Automatic Defect Categorization Based On Fault Triggering Conditions

Xin Xia*‡, David Lo†, Xinyu Wang*, and Bo Zhou*§
*College of Computer Science and Technology, Zhejiang University, China
†School of Information Systems, Singapore Management University, Singapore
xxkidd@zju.edu.cn, davidlo@smu.edu.sg, {xinyuwang, bzhou}@zju.edu.cn

***Abstract*—Due to the complexity of software systems, defects are inevitable. Understanding the types of defects could help developers to adopt measures in current and future software releases. In practice, developers often categorize defects into various types. One common categorization is based on *fault triggers* of defects. Fault trigger is a set of conditions which activate a defect (i.e., fault) and propagate the defect into a failure. In general, there are two types of defect based fault triggering conditions, Bohrbug and Mandelbug. Bohrbug refers to a bug which can be easily isolated, and its activation and error propagation is simple. Mandelbug refers to a bug whose activation and/or error propagation is complex (e.g., a time lag between the fault activation and the failure occurrence). With these category labels, developers can better perform post-mortem analysis to identify common characteristic of the defects, and design specific fault-tolerance mechanisms.**

However, in most software systems, these category labels are often unavailable. To address this problem, in this paper, we propose a text mining solution which categorize defects into fault trigger categories by analyzing the natural-language description of bug reports. A previous study shows that Mandelbug is more complex and needs more time to be fixed. Thus, to better identify Mandelbugs, we propose a novel fUzzy Set based fEature Selection algorithm named USES, which selects the features (i.e., terms) which have high ability to distinguish Mandelbugs from Bohrbugs. USES first caches a set of terms based on their fuzzy affinity scores to Bohrbug or Mandelbug. Next, it iterates many times, and in each iteration, it selects a subset of terms, and builds a classifier on these terms. USES selects the classifier and the terms which could achieve the best performance on a training data. We evaluate our solution on 4 datasets including Linux, Mysql, Apache HTTPD, and AXIS containing a total of 809 bug reports. We show that USES with naive Bayes multinomial achieves the best performance; it achieves Mandelbug F-measure scores of 0.298 – 0.615. We also compare USES with other baseline approaches. The results show that USES on average improves Mandelbug F-measure scores of the best performing baseline by 12.3%.

Keywords*—*Fault Triggers, Bohrbug, Mandelbug, Feature Selection, Fuzzy Set, Categorization, Machine Learning

I. Introduction

Defects (i.e., bugs or faults) are prevalent in software systems, and appear in all stage of software development lifecycle. A previous study shows that the cost of debugging in a software system consumes 50% - 80% of the development and maintenance cost [1]. There are various kinds of defects. Some kinds of defects are easy to reproduce, and show consistent behaviors under the same inputs. Some kinds of defects are hard to reproduce, and their behaviors are inconsistent when the same inputs are given, e.g., concurrency defects [2], [3]. To ensure the reliability of software systems, the management of defects is necessary. Understanding the types of defects could help developers to perform post-mortem analysis, and adopt corresponding measures, such as adding more human resource, refactoring, and developing an automated defect detection tool, to prevent the recurrence of defects in future software releases.

One common categorization is based on *fault triggers* of defects. *Fault trigger* is a set of conditions which activate a defect (i.e., fault) and propagate the defect into a failure. In some cases, fault triggers are complex, which would cause the failures become extremely hard to reproduce. For example, the timing of events and interactions with other systems (e.g., operating system) [4]. Traditional dynamic testing techniques cannot easily detect these kinds of faults, due to the challenges of reproducing complex fault triggers in a test environment. In general, Grottke and Trivedi propose two categories of defects based fault triggering conditions, Bohrbug and Mandelbug [5]. Bohrbug refers to a bug which can be easily isolated, and its activation and error propagation is simple. Mandelbug refers to a bug whose activation and/or error propagation is complex (e.g., a time lag between the fault activation and the failure occurrence).

Albeit the benefits of categorizing defects into types, in most software systems, these category labels are often unavailable as such categorization potentially involves much manual effort, and the project team may not have the budget for defect categorization. Thus, there would be a need for an automated tool which could help developers in assigning categories to defects during post-mortem analysis.

In this paper, we propose a text mining solution which categorize defects into two *fault trigger* categories: Bohrbug and Mandelbug. Our goal is to automatically classify a defect into one of the two categories according to its natural-language description available in the corresponding bug report. Cotroneo et al. conclude that Mandelbugs need longer time to fix, and require specific strategies to be dealt with [6]. For example, in Linux, the average time to fix a Bohrbug is 157 hours, while the average time to fix a Mandelbug is 230 hours [6]. Also in some projects (e.g., apache HTTP and AXIS), the number of bugs which belongs to Mandelbug category are much less than these of Bohrbug category. Thus, we focus on identifying Mandelbugs.

[‡] The work was done while the author was visiting Singapore Management University.

[§] Corresponding author.

