. Specifically, we apply the fixed-depth feature detectors in TBCNN to slide over each tree to detect the features represented in the coding method. Formally, if there are n nodes with vector representations, the output of the feature detectors is as follows:

$$y = tanh\left(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv}\right) \tag{3}$$

where $y, b_{conv} \in \mathbb{R}^{N^c}$ represent the feature vectors and bias respectively, $W_{conv,i} \in \mathbb{R}^{N_c \times N_f}$ is the weight matrix corresponding to the vector x_i. N^c is the number of feature detectors. N^f is the feature dimension.

A standard dynamic pooling layer using one-way pooling is then applied to aggregate all extracted features from the feature detectors. Therefore, the output of the representation method are these aggregated features, which can be used in the following classification model. The source code for constructing and representing the fix trees is available online.[14]

3.3. Classification

With the fix tree representation obtained from the above step, we can use the classification model for supervised learning to classify bugs into their bug cause category. The focus of this paper is to exploit the bug cause with the fix trees, and we do not limit our model to one fixed classification technique. In our experiment, we used three classification techniques for bug cause classification, i.e., standard output layer (i.e. softmax) of CNN (Kalchbrenner et al., 2014), Bayesian learning (McCallum et al., 1998; Zolnierek and Rubacha, 2005) and support vector machine (SVM) (Pellin, 2000; Elish and Elish, 2008). For the CNN model, the settings in the representation step is kept. The network is trained by forward propagation of training data and then backward propagation of its errors. A dropout method (Srivastava et al., 2014) was applied to prevent overfitting of the training data. Finally, softmax activation function was used at the output layer. After the representation step, each fix tree is represented as a 600-dimensional vector. These 600 dimensions are regarded as 600 attributes of each sample. In addition to the category label, each data is represented as a 600-dimensional vector. Therefore, the 601-dimensional vectors are used as the input of the Bayesian learning model and the SVM model, where the last dimension is the category attribute.

Take the source code fix in Fig. 2 as an example. As the vectors of the code element terms (‘phones’, ‘ckd_free’, ‘0’, ‘==’, etc.), the code element types (‘IfStmt’, ‘IfCond’, ‘InfixExpr’, etc.), and the fix actions (‘IfInsert’, ‘IfCondInsert’, ‘ifBodyInsert’, etc.) are mapped into closer points, the features of them are finally aggregated together. Then, with the manual label, the aggregated features are trained as the identification basis for the cause type of “Ignore extreme conditions” (Abbreviation: IgnrCond). Consequently, in the testing phase, the fix trees with similar nodes which contain similar features are classified into the same category.

4. Experiment

4.1. Research questions

To evaluate the effectiveness of our approach, we are concerned about the following research questions.

RQ1: What is the performance of our classification approach?

In this paper, we aim to exploit the corresponding relationship between the bug fix and the bug cause. We first did a preliminary statistical analysis to verify the existence of the correlation between bug causes and the AST-level modifications in the bug fixes. The preliminary analysis is detailed in Section 4.3. Then we trained different classification models using fix trees and bug cause labels to classify bugs from the perspective of bug causes. In order to further verify the validity of this correlation, we investigate the performance of our classification approach.

RQ2: How effective does our classification approach based on fix trees perform compared with the classification technique without fix trees?

We use fix trees to represent the knowledge in fixing source code. In order to see whether the method using fix trees is more effective than that not using them, we investigate the effectiveness of our classification approach by comparing the classification models using fix trees with those not using fix trees.

RQ3: How does our approach perform across different projects or products?

In the process of performing supervised classification learning, we manually labeled the categories for the samples in training set. But it is impracticable to do manual labeling for every project. So we have to see whether our approach of bug classification can achieve an acceptable accuracy across different projects and products.

4.2. Data collection and manual labeling

In our study, we used the data sets using two programming languages, C and Java, from the Mozilla Project and the Radare2 Project. To get the diff source code of software bugs, we selected the diff source code in C from the Mozilla Project and the Radare2 Project, and those in Java from Android Firefox and Rhino, which are both products in Mozilla. All the products are popularly used and have a large number of fixed/closed bugs. Mozilla is an open source web development project that develops code for use in the Mozilla Application Suite. The Mozilla Application Suite is a complete set of web applications, such as browser, chat client, news client, mail client, etc. Mozilla’s bug repository is hosted on Bugzilla. Radare project started as a forensics tool and a scriptable commandline hexadecimal editor able to open disk files, but later support for analyzing binaries, disassembling code, debugging programs, attaching to remote gdb servers, etc. Radare2 is a rewrite

[14] https://github.com/NizhenJenny/FixTree.