978-1-4799-5482-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ICECCS.2014.14

We propose a novel fUzzy Set fEature Selection algorithm named *USES*, which selects the features (i.e., terms) which have high ability to distinguish Mandelbugs from Bohrbugs. In more detail, *USES* first computes the fuzzy affinity scores for each terms to each category (i.e., Bohrbug and Mandelbug). Then, *USES* caches a set of terms which have the highest fuzzy affinity scores to Bohrbug or Mandelbug. Next, it iterates many times, and in each iteration, it selects a subset of terms, and builds a classifier on these terms. Also, *USES* evaluates the performance of each of the classifiers on a training data. Finally, the classifier and the terms which could achieve the best performance on the training data are used to predict the categories of new unlabeled defects.

We evaluate our solution on 4 datasets including Linux, Mysql, Apache HTTPD, and AXIS containing a total of 809 bug reports.[1] We show that *USES* with naive Bayes multinomial [7] achieves Mandelbug F-measure scores of 0.298 - 0.615 which outperforms a number of baseline approaches. We compare USES with other state-of-the-art classification techniques without feature selection, e.g., naive Bayes multinomial [7], naive Bayes [7], SVM [8], logistic regression [8], and RBF network [9], and naive Bayes multinomial with information gain [8] as the feature selection technique. The results show that *USES* on average improves Mandelbug F-measure scores of the best baseline techniques (i.e., naive Bayes multinomial with information gain as the feature selection technique) by 12.3%.

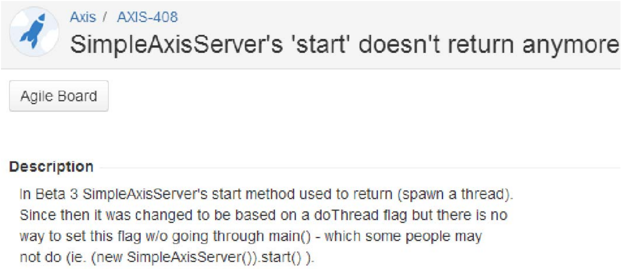
The main contributions of this paper are as follows:

1) We propose a text mining solution that automatically classifies defects into categories based on fault triggering conditions. These categories could be used for post-mortem analysis, and design fault-tolerance mechanisms.
2) Considering Mandelbugs are complex and need more time to fix, we propose a fuzzy set feature selection method *USES* to select important features from the natural language description of defects.
3) We perform an empirical evaluation of our automated defect categorization tool. The result based on 809 manually categorized defects shows that our tool improves a number of baseline techniques by a substantial margin.

The remainder of the paper is organized as follows. We describe the preliminary materials and motivating examples in Section II. We outline the overall framework of our defect categorization based on fault triggering conditions solution in Section III. We elaborate how the features are extracted from bug reports in Section IV. We present fuzzy set feature selection method *USES* in Section V. We report the experiment results in Section VI. We present the threats to validity of our paper in Section VII. We describe related work in Section VIII. We conclude and mention future work in Section IX.

II. Preliminaries and Motivating Example

In this section, we first describe preliminary materials on fault trigger categories in Section II-A. Next, we present examples which motivate the need for classifying defects into fault trigger categories in Section II-B.



SimpleAxisServer's 'start' doesn't return anymore

Description

In Beta 3 SimpleAxisServer's start method used to return (spawn a thread).
Since then it was changed to be based on a doThread flag but there is no
way to set this flag w/o going through main() - which some people may
not do (ie. (new SimpleAxisServer()).start()).

Fig. 1. An Example of Bohrbug in AXIS with BUGID=AXIS-1261.

A. Fault Trigger Categories

Grottke and Trivedi define two types of defects based on their fault trigging conditions, i.e., Bohrbug and Mandelbug [5]. These two categories are related to the conditions of fault activation and error propagation.

Definition 1: (**Bohrbug.**) A bug (i.e., fault) which can be easily isolated and manifests consistently under a well-defined set of conditions, since its activation and error propagation lack "complexity".

Definition 2: (**Mandelbug.**) A bug which is difficult to isolate, and/or the failures caused by it are hard to reproduce. Its activation and/or error propagation conditions are complex, where "complexity" can take the following two forms:

1) A time lag between fault activation and failure occurrence.
2) The activation and/or error propagation conditions depend on interactions between the conditions occurring inside the application and conditions that accrue within the system-internal environment (e.g., hardware, operating system).

According to Grottke and Trivedi, a software defect only belongs to exactly one of the above two categories [5]. Thus, these two types are complementary of each other.

B. Motivating Examples

Figure 1 shows an example of Bohrbug in AXIS with BugID=AXIS-408.[2] It describes a defect that the method new `SimpleAxisServer()).start()` does not return anymore. To reproduce this defect, developers just need to call the method `SimpleAxisServer()).start()` in their source code.

Figure 2 shows an example of Mandelbug in AXIS with BugID=AXIS-1261.[3] It describes a defect that causes a `NullPointerException` to be thrown out. The root cause of this defect is a race condition. Notice that this defect would be hard to reproduce, since it only happens occasionally. Thus, the defect is classified as a Mandelbug by Cotroneo et al. [6].

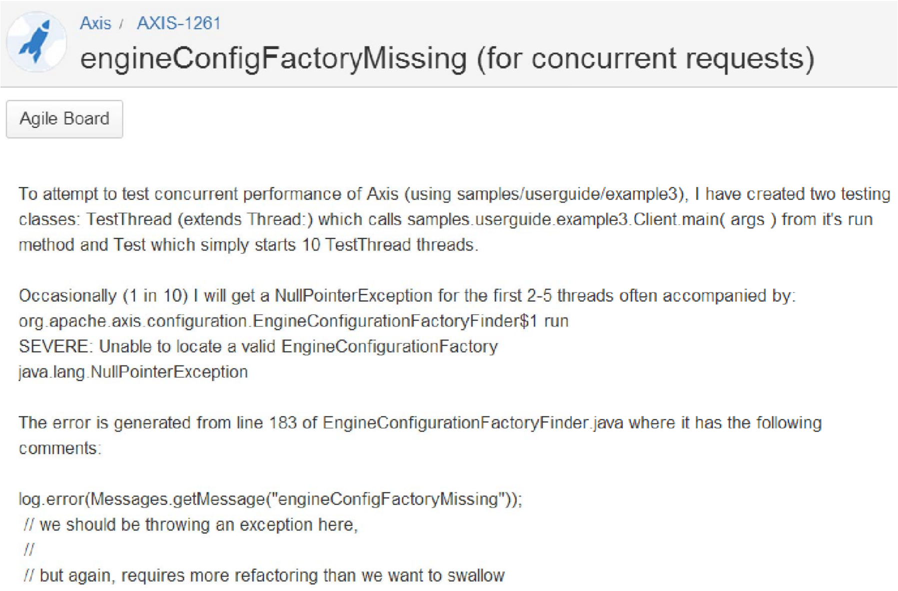
Observations and Implications. From the above 2 defects, we can observe the following:

1) Considering both Bohrbug and Mandelbug, we find Mandelbug would be difficult to fix, since it is hard

[1] We have made the dataset publicly available from: http://goo.gl/aeKoGR

[2] https://issues.apache.org/jira/browse/AXIS-408

[3] https://issues.apache.org/jira/browse/AXIS-1261



engineConfigFactoryMissing (for concurrent requests)

To attempt to test concurrent performance of Axis (using samples/userguide/example3), I have created two testing classes: TestThread (extends Thread:) which calls samples.userguide.example3.Client.main(args) from it's run method and Test which simply starts 10 TestThread threads.

Occasionally (1 in 10) I will get a NullPointerException for the first 2-5 threads often accompanied by:

```
org.apache.axis.configuration.EngineConfigurationFactoryFinder$1 run
SEVERE: Unable to locate a valid EngineConfigurationFactory
java.lang.NullPointerException
```

The error is generated from line 183 of EngineConfigurationFactoryFinder.java where it has the following comments:

```
log.error(Messages.getMessage("engineConfigFactoryMissing"));
 // we should be throwing an exception here,
 //
 // but again, requires more refactoring than we want to swallow
```

Fig. 2. An Example of Mandelbug in AXIS with BUGID=AXIS-1584.

to reproduce, and its fault trigging conditions are much more complex. Also previous study shows that Mandelbug needs more time to fix [6]. From the descriptions of defects in Figure 1 and 2, we notice that the defect in Figure 2 is much more difficult to fix, since it is a concurrency bug, and to reproduce the occurrence of the bug, developers need to collect more information.

2) Some terms in a bug report are good indicators to identify whether it is a Mandelbug, while some other terms are noise. For example, in Figure 2, the terms “concurrent”, “occasionally” could help to identify a Mandelbug; while the terms “error”, “test”, and “class” are noise since both Bohrbug and Mandelbug are likely to use these terms when describing a defect. Thus, it is good to select good indicators (i.e., useful terms), and remove noise (i.e., useless terms) from the natural-language description of a defect.

The above observations tell us that we could use the natural-language description of defects to categorizes defects based on their fault triggering conditions, and selecting good terms (indicators) from defects could help to improve classification performance. Therefore, an automated tool which applies text mining techniques on the natural-language description of defects could assist developers to categorize defects. Based on these defect categories, developers can perform post-mortem analysis and decide appropriate defect management strategies, e.g., refactoring of some components that are often affected by Mandelbugs.

III. Overall Framework

Figure 3 shows our defect categorization framework. The whole framework includes two phases: model building phase and prediction phase. In the model building phase, our goal is to build a classifier (i.e., statistical model) by leveraging text mining techniques from historical bug reports with known labels (i.e., configuration or not). In the prediction phase, this classifier would be used to predict if an unknown bug report would be a Bohrbug or Mandelbug.

Our framework first extracts features from a set of training bug reports (i.e., bug reports with known status) (Step 1). Features are various quantifiable characteristics of bug reports that could potentially distinguish defects that are related to Mandelbugs from those are related Bohrbugs. The goal of feature extraction is to reduce the defects to some important, quantitative aspects. In this paper, we use textual features from the natural-language description of bug reports. Our framework extracts the texts from various fields (e.g., description, and summary fields) of bug reports. For each text, our framework tokenizes them, removes stop words (e.g., I, you, he, the), stems them (i.e., reduces them to their root forms, e.g., “configuration” and “configure” are reduced to “config”), and represents them in the form of a “bag of words” [10].[4].