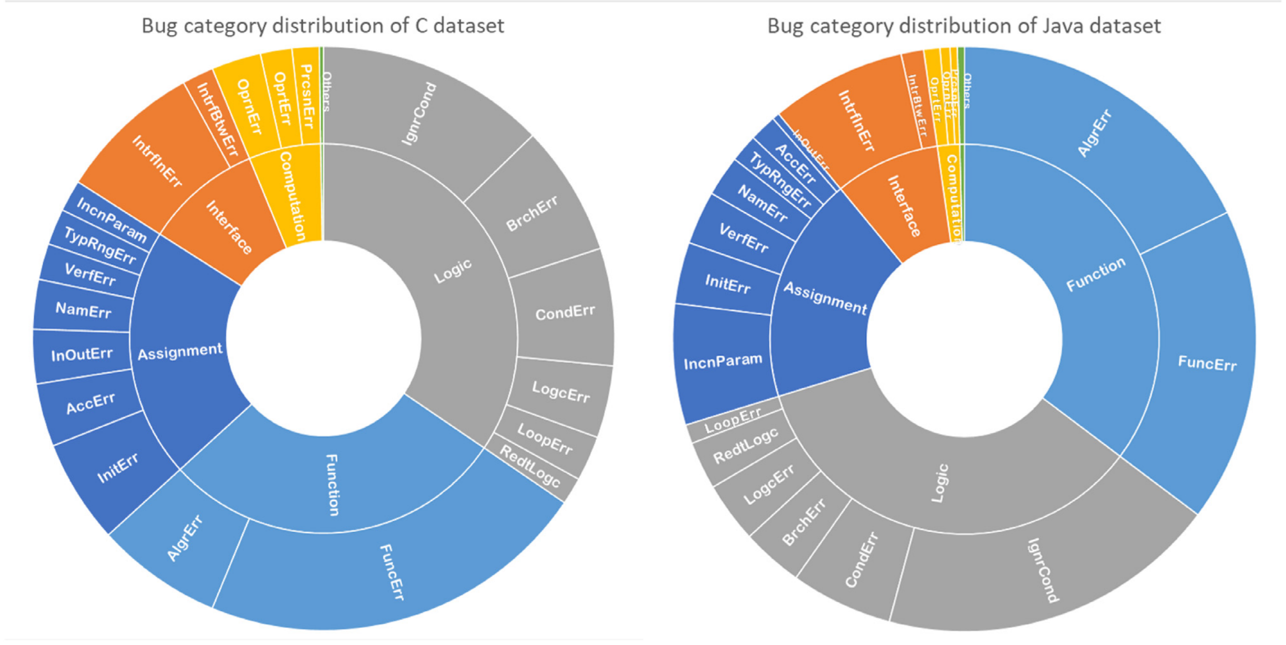


Fig. 7. The distribution of bug cause categories in two datasets.

from scratch of Radare in order to provide a set of libraries and tools to work with binary files. Bug tracking of Radare2 is provided by GitHub. Firefox for Android (codenamed Fennec) is an open, hackable, standards-based browser, just like the desktop Firefox. It constructs its user interface from native Android widgets. Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users. It is embedded in J2SE 6 as the default Java scripting engine.

In order to ensure the usability and validity of the data source, we only select the bugs with the status of *fixed* and *verified* on Bugzilla. Particularly, The historical fixed bugs of Radare2 are extracted from the issues tagged by *bug* with the status of *closed* on GitHub. In addition, we filtered the bugs whose bug fixes do not use the relevant programming language. In order to verify the validity of the ground truth (i.e. the issues are real bugs rather than functional improvements, and the source code files actually modified to fix the bug described in the bug report), the dataset was manually analyzed. During this verification, two of the authors manually verified the bug reports and their fixes, each of them focusing on half of the bugs. In order to prevent each fix tree from being too large or having redundant branches, source code files that are correspondingly modified because of the modification of the buggy files are excluded during this process. Then a third author double-checked the manual analysis results done by the other two authors. This step is necessary to identify any misunderstandings, as well as to conduct a discussion when there is a inconsistency. It is also important to note that all the complete source code files (including the ones before and after modification) of the corresponding bugs are obtained from Github. We randomly selected 400 diff files written in C programming language and 1600 diff files written in Java programing language from Bugzilla and Github.

During the process of manual labeling, four of the authors investigate the bug cause categories. Before manual labeling, all participants are trained to understand the definition of each cause type and the classification method. These participants are familiar with the category definition, and they all have some experiences on using Bugzilla and Github. In the first round, each participant labeled a quarter of the dataset which had been randomly divided into four equal parts. After investigating the bug report and commit messages of each bug, they determined its cause type. In the second round, four participants exchanged their respective quarter for the second time labeling. That is, each bug was investigated twice by two participants independently. We introduced Cohens Kappa Cohen (1960) for the labeling results to measure the agreement between the participants. Cohens Kappa is widely used as a rating of inter-rater reliability (Le et al., 2019; Schutze et al., 2008), and it represents the extent to which the data collected in the study are correct representations of the variables being measured. The value of the Cohens Kappa between two raters in every round is computed in Eq. (4).

$$\kappa = \frac{P_O - P_E}{1 - P_E} \tag{4}$$

where P_O is the sum of the number of bugs with consistent classification results in each category divided by the total number of bugs, that is, the accuracy of the overall classification consistency. Suppose the number of bugs classified by labeler A into each category is $a_1, a_2, \ldots a_n$, the number of bugs classified by labeler B into each category is $b_1, b_2, \ldots b_n$ (n is the number of categories, in this case it is 20), then P_E is calculated according to the following equation: $P_E = \frac{\sum_{i=1}^{n} a_i \times b_i}{n \times n}$. The average value of Cohens Kappa after the second round is 0.88. According to the interpretation of kappa value by Schutze et al. (2008), a kappa value falling between 0.81 and 1 demonstrates an almost perfect agreement between raters the first highest level of agreement by their interpretation. Finally, all the participants discussed and reached a consensus for the inconsistent results. In particular, during the process of manual labeling, the participants did not investigate the bug fixes of these bugs. The distribution of the cause categories of these 2000 bugs is illustrated in Fig. 7.

4.3. Preliminary analysis

In order to verify the existence of the observed correlation between bug causes and the AST-level modifications in the bug fixes, we did a preliminary statistical analysis. From the dataset in

Table 2
The cause categories and the corresponding modification types.