Then, our framework applies our fuzzy set based feature selection (*USES*) techniques to select a subset of relevant textual features to further improve the prediction performance (Step 2).[5]

After we select a subset of textual features, our framework next constructs a classifier (i.e., statistical model) based on the selected textual features of the training bug reports (Step 3). A classifier is a statistical model which assigns labels (in our case: Bohrbug or Mandelbug) to a data point (in our case: a defect) based on its textual features. The classifier construction phase would compare and contrast the features of bug reports that are Bohrbugs bugs, and those of bug reports that are Mandelbugs. Various classification algorithms can be used to build the classifier, e.g., naive Bayes multinomial [7], naive

[4] Detailed information of the feature extraction is presented in Section IV.
[5] Detailed information of USES is presented in Section V.

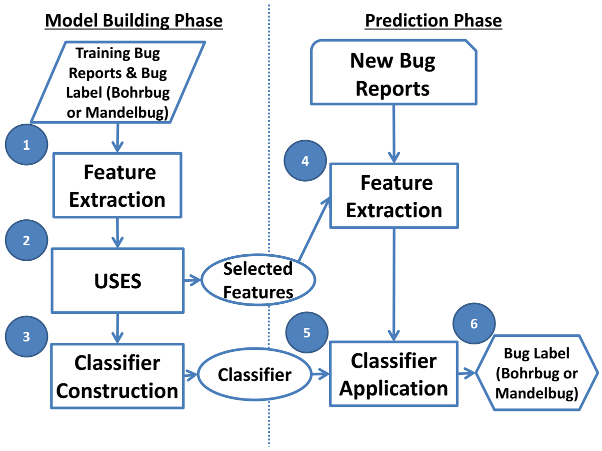


Fig. 3. Proposed Defect Categorization Framework.

Bayes [7], SVM [8], logistic regression [8], RBF network [9], and decision tree [8].

In the prediction phase, the classifier is then used to predict the categories of a defect with unknown label. For each of such bug reports, our framework first preprocesses and extracts textual features from it, and represents it by using the features selected in the model building phase (Step 4). Next, these features are input into the classifier in the classifier application step (Step 5). This step would output the prediction result which is one of the following labels: Bohrbug or Mandelbug (Step 6).

IV. Pre-Processing and Feature Extraction

In this section, we first present our text pre-processing in Section IV-A. Then, we describe the textual feature extraction in Section IV-B.

A. Text Pre-Processing

We pro-process the natural-description of defects in 3 steps: tokenization, stop-word removal, and stemming.

1) Tokenization: Tokenization is the process that breaks a stream of text up into words, phrases, symbols, or other meaningful elements called tokens. We remove all numbers and punctuation marks appearing in the reports as they often have weak correlation with defect categories. We then extract the remaining word tokens.

2) Stop-word Removal: Stop words are words that are used often and carry little meaning to distinguish different categories of defects. Examples of stop word include "I", "you", "he", "the". We take a set of standard stop words from WVTool.[6] These stop words are removed from the extracted word tokens.

3) Stemming: Stemming is the process to reduce inflected (or sometimes derived) words to their stem, base or root form. For example, the words "fishing", "fisher", and "fished" would all be reduced to "fish". We employ the well-known Porter stemmer[7] to reduce a word to its representative root form.

B. Feature Extraction

A typical bug report contains many fields. Some fields such as summary and description fields provide the natural-language description of a bug. The summary field is a condensed representation of a bug report, and the description field provides more detailed information. For example, Figures 1 and 2 present the summary and description fields of two bug reports in AXIS. In this paper, we extract features from the summary and description fields of bug reports.

We first perform text preprocessing strategies as described in Section IV-A. Next, we extract three kinds of textual features from these two bug report fields:

$TEXT^{Sum}$: We extract pre-processed word tokens from the summary field of a bug report as features. The value of each feature is the number of times the corresponding word token appears in the bug report.

$TEXT^{Desc}$: We extract pre-processed word tokens from the description field of a bug report as features. The value of each feature is the number of times the corresponding word token appears in the bug report.

$TEXT^{All}$: We extract pre-processed word tokens from both summary and description fields of a bug report as features. The value of each feature is the number of times the corresponding word token appears in the bug report.

By default, we use $TEXT^{All}$ as the features to construct a classifier, and we would also compare the performance of using $TEXT^{Sum}$ and $TEXT^{Desc}$ in Section VI.

V. Fuzzy Set Based Feature Selection Method

We use the processed word tokens as the features to construct a classifier. In total, we have a large number of word tokens. In machine learning literature, a feature can be viewed as a dimension, and a data point (i.e., a defect) can then be viewed as a point in a high-dimensional space. An overly high number of dimensions can cause the *curse-of-dimensionality* problem [8]. Aside from this, we notice for some datasets (e.g., apache HTTPD, and AXIS), only a very small minority of defects are Mandelbugs, i.e., class imbalance problem exists [11].

To address the above problems, in this section, we propose a new fuzzy set based feature selection method *USES*. We first present the definition of fuzzy category-word affinity score in Section V-A. Next, we present our feature selection method USES in Section V-B.

A. Fuzzy Category-Word Affinity Score

We denote the category of the i^{th} defects as c_i, and following vector space modeling [10], we represent the textual description of the i^{th} defect as a vector $Defect_i = \langle t_{i,1}, t_{i,2}, \cdots, t_{i,v}\rangle$, where $t_{i,j}$ is 1 if the word token w_j appears in the i^{th} defects and 0 otherwise, and v is the total number of unique terms in the whole defect collection. Based on these notations, we define fuzzy category-word affinity score as follows:

Definition 3: (**Fuzzy Category-Word Affinity Score.**) Consider a historical defect collection D, and a set of categories C. For each category $c \in C$, and word $w \in D$, the fuzzy

[6] Available from: http://sourceforge.net/projects/wvtool/
[7] http://tartarus.org/Xmartin/PorterStemmer/

category-word affinity score of c and w, denoted as $\mathit{Aff}(c, w)$, is computed as follows:

$$\mathit{Aff}(c, w) = \frac{n_{c,w}}{n_c + n_w - n_{c,w}} \quad (1)$$

where $n_{c,w}$ denotes the number of defects whose descriptions contain word w and are of category c, n_c denotes the number of defects that are of category c, and n_w denotes the number of defects that contain word w.

In our paper, we have two categories, Bohrbug and Mandelbug. We denote Bohrbug as B, and Mandelbug as M. Thus, for each word tokens $w \in D$, we have two fuzzy category-word affinity scores, $\mathit{Aff}(B, w)$, and $\mathit{Aff}(M, w)$. We define the feature selection score of word w as follows:

Definition 4: (**Feature Selection Score.**) For each word $w \in D$, its fuzzy Bohrbug-term affinity score and fuzzy Mandel-term affinity score are denoted as $\mathit{Aff}(B, w)$, and $\mathit{Aff}(M, w)$. The feature selection score of w, denoted by $Feature(w)$, is the difference between these two scores, i.e.,

$$\mathit{Feature}(w) = Aff(M, w) - Aff(B, w) \quad (2)$$

Notice that the feature selection score for a word token w could be a positive or a negative value. If a defects has many words with large positive feature selection scores then there is a high likelihood that it will be a Mandelbug. On the other hand, if a defect has many words with large negative feature selection scores, then there is a high likelihood that it is a Bohrbug.

Table I presents an example of a dataset with 4 words and 2 categories (Bohrbug (B) and Mandelbug (M)). We want to compute the fuzzy category-term affinity scores of word 1, i.e., $\mathit{Aff}(B, \mathit{Word\ 1})$, and $\mathit{Aff}(M, \mathit{Word\ 1})$. We notice that two defects (Defects 1 and 3) are Bohrbugs, i.e., $n_B = 2$; Also, $\mathit{Word\ 1}$ appears in two defects (Defects 1 and 3), i.e., $n_{Word\ 1} = 2$. Furthermore, defects 1 and 3 both have $\mathit{Word\ 1}$ and are Bohrbugs, i.e., $n_{B,Word\ 1} = 2$. Thus, the fuzzy affinity score for $\mathit{Word1}$ and Bohrbug, denoted by $\mathit{Aff}(B, \mathit{Word\ 1})$, is

$$\mathit{Aff}(B, Word\ 1) = \frac{2}{2+2-2} = 1$$

Similarly, the fuzzy affinity score for word 1 and Mandelbug, i.e., $\mathit{Aff}(M, \mathit{Word\ 1})$, is

$$\mathit{Aff}(M, Word\ 1) = \frac{0}{2+0-0} = 0$$

Finally, the feature selection score for $\mathit{Word\ 1}$, i.e., $Feature(Word\ 1)$, is:

$$\mathit{Feature}(\mathit{Word\ 1}) = \mathit{Aff}(M, Word\,1) - \mathit{Aff}(B, Word\,1) = -1$$

TABLE I. An Example of A Dataset with 4 Words and 2 Categories (Bohrbug (B) and Mandelbug (M)). A Value in A Cell Is 1, If The Corresponding Word Exists in The Description of The Corresponding Defect, And It Is 0 Otherwise.

Defect ID	**Word 1**	**Word 2**	**Word 3**	**Word 4**	**Category**
1	1	0	1	1	**B**
2	0	1	0	1	**M**
3	1	0	1	0	**B**
4	0	1	0	0	**M**

```
USES($D$, $p\%$, $sp\%$, $ITER$)
Input:
$D$: Training bug report collection
$p\%$: Percentage of features to be selected
$sp\%$: Percentage of training bug reports for classifier building
$ITER$: Number of iterations
Output: $selectedFeatures$
$selectedFeatures$: Final selected features (i.e., words)
Method:
Let $N$ = Total number of features (i.e., unique words) in $D$
Let the number of selected features $l$ = $N \times p\%$;
Compute feature selection scores for each word in $D$;
$candPos$ = $l$ words with the largest positive feature selection scores from $D$;
$candNeg$ = $l$ words with the largest negative feature selection scores from $D$;
Merge $candPos$ and $candNeg$ into one set $Merge$.
Divide $D$ into two subsets $D_{build}$ and $D_{validate}$ according to $sp\%$;
Let $bestFMeasure$ = 0;
Let $iter$ = 0;
Let $selectedFeatures$ = {};
while $iter \leq ITER$ do
    Randomly select a subset of $l$ words ($T_{tmp}$) from $Merge$;
    Build a classifier from $D_{build}$ based on the selected $l$ words;
    Evaluate the Mandelbug F-measure $f_{tmp}$ of the classifier using $D_{validate}$;
    if $f_{tmp} > bestFMeasure$ then
        $bestFMeasure = f_{tmp}$;
        $selectedFeatures$=$T_{tmp}$;
    end if
    $iter = iter + 1$;
end while
Return $selectedFeatures$;
```