Cause category	Modification type
A1 Functional error	B1 The definition of one or more classes/functions in one or more files has been changed, and the amount of modification is large.
A2 Algorithm error	B2 In the definition of a method/function, multiple expression statements in a block are modified, including more than 3 expression statements.
A3 Interface error	B3 One or more interface/function call statements are changed, sometimes with the import statement being changed.
A4 Incorrect Branch	B4 A logical statement such as an ifelse statement or a branch in a switch statement is modified.
A5 Ignore extreme conditions	B5 Conditional expressions are added, or if conditional statements are added.
A6 Redundant logic	B6 Extra logical statements are deleted.
A7 Conditional test error	B7 A conditional expression is modified.
A8 Logical order error	B8 The position of an expression statement is transformed.
A9 Initialization error	B9 The definition of a variable is modified.
A10 Inconsistent subroutine parameters	B10 The arguments in a method call are modified.
A11 Data verification error	B11 The exception handling statements are modified.
A12 Others	B12 Others

Table 3
The Pearson Correlation Coefficient value between cause type A_i and modification type B_j.

$\gamma_{A_iB_j}$	B_1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
A1	**0.89**	−0.32	−0.14	−0.09	−0.26	−0.08	−0.11	−0.09	−0.10	−0.13	−0.08
A2	−0.33	**0.86**	−0.14	−0.08	−0.23	−0.05	−0.10	−0.08	−0.07	−0.12	−0.07
A3	−0.15	−0.16	**0.81**	−0.05	−0.12	−0.04	−0.06	−0.04	−0.04	−0.06	−0.01
A4	−0.11	−0.08	−0.05	**0.88**	−0.07	−0.02	−0.03	−0.03	−0.03	−0.04	−0.02
A5	−0.26	−0.23	−0.10	−0.07	**0.93**	−0.04	−0.07	−0.06	−0.07	−0.09	−0.05
A6	−0.08	−0.07	−0.02	−0.03	−0.05	**0.85**	−0.02	−0.02	0.01	−0.04	−0.02
A7	−0.11	−0.11	−0.03	−0.03	−0.07	−0.02	**0.97**	−0.02	−0.03	−0.03	−0.02
A8	−0.09	−0.05	−0.05	−0.03	−0.06	−0.02	−0.02	**0.92**	0.00	−0.02	−0.01
A9	−0.09	−0.08	−0.05	−0.03	−0.05	−0.03	−0.03	−0.02	**0.84**	−0.03	−0.02
A10	−0.13	−0.13	−0.04	−0.04	−0.08	−0.04	−0.04	−0.02	0.00	**0.95**	−0.03
A11	−0.08	−0.04	−0.04	−0.02	−0.05	−0.02	−0.02	−0.01	−0.02	−0.03	**0.90**

Section 4.2, we selected the cause categories in which the number of bugs is more than 30 per category. For categories with more than 200 bugs, we randomly sample 200 bugs to keep in this dataset. The threshold (i.e. 30) is set to prevent the sparseness problem of data distribution in the subsequent grouping. The names of these cause categories are shown in the first column of Table 2. The total number of the bugs in this dataset is 1089. The choice of modification categories is mainly based on the fault taxonomies in the literature (Hayes et al., 2005) and the change types in the literature (Zhao et al., 2017a), the categories of modifications in the bug fixes are shown in the second column of Table 2. Therefore, we need to verify the correlation between A_i and B_i ($i = 1, 2, 3, \dots, 11$).

We did a manual labeling for this dataset using the modification categories. Before manual labeling, all participants are trained to understand the definition of each modification type and the classification method. Similar to the last manual labeling, four of the authors investigate the modification categories. In the first round, each participant labeled a quarter of the dataset which had been randomly divided into four equal parts. The bug fix of each bug was analyzed in order to determine its modification type. In the second round, four participants exchanged their respective quarter for the second time labeling. The average value of Cohens Kappa for the labeling results after the second round is 0.95. Finally, all the participants discussed and reached a consensus for the inconsistent results. In particular, during this process of manual labeling, the participants did not investigate the bug report or commit messages of the bugs, and the labels of bug cause categories are not known to them.

We use the Pearson Correlation Coefficient (PCC) (Benesty et al., 2009) to measure the degree of the correlation between A_i and B_j ($i, j = 1, 2, 3, \dots, 11$).The Pearson Correlation Coefficient is a statistical indicator used to reflect the close relationship between the two variables, which is also suitable for software data analysis (Lientz and Swanson, 1981). The Pearson correlation coefficient varies between -1 and $+1$ with 0 implying no correlation. Correlations of -1 or $+1$ imply an exact Linear correlation. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases. We performed 1000 rounds of calculations. In each round, we randomly shuffled the bugs and divide them into 10 groups. The value of the Pearson Correlation Coefficient between A_i and B_j in every round is computed in Eq. (5).

$$\gamma_{A_iB_j} = \frac{\sum_{a=1}^{N}\left(A_{ia} - \overline{A_i}\right)\left(B_{ja} - \overline{B_j}\right)}{\sqrt{\sum_{a=1}^{N}\left(A_{ia} - \overline{A_i}\right)^2 \sum_{a=1}^{N}\left(B_{ja} - \overline{B_j}\right)^2}} \tag{5}$$

where $N = 10$ is the number of groups, $i, j = 1, 2, 3, \dots, 11$, A_{ia} is the number of the bugs whose cause type is A_i in group a, B_{ja} is the number of the bugs whose cause type is B_j in group a, $\bar{A_i}$ is the average number of the bugs in cause category A_i, $\bar{B_j}$ is the average number of bugs in cause category B_j.