Fig. 4. *USES*: fUzzy Set based fEature Selection Algorithm

B. USES

Since there are many features and feature selection scores could be positive or negative, the selection of good features for effective classification is a challenging problem. One naive way is to select an equal number of features with the highest positive and negative scores. However, we propose *USES* that can perform better.

USES first selects l features (i.e., words) with highest positive feature selection scores, and l features with highest negative feature selection scores. In total, USES takes as input $2l$ features. Then, it iterates many times. And in each iteration, it randomly selects a subset of l features from the $2l$ features, and a classifier is built on the l features. We divide the *training* dataset into two subsets: one is used to select l features, and another is used to evaluate the performance of the selected l features. The classifier and the features with the best performance are selected. We set the value of l to be high enough (i.e., 20% of the total number of features) so that a sufficient number of positive and negative word tokens are included.

Figure 4 presents our proposed method. First, we compute the number of features l as $N \times p\%$, where N is the total number of features (i.e., unique words) in a training bug report collection D and p is a parameter that decides the percentage of the features that would be selected. Then, we compute the feature selection scores for each word in D, and choose l words with the largest positive feature selection scores ($candPos$) and l words with the largest negative feature selection scores ($candNeg$), and merge these two sets (i.e., $candPos$ and $candNeg$) into one set $Merge$ (Lines 13, 14, and 15). Next, we divide the training set of bug reports D into two subsets: D_{build} and $D_{validate}$. We put $sp\%$ of the training bug reports into D_{build} and the remaining into $D_{validate}$. By default, we put 90% of the defects in D inside D_{build}, and the remaining 10% of the defects inside $D_{validate}$. Then, we iterate the process $ITER$ times. By default, we set the number of iterations as 100 (i.e., $ITER$=100). For each iteration, we randomly select a subset of l words (denoted as T_{tmp}) from $Merge$ (Line 19). We train a classifier based on the selected features using D_{build}, and investigate the classifier's performance on $D_{validate}$, to evaluate how good a set of features (i.e., T_{tmp}) is. We record the Mandelbug F-measure that is achieved by the classifier. F-measure is a common measure to evaluate how good a classifier is. It is the harmonic mean of precision and recall. Precision refers to the proportion of defects predicted as Mandelbugs that are correctly predicted. Recall refers to the proportion of Mandelbugs that are correctly identified. We record the set T_{tmp} that gives us the highest F-measure.

VI. Experiments and Results

In this section, we evaluate the effectiveness of our proposed tool. The experimental environment is an Intel(R) Core(TM) i5 3.20 GHz CPU, 4GB RAM desktop running Windows 7 (32-bit). We first present our experiment setup, evaluation metrics, and 4 research questions in Section VI-A, VI-B, and VI-C, respectively. We then present our experiment results that answer the four research questions (Sections VI-D, VI-E, VI-F, and VI-G).

A. Experiment Setup

We evaluate our proposed tool on 4 datasets from different open source software projects: Linux, Mysql, Apache HTTPD, and AXIS. These datasets were used by Cotroneo et al. in a previous empirical study [6]. Cotroneo et al. have manually analyzed hundreds of bug reports and classified each of them as a Bohrbug or a Mandelbug. For the Mysql dataset, we remove 7 duplicated defects from the original dataset by Cotroneo et al. Table II presents the statistics of the bug reports from the 4 projects. The columns correspond to the project name (Project), the number of bug reports collected (# Bugs), the time period of the collected bug reports (Time), the number of Mandelbugs (# M.), the number of Bohrbugs (# B.), and the number of unique terms (# Term). We remove terms which appear less than 5 times to reduce noise.

Stratified ten-fold cross validation [8] is used to evaluate the performance of our automated tool. We randomly divide the dataset into 10 folds. Of these 10 folds, 9 folds are use to train a classifier, while the last one fold is used to evaluate the performance of classifier. The whole process is iterated 10 times, and the average performance across the 10 iterations is recorded. Moreover, the distribution of labels in the training and test folds are the same as the original dataset to simulate the actual usage of our tool. Stratified cross validation is a standard evaluation setting, which is widely used in software engineering studies, c.f., [12]–[17].

TABLE II. Statistics of Collected Datasets.

Project	#Bugs	Time	#M.	#B.	#Term
Linux	267	2003.07-2011.05	145	122	883
Mysql	202	2006.08-2011.02	78	124	515
HTTPD	141	2002.03-2007.10	25	116	376
AXIS	199	2001.07-2005.11	15	184	396

TABLE IV. Confusion Matrix.

Classified as	True Class	
	Mandelbug	Bohrbug
Mandelbug	TP	FP
Bohrbug	FN	TN

We implement *USES* on top of Weka [18].[8] By default, we set the number of selected words as 20% of the total number of words in the defects collections (i.e., $p\% = 20\%$). Also, we use naive Bayes multinomial as the classifier after we leverage *USES* to select words – we denote the combination of naive Bayes multinomial and *USES* as $USES^B$.

B. Evaluation Metrics

To evaluate the predictive performance of our proposed tool, we create a confusion matrix to store prediction results. Table IV presents an example of a confusion matrix. The rows of the matrix correspond to predicted labels of defects. The columns of the matrix correspond to correct labels of defects. A cell in the matrix contains the number of bug reports of a particular predicted label and a particular correct label.

For each defects, there would be fours possible outcomes: a defect can be classified as a Mandelbug when it truly is a Mandelbug (true positive, TP); it can be classified as a Mandelbug when it is a Bohrbug (false positive, FP); it can be classified as a Bohrbug when it is a Mandelbug (false negative, FN); or it can be classified as a Bohrbug and it truly is a Mandelbug (true negative, TN). By using the values stored in the confusion matrix, in this paper, we calculate the accuracy, precision, recall and F-measure scores for each label (i.e., Mandelbug and Bohrbug) to evaluate the performance of our proposed tool.

Accuracy: the number of correctly classified defects (both Mandelbugs and Bohrbugs) over the total number of bugs, i.e., $Acc = \frac{TP+TN}{TP+FP+TN+FN}$.

Mandelbug Precision: the proportion of defects that are correctly labeled as Mandelbugs among those labeled as Mandelbugs, i.e., $P(C) = \frac{TP}{TP+FP}$.

Mandelbug Recall: the proportion of Mandelbugs bugs that are correctly labeled, i.e., $R(C) = \frac{TP}{TP+FN}$.

Bohrbug Precision: the proportion of bugs that are correctly labeled as Bohrbugs among those labeled as Bohrbugs, i.e., $P(NC) = \frac{TN}{TN+FN}$.

Bohrbug Recall: the proportion of Bohrbugs that are correctly labeled, i.e., $R(NC) = \frac{TN}{TN+FP}$.

F-measure: a summary measure that combines both precision and recall – it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision). For Mandelbug F-measure, it is $F(C) = \frac{2*P(C)*R(C)}{P(C)+R(C)}$. And for Bohrbug F-measure, it is $F(NC) = \frac{2*P(NC)*R(NC)}{P(NC)+R(NC)}$.

[8] http://www.cs.waikato.ac.nz/ml/weka/

TABLE III. MANDELBUG F-MEASURE, BOHRBUG F-MEASURE, AND ACCURACY FOR $USES^B$, NAIVE BAYES MULTINOMIAL (NBM), NAIVE BAYES (NB), SVM, LOGISTIC REGRESSION (LR), AND RBF NETWORK (RBF). THE LAST COLUMN SHOWS THE AVERAGE MANDELBUG F-MEASURE, BOHRBUG F-MEASURE, AND ACCURACY SCORES ACROSS THE 4 DATASETS.

Evaluation	**Techniques**	**Linux**	**Mysql**	**HTTPD**	**Axis**	**Average**
Mandelbug F-measure	$USES^B$	**0.524**	**0.615**	**0.375**	**0.298**	**0.453**
	NBM	**0.427**	**0.605**	**0.304**	**0.261**	**0.399**
	NB	0.427	0.538	0.393	0.273	0.408
	SVM	0.695	0.196	0	0	0.223
	LP	0.502	0.497	0.391	0.054	0.361
	RBF	0.535	0.538	0.333	0.174	0.395
Bohrbug F-measure	$USES^B$	**0.587**	**0.758**	**0.872**	**0.906**	**0.781**
	NBM	**0.599**	**0.762**	**0.864**	**0.903**	**0.782**
	NB	0.599	0.710	0.833	0.910	0.763
	SVM	0.090	0.763	0.903	0.961	0.679
	LP	0.502	0.693	0.881	0.903	0.745
	RBF	0.528	0.741	0.863	0.949	0.770
Accuracy	$USES^B$	**55.8%**	**70.3%**	**78.7%**	**83.4%**	**72.1%**
	NBM	**52.8%**	**70.3%**	**77.3%**	**82.9%**	**70.8%**
	NB	52.8%	64.4%	73.8%	83.9%	68.7%
	SVM	54.3%	63.4%	82.3%	92.5%	73.1%
	LP	50.2%	61.9%	80.1%	82.4%	68.7%
	RBF	55.8%	60.9%	83.0%	92.0%	71.9%

Notice that precision and recall are both important metrics for defect categorization since they measure quality of our tool in two aspects. If the precision is low, then the developer would not use the tool, due to a high number of false positives. If the recall is low, developers would not use the tool also, since most Mandelbugs (Bohrbugs) are not successfully predicted. There is a trade off between precision and recall, and one can increase precision by sacrificing recall (and vice versa) [8]. F-measure, which is the harmonic mean of precision and recall, is often used to judge whether an increase in precision outweighs a loss in recall (and vice versa) [8]. In many past papers in software engineering literature, e.g., [15], [19]–[21], F-measure is often used as a summary measure. Moreover, since Mandelbugs are more complex and require more time to be fixed [6], Mandelbug F-measure is the most important evaluation metric.