The final value of the Pearson Correlation Coefficient between A_i and B_j is the average of 1000 rounds of calculations for each $\gamma_{A_iB_j}$. The results are shown in Table 3, where the element with the position of ith row and jth column represents the PCC value of A_i and B_j. As can be seen from Table 3, the elements on the diagonal are positive numbers greater than 0.8, and the elements at other locations are all numbers having an absolute value less than 0.4. Therefore, there is indeed a corresponding relationship between cause type A_i and modification type B_i. With the correlation between bug causes and the AST-level modifications in the bug fixes, we leveraged the bug fixes to build the classification model to automatically classify bugs into cause categories.

Table 4
Hyperparameters of the representation step.

Hyperparameter	Value
Initial learning rate	0.1
Learning rate decay	None
Vector dimension of embedding	30
Dimension of convolutional layer	600
Dimension of penultimate layer	600

4.4. Experimental setup

In this paper, we applied the method based on fix trees of the bugs to classify diff source code in our dataset. The target label of a data sample is one of 20 cause categories (except for the *Others* category) of bugs as we presented before. That means, the bugs whose diff code have a same target label belong to the same cause category. For convenience, each category is represented as an ID illustrated in Table 1.

In the representation step, we extracted vector representations of the fix trees based on the training approach of TBCNN. That is, the vector representations are generated when the best model are trained. These vector representations are used as input to the classifier in the next step. After division of the samples, the proportion of the training set, validation set, and test set is set to 3:1:1. Samples of each category are randomly drawn to each set according to this proportion. The validation set is used for model evaluation and selection, that is, model selection and tuning parameters is based on the performance of the model on the validation set. In our experiment, the iteration (epoch) is 50 and 60 for C and Java respectively according to the best results on the validation set. The hyperparameters in the representation step are illustrated in Table 4.

We used a mini-batch based Stochastic Gradient Descent (mini-batch based SGD) (Dekel et al., 2012) to optimize the parameters and prevent overfitting, because this method is more efficient than the ordinary gradient descent method, and does not fall into the local optimal solution which cannot converge to the ideal state. Mini-batch based SGD iteratively updates model parameters by calculating a batch of samples each time. In the training model phase, mini-batch based SGD optimizes parameters by iteratively updating. In each round of batch processing, a batch of samples is randomly selected, and the error is obtained by the feed-forward operation and the error is calculated. Then, the parameters are updated by the gradient descent method, and the gradient is fed back from layer to layer until the parameters in the first layer of the network are updated. The parameters that need to be optimized by mini-batch based SGD in our method are size of samples per batch, weights for the hidden layer and biases for the output layer. Furthermore, a dropout of 0.5 was applied to prevent overfitting of the training data.

To compare the classification models using fix trees with the those not using fix trees, we need a representation method without fix trees. We used word embedding (Bengio et al., 2003) to represent the diff source code because it shows extraordinary performance in text classification tasks (Maas et al., 2011; Mikolov et al., 2013; Tang et al., 2014; 2015; Yang et al., 2016). For comparative experiments, we used word embeddings of the same code tokens in the fix trees, and the dimension of word vectors is set to the same with the vector representation of fix trees. In the classification step, we used the same three classification models. For the CNN model, we used a straightforward CNN-based approach which takes all textual diff source code together and feeds them directly to a CNN. That is, this CNN-based approach takes all word embeddings of the entire fixing source code at a time. The CNN model was trained using 50 epochs initially, and the model achieving the highest validation performance were used for evaluation on test set. For the Bayesian learning model and the SVM model, we also used the 601-dimensional word vectors (600 dimensions for the data attributes and one dimension for the category label) as input to each classifier. To ensure consistent data distribution across different approaches, we used the same dataset and trained all the models using the same 3/1/1 train/validation/test splitting as the approach with fix trees. We randomly selected the data into each division, and performed this 10 times.

The performance evaluation metrics Hossin and Sulaiman (2015) of the learning model in multi-class classification problems mainly include macro-average of Precision, Recall and F1-Measure (macro-P, macro-R, macro-F1 for short) for evaluating accuracy. So we use the above metrics to evaluate the performance of our classification model in our study. The calculation formulas for these metrics are as follows.

$$macroP = \frac{1}{n}\sum_{1}^{n} P_i \tag{6}$$

$$macroR = \frac{1}{n}\sum_{1}^{n} R_i \tag{7}$$

$$macroF1 = \frac{2 \cdot macroP \cdot macroR}{macroP + macroR} \tag{8}$$

where n is the number of categories. P_i and R_i are the precision value and recall value of the ith category, which are calculated according to the confusion matrix of the ith category Gu et al. (2009).

4.5. Empirical results

RQ1: The effectiveness of our approach

To answer RQ1, we evaluate the effectiveness of our approach in terms of macro-P, macro-R and macro-F1. It should be noted that because the C dataset is small, and the number of samples for the corresponding category in the test set is also small, so the overall performance on the C dataset is significantly lower than that on the Java dataset. The results under the “with fix trees” columns in Table 5 show that the values of macro-P, macro-R, and macro-F1 are all above 70%. That is, no matter which of the three classification models is used, the classifier can automatically classify most of the bugs with an accuracy of more than 70% as long as the representations of the fix trees are used as input. The results confirm our assumption that the bug fix at different hierarchy have a corresponding relationship with the bug cause categories, and the representations of the knowledge in the fix tree can effectively express the characteristics in the bug fix related to the cause of bugs.

RQ2: The effectiveness of fix tree representation

In order to show the effectiveness of fix trees, we compare the classification models using fix trees as input with the same classification models using word embeddings as input. Word embedding is a state-of-the-art feature representation technique in text classification, and it is similar to the feature representation technique in this paper. Both of them convert text into numeric vectors. As shown in Table 5, the models using fix trees outperform the same classifiers with word embeddings. For example, there is an improvement of 13.2%, 15.1%, 14.8% in macro-P, macro-R, macro-F1 over the CNN model not using fix trees for the Java dataset in Table 5. The results indicate that the representation of fix trees can express the role of the code element in source code and the structural features of the diff source code, which are closely related to the cause type of bugs.

RQ3: The effectiveness of our approach for cross projects/products

In order to show whether our approach is also effective in the case of cross-projects or cross-products, we trained the classifier

Table 5
Performance results of different classifiers on the datasets.

Language	Classifier	with fix trees			with word embeddings		
		macro-P	macro-R	macro-F1	macro-P	macro-R	macro-F1
C	CNN	0.756	0.75	0.749	0.538	0.544	0.524
	Bayesian	**0.799**	**0.769**	**0.778**	0.606	0.625	0.603
	SVM	0.766	0.738	0.724	0.562	0.538	0.538
Java	CNN	**0.847**	**0.844**	**0.844**	0.715	0.693	0.696
	Bayesian	0.835	0.822	0.825	0.691	0.68	0.68
	SVM	0.791	0.756	0.759	0.634	0.6	0.612

Table 6
Results of our approach for cross projects/products.

train/test	Classifier	macro-P	macro-R	macro-F1
Mozilla/Radare2	CNN	0.728	0.719	0.71
	Bayesian	0.739	0.738	0.732
	SVM	0.637	0.638	0.635
Radare2/Mozilla	CNN	0.719	0.706	0.704
	Bayesian	0.721	0.706	0.709
	SVM	0.66	0.625	0.629
Android FireFox/Rhino	CNN	0.777	0.778	0.773
	Bayesian	0.769	0.733	0.744
	SVM	0.718	0.711	0.701
Rhino/Android FireFox	CNN	0.798	0.778	0.784
	Bayesian	0.779	0.756	0.754
	SVM	0.724	0.711	0.702

on the Mozilla dataset and tested it on the Radare2 dataset. Similarly, we trained the classifier on the Android Firefox dataset and tested it on the Rhino dataset, and vice versa. Finally, the results of our approach for cross projects/products are shown in Table 6, which show that the gap between each pair of prediction is small. We think that although the two projects or products are for different applications, the code structures of them using the same programming language are similar, which leads to the similarity of the structures of the corresponding fix trees. Therefore, the results show that our approach can be also used to identify bug causes when there is few available labeling data for a project.

5. Threats to validity

We first discuss the threats to external validity, which are about the generalizability of our findings. Our data samples come from two open source projects, Mozilla and Radare2, and two open source products, Android Firefox and Rhino. They are all application software and cannot represent the characteristics of other types of software projects such as those for operating system software. In the future, we plan to reduce this threat by analyzing more software systems in different platforms.

Second, we discuss threats to internal validity, which refer to experiment bias and errors. (1) the tree structure of a bug fix is high-dimensional, and the scale and depth of the fix trees vary greatly among different data samples (diff source code). High-dimensional and large-scale data will increase training time and waste more memory, which in turn affects the performance of the approach. (2) Manual labeling is subjective to a certain extent. Although we have cross-validated, there may still be a small number of incorrect labels. (3) We only sampled 2000 bugs in our experiment, and the small size of data samples will affect the accuracy of the classification model.

Finally, we discuss the threats to construct validity, which refer to the suitability of the evaluation metrics used in our study. In our experiment, the evaluation is based on the metrics of macro-average of Precision, Recall and F1-Measure, and other evaluation measures may yield different results. However, these metrics are widely used to evaluate multi-class classification techniques (Hossin and Sulaiman, 2015; Alreshedy et al., 2018; Lai et al., 2015; Li et al., 2011).

6. Related work

6.1. Bug fix analysis

There are a lot of studies on exploiting patterns of bug fix (Allamanis et al., 2018). Ray et al. (2015) presented a definition of unique changes of source code and provided a method for identifying them in software project history. They presented an empirical study of these unique changes by evaluating their identification technique on the Linux kernel and two large projects at Microsoft. Their method also extracts fix patterns of source code, but only for a few specific change types. Our work covers almost all common fix patterns, and the extraction of fix patterns in our work is aimed at identifying the cause of bugs. By leveraging the features extracted from the fixes of historical bugs, Le et al. (2015) built an oracle to verify the effectiveness of the existing automated program repair techniques. Le et al. (2016) automatically mined bug fix patterns from the history of many projects, and then employ existing mutation operators to generate fix candidates for a given buggy program. The construction of a fix tree proposed in this paper is enlightened by these history driven program repair methods to make use of the bug fixes of historical fixed bugs. However, the difference of this paper from the above two studies lies in that the former builds a representation model of the bug fixes and it is focus on using this model to automatically classify bugs into cause categories while the latter is focusing on automated program repair by using the fix patterns of the historical bugs. To improve existing automated program repair (APR) techniques and build the knowledge on repair actions, Liu et al. (2018) aimed to build the knowledge via a systematic and fine-grained study of 16,450 bug fix commits from seven Java open-source projects, and discussed several insights into tuning automated repair tools. They aim at using fix patterns to adjust the automated program repair (ARP) tools to improve the effectiveness. The aim of our work is to use the knowledge in fixing source code including fix patterns to identify the cause of bugs. Martinez et al. (2013) proposed an approach based on the analysis of AST files within a given commit for searching pattern instances from software history. Delfim et al. (2018) designed and implemented a detector of repair patterns in bug fixes, which performs source code change analysis at AST level. Zhong and Mei (2018) proposed an approach that mines a repair model for exception-related bug. Zhao et al. (2017b) conducted a thorough empirical study to investigate the characteristics of change types in bug fixing source code, and developed an automatic classification tool CTforC to categorize changes. These studies can extract various fix patterns and their extraction methods are also based on the AST structure, but they focus only on the fixes themselves. In this paper, we not only study fixes, but also explore their relation to the causes of bugs.

6.2. Bug classification

Seaman et al. (2008) described an experience report in aggregating a number of historical datasets containing inspection of defect data using different categorization schemes. Among them, the source code inspection defect types are defined based on source code errors. Based on their empirical study, they pointed out that categories that required technical details about the structural characteristics of the software could be deciphered from the code, giving us the motivation to take the code structure into consideration. Ray et al. (2013) investigated the types of porting errors found in practice. By analyzing version histories, they defined five categories of porting errors. Leveraging this categorization, they designed a static control- and data-dependence analysis technique, SPA, to detect and characterize porting inconsistencies. This work only classifies a specific type of bug, namely porting error. In this paper, the scope of our work is on the code-related bug cause category. Thung et al. (2015) propose an active semi-supervised defect prediction approach which is able to learn a good classification model while minimizing the manual labeling effort by actively selecting a small subset of diverse and informative defect examples to label and by making use of both labeled and unlabeled defect examples in the prediction model learning process. Their goal is to minimizing the manual labeling effort in the supervised defect categorization by leveraging active learning. This paper aims at exploiting the corresponding relationship between bug causes and bug fixes to automatically classify bugs into their cause categories.

7. Conclusion and future work

Analyzing the bug fix corresponding to software bugs is an important means for developers to find the cause of bugs. At the same time, the source code of historical fixed bugs contains a wealth of knowledge about bugs. In this paper, we proposed to extract and represent the knowledge in the source code with a fix tree model. Using the fix tree representation as input of different classification models, we can automatically classify the bugs into bug cause categories. The experimental results show that there does exist a relation between the bug fix and the cause of bugs, and the representation model we proposed can improve the accuracy of bug classification. In future work, we attempt to further prune the fix trees to reduce the scale of the training data and improve the efficiency of the model. In this paper, we represent each bug fix as one tree in order to intuitively represent the fix action for each node on the tree. And in the experiment, we only compared our method to the method without using a tree structure, but not to the methods which put the original tree and the modified tree side by side (Le et al., 2016). In future work, we will consider using this method (Le et al., 2016) to study the representation model of bug fixes more comprehensively. In addition, we also plan to study how to use the knowledge in the source code directly to assist in the bug localization and fixing (Wang et al., 2017; Sun et al., 2019b; 2016).

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Zhen Ni: Methodology, Software, Validation, Writing - original draft, Writing - review & editing. **Bin Li:** Conceptualization, Methodology, Supervision, Project administration. **Xiaobing Sun:** Methodology, Supervision, Writing - review & editing. **Ben Tang:** Data curation, Software. **Xinchen Shi:** Data curation, Software.

Acknowledgments

This work is supported partially by Natural Science Foundation of China under Grant No. 61872312, No. 61972335, No. 61472344, No. 61402396 and No. 61611540347, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2018B12, partially by the Jiangsu Qin Lan Project, partially by the Jiangsu “333” Project, partially by the Six Talent Peaks Project in Jiangsu Province under Grant No. RJFW-053, partially by the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project under Grant No. YZU201803, and partially by the Research Innovation Program for College Graduates of Jiangsu Province under Grant No. XKYCX18_035.

References

Allamanis, M., Barr, E.T., Devanbu, P.T., Sutton, C.A., 2018. A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51 (4), 81:1–81:37. doi:10.1145/3212695.

Alreshedy, K., Dharmaretnam, D., German, D.M., Srinivasan, V., Gulliver, T.A., 2018. [engineering paper] scc: Automatic classification of code snippets. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 203–208.

Benesty, J., Chen, J., Huang, Y., Cohen, I., 2009. Pearson correlation coefficient. In: Noise Reduction in Speech Processing. Springer, pp. 1–4.

Bengio, Y., Courville, A.C., Vincent, P., 2013. Representation learning: a review and new perspectives. IEEE Trans. Pattern Anal. Mach. Intell. 35 (8), 1798–1828. doi:10.1109/TPAMI.2013.50.

Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C., 2003. A neural probabilistic language model. J. Mach. Learn. Res. 3 (Feb), 1137–1155.

Bhoopchand, A., Rocktäschel, T., Barr, E. T., Riedel, S., 2016. Learning python code suggestion with a sparse pointer network. arXiv:1611.08307.

Campos, E.C., 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. doi:10.1109/ESEM.2017.55.

Chang, K., Parvez, M.R., Chakraborty, S., Ray, B., 2018. Building language models for text with named entities. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15–20, 2018, Volume 1: Long Papers, pp. 2373–2383.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M., 1992. Orthogonal defect classification - a concept for in-process measurements. IEEE Trans. Softw. Eng. 18 (11), 943–956. doi:10.1109/32.177364.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46.

Dekel, O., Gilad-Bachrach, R., Shamir, O., Xiao, L., 2012. Optimal distributed online prediction using mini-batches. J. Mach. Learn. Res. 13, 165–202.

Delfim, F. M., Durieux, T., Sobreira, V., de Almeida Maia, M., 2018. Towards an automated approach for bug fix pattern detection. arXiv:1807.11286.

Elish, K.O., Elish, M.O., 2008. Predicting defect-prone software modules using support vector machines. J. Syst. Softw. 81 (5), 649–660.

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. doi:10.1145/2642937.2642982.

Gu, Q., Zhu, L., Cai, Z., 2009. Evaluation measures of the classification performance of imbalanced data sets. In: International Symposium on Intelligence Computation and Applications. Springer, pp. 461–471.

Hayes, J.H., Raphael, C.I., Surisetty, V.K., Andrews, A., 2005. Fault links: exploring the relationship between module and fault types. In: European Dependable Computing Conference. Springer, pp. 415–434.

Hossin, M., Sulaiman, M., 2015. A review on evaluation metrics for data classification evaluations. Int. J. Data Min. Knowl. Manage. Process 5 (2), 1.

Huo, X., Li, M., 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19–25, 2017, pp. 1909–1915. doi:10.24963/ijcai.2017/265.

Huo, X., Li, M., Zhou, Z., 2016. Learning unified features from natural and programming languages for locating buggy source code. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9–15 July 2016, pp. 1606–1612.

Ieee, S., 1994. 1044–1993 - Ieee standard classification for software anomalies. IEEE Stand. Indus 9 (2), 1–4.

Kalchbrenner, N., Grefenstette, E., Blunsom, P., 2014. A convolutional neural network for modelling sentences. arXiv:1404.2188.

Kreutzer, P., Dotzler, G., Ring, M., Eskofier, B.M., Philippsen, M., 2016. Automatic clustering of code changes. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14–22, 2016, pp. 61–72. doi:10.1145/2901739.2901749.
Lai, S., Xu, L., Liu, K., Zhao, J., 2015. Recurrent convolutional neural networks for text classification. In: Twenty-ninth AAAI Conference on Artificial Intelligence.
Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2017. Bug localization with combination of deep learning and information retrieval. In: Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22–23, 2017, pp. 218–229. doi:10.1109/ICPC.2017.24.
Le, X.-B.D., Bao, L., Lo, D., Xia, X., Li, S., Pasareanu, C., 2019. On reliability of patch correctness assessment. In: Proceedings of the 41st International Conference on Software Engineering. IEEE Press, pp. 524–535.
Le, X.-B.D., Le, T.-D.B., Lo, D., 2015. Should fixing these failures be delegated to automated program repair? In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 427–437.
Le, X.B.D., Lo, D., Le Goues, C., 2016. History driven program repair. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1. IEEE, pp. 213–224.
Li, K., Xie, J., Sun, X., Ma, Y., Bai, H., 2011. Multi-class text categorization based on lda and svm. Procedia Eng. 15, 1963–1967.
Lientz, B.P., Swanson, E.B., 1981. Problems in application software maintenance. Commun. ACM 24 (11), 763–769.
Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., Traon, Y. L., 2017. Mining fix patterns for findbugs violations. arXiv:1712.03201.
Liu, K., Kim, D., Koyuncu, A., Li, L., Bissyande, T. F. D. A., Le Traon, Y., 2018. A closer look at real-world patches.
Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M., 2005. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005, pp. 190–200. doi:10.1145/1065010.1065034.
Maas, A.L., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C., 2011. Learning word vectors for sentiment analysis. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-volume 1. Association for Computational Linguistics, pp. 142–150.
Martin, R.A., 2007. Common weakness enumeration. Mitre Corporation.
Martinez, M., Duchien, L., Monperrus, M., 2013. Automatically extracting instances of code change patterns with AST analysis. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22–28, 2013, pp. 388–391. doi:10.1109/ICSM.2013.54.
McCallum, A., Nigam, K., et al., 1998. A comparison of event models for naive bayes text classification. In: AAAI-98 Workshop on Learning for text Categorization, 752. Citeseer, pp. 41–48.
Mikolov, T., Yih, W.-t., Zweig, G., 2013. Linguistic regularities in continuous space word representations. In: Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 746–751.
Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12–17, 2016, Phoenix, Arizona, USA., pp. 1287–1293.
Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2013. A statistical semantic language model for source code. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18–26, 2013, pp. 532–542. doi:10.1145/2491411.2491458.
Pellin, B. N., 2000. Using Classification Techniques to Determine Source Code Authorship. White Paper Department of Computer Science, University of Wisconsin.
Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., Guibas, L.J., 2015. Learning program embeddings to propagate feedback on student code. In: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6–11 July 2015, pp. 1093–1102.
Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P.T., 2016. On the "naturalness" of buggy code. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, pp. 428–439. doi:10.1145/2884781.2884848.
Ray, B., Kim, M., Person, S., Rungta, N., 2013. Detecting and characterizing semantic inconsistencies in ported code. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, pp. 367–377. doi:10.1109/ASE.2013.6693095.
Ray, B., Nagappan, M., Bird, C., Nagappan, N., Zimmermann, T., 2015. The uniqueness of changes: characteristics and applications. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16–17, 2015, pp. 34–44. doi:10.1109/MSR.2015.11.
Raychev, V., Vechev, M.T., Yahav, E., 2014. Code completion with statistical language models. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, pp. 419–428. doi:10.1145/2594291.2594321.
Schutze, H., Manning, C.D., Raghavan, P., 2008. Introduction to information retrieval. In: Proceedings of the International Communication of Association for Computing Machinery Conference, p. 260.
Seaman, C.B., Shull, F., Regardie, M., Elbert, D., Feldmann, R.L., Guo, Y., Godfrey, S., 2008. Defect categorization: making use of a decade of widely varying historical data. In: Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9–10, 2008, Kaiserslautern, Germany, pp. 149–157. doi:10.1145/1414004.1414030.
Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.
Sun, X., Peng, X., Li, B., Li, B., Wen, W., 2016. IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra. Frontiers Comput. Sci. 10 (5), 812–831. doi:10.1007/s11704-016-5226-y.
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. SCIENCE CHINA Inf. Sci. 62 (1), 19102:1–19102:3. doi:10.1007/s11432-017-9459-5.
Sun, X., Yang, H., Leung, H., Li, B., Li, H.J., Liao, L., 2018. Effectiveness of exploring historical commits for developer recommendation: an empirical study. Frontiers Comput. Sci. 12 (3), 528–544. doi:10.1007/s11704-016-6023-3.
Sun, X., Yang, H., Xia, X., Li, B., 2017. Enhancing developer recommendation with supplementary information via mining historical commits. J. Syst. Softw. 134, 355–368. doi:10.1016/j.jss.2017.09.021.
Sun, X., Zhou, T., Li, G., Hu, J., Yang, H., Li, B., 2017. An empirical study on real bugs for machine learning programs. In: 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4–8, 2017, pp. 348–357. doi:10.1109/APSEC.2017.41.
Sun, X., Zhou, W., Li, B., Ni, Z., Lu, J., 2019. Bug localization for version issues with defect patterns. IEEE Access 7, 18811–18820. doi:10.1109/ACCESS.2019.2894976.
Tang, D., Qin, B., Liu, T., 2015. Document modeling with gated recurrent neural network for sentiment classification. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 1422–1432.
Tang, D., Wei, F., Yang, N., Zhou, M., Liu, T., Qin, B., 2014. Learning sentiment-specific word embedding for twitter sentiment classification. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 1, pp. 1555–1565.
Thung, F., Le, X.D., 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. doi:10.1109/ICPC.2015.15.
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 Volume, pp. 189–191. doi:10.1109/ICSE-C.2017.102.
White, M., Vendome, C., Vásquez, M.L., Poshyvanyk, D., 2015. Toward deep learning software repositories. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16–17, 2015, pp. 334–345. doi:10.1109/MSR.2015.38.
Xia, X., Lo, D., 2017. An effective change recommendation approach for supplementary bug fixes. Autom. Softw. Eng. 24 (2), 455–498. doi:10.1007/s10515-016-0204-z.
Xu, Z., Kremenek, T., Zhang, J., 2010. A memory model for static analysis of C programs. In: Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18–21, 2010, Proceedings, Part I, pp. 535–548. doi:10.1007/978-3-642-16558-0_44.
Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., Hovy, E., 2016. Hierarchical attention networks for document classification. In: Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1480–1489.
Zaremba, W., Sutskever, I., 2014. Learning to execute. arXiv:1410.4615.
Zhao, Y., Leung, H., Yang, Y., Zhou, Y., Xu, B., 2017. Towards an understanding of change types in bug fixing code. Inf. Softw. Technol. 86, 37–53.
Zhao, Y., Leung, H., Yang, Y., Zhou, Y., Xu, B., 2017. Towards an understanding of change types in bug fixing code. Inf. Softw. Technol. 86, 37–53. doi:10.1016/j.infsof.2017.02.003.
Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., Vouk, M.A., 2006. On the value of static analysis for fault detection in software. IEEE Trans. Software Eng. 32 (4), 240–253.
Zhong, H., Mei, H., 2018. Mining repair model for exception-related bug. J. Syst. Softw. 141, 16–31. doi:10.1016/j.jss.2018.03.046.
Zhong, H., Meng, N., 2017. An empirical study on using hints from past fixes: poster. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 20p17 - Companion Volume, pp. 144–145. doi:10.1109/ICSE-C.2017.88.
Zhong, H., Meng, N., 2018. Towards reusing hints from past fixes - an exploratory study on thousands of real samples. Empir. Softw. Eng. 23 (5), 2521–2549. doi:10.1007/s10664-017-9584-3.
Zolnierek, A., Rubacha, B., 2005. The empirical study of the naive bayes classifier in the case of Markov chain recognition task. In: Computer Recognition Systems, Proceedings of the 4th International Conference on Computer Recognition Systems, CORES'05, May 22–25, 2005, Rydzyna Castle, Poland, pp. 329–336. doi:10.1007/3-540-32390-2_38.

Ni Zhen is a student in School of Information Engineering, Yangzhou University. Her current research interests include datadriven automated software repair and software data analysis.

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

SUN Xiaobing is an associate professor in School of Information Engineering, Yangzhou University. His current research interests include intelligent software engineering, software data analytics.

CHEN Tianhao is a student in School of Information Engineering, Yangzhou University. His current research interests include bug localization and software data analysis.

TANG Ben is a student in School of Information Engineering, Yangzhou University. His current research interests include recommendation algorithm, Question Answering System.

SHI Xinchen is a student in School of Information Engineering, Yangzhou University. His current research interests include knowledge graph and intelligence analysis.