C. Research Questions

We are interested to answer the following research questions:

RQ1 *How effective is our proposed tool? How much improvement could our proposed tool achieve over vanilla classification techniques without any feature selection?*

In our proposed tool, we first use USES to select features, and then we build a classifier based on the selected features. We would like to investigate whether our proposed tool could achieve better performance over standard classification techniques without any feature selection. Answer to this research question would shed light to whether our proposed tool advances existing classification methods. To answer this research question, we select 5 state-of-the-art classification techniques, i.e., naive Bayes multinomial [7], naive Bayes [7], SVM [8], logistic regression [8], and RBF network [9]. For SVM, we use its libSVM implementation [22]. For the other techniques, we use their implementations in Weka. We compare $USES^B$ with these techniques.

RQ2 *Can USES achieve a better performance over other feature selection techniques such as information gain, gain ratio, and one rule [8]?*

In data mining literature, there are many feature selection techniques. Information gain, gain ratio, and one rule based approaches are popular feature selection techniques [8]. We want to investigate whether USES could achieve a better performance over information gain (IG), gain ratio (GR), and one rule (OneR). We first select features using USES, IG, GR, and OneR, and then for each resulting feature set, we create a classifier. We compare the effectiveness of these classifiers to demonstrate whether USES is better than the other feature selection techniques.

RQ3 *What are the impact of using various textual features on the effectiveness of our proposed approach?*

Based on the description in Section IV-B, we can extract 3 types of feature sets from bug reports, i.e., $Text^{Sum}$, $Text^{Desc}$, $Text^{All}$. We would like to investigate the effectiveness of these 3 feature sets to categorize defects. To answer this research question, we build 3 classifiers each using one of these 3 feature sets, and evaluate the performance of these 3 classifiers.

RQ4 *Do different numbers of selected features affect the performance of our proposed tool?*

By default, USES selects 20% of all textual features. We investigate whether different numbers of selected features (i.e., words) would affect the performance of our proposed tool. To answer this research question, we vary the number of selected features from 5%, 10%, 15%, 20%, 25% to 30% of the total number of features.

D. RQ1: Performance of Our Tool

Table III presents the Mandelbug F-measure, Bohrbug F-measure, and accuracy for $USES^B$ compared with naive Bayes multinomial, naive Bayes, SVM, logistic regression, and RBF network. The Mandelbug F-measure, Bohrbug F-measure, and accuracy for $USES^B$ vary from 0.298 – 0.615, 0.587 – 0.906, and 55.8% – 83.4%, respectively. Considering all five other classification techniques, we notice that naive Bayes multinomial achieves the best performance. Our proposed $USES^B$ improves the Mandelbug F-measure of naive Bayes multinomial by 22.7%, 1.7%, 23.4%, and 14.2% for Linux, Mysql, Apache HTTPD, and Axis, respectively. Averaging across the 4 datasets, the average improvement achieved by $USES^B$ is 15.48%.

TABLE V. Our Proposed Tool USES with Naive Bayes Multinomial(NBM) VS. Information Gain with NBM (IG).

Evaluation	Techniques	Linux	Mysql	HTTPD	Axis	Average.
Mandelbug F-measure	$USES^B$	0.524	0.615	0.375	0.298	0.453
	IG	0.486	0.532	0.308	0.286	0.403
	Impro.	**7.8%**	**15.6%**	**21.8%**	**4.2%**	**12.3%**
Bohrbug F-measure	$USES^B$	0.587	0.758	0.872	0.906	0.781
	IG	0.557	0.699	0.843	0.916	0.754
	Impro.	**5.4%**	**8.4%**	**3.4%**	**-1.1%**	**4.0%**
Accuracy	$USES^B$	55.8%	70.3%	78.7%	83.4%	72.1%
	IG	52.4%	63.4%	74.5%	84.9%	68.8%
	Impro.	**6.4%**	**10.9%**	**5.7%**	**-1.8%**	**5.3%**

TABLE VI. Our Proposed Tool USES with Naive Bayes Multinomial(NBM) VS. Gain Ratio with NBM (GR).

Evaluation	Techniques	Linux	Mysql	HTTPD	Axis	Average.
Mandelbug F-measure	$USES^B$	0.524	0.615	0.375	0.298	0.453
	GR	0.486	0.532	0.302	0.286	0.402
	Impro.	**7.8%**	**15.6%**	**24.2%**	**4.2%**	**12.9%**
Bohrbug F-measure	$USES^B$	0.587	0.758	0.872	0.906	0.781
	GR	0.557	0.699	0.838	0.916	0.753
	Impro.	**5.4%**	**8.4%**	**4.1%**	**-1.1%**	**4.2%**
Accuracy	$USES^B$	55.8%	70.3%	78.7%	83.4%	72.1%
	GR	52.4%	63.4%	74.5%	84.9%	68.8%
	Impro.	**6.4%**	**10.9%**	**5.7%**	**-1.8%**	**5.3%**

We notice that for SVM, its Mandelbug F-measure for Linux is quite high, i.e., 0.695. However, considering its Bohrbug F-measure for Linux, we find that SVM predicts nearly every defect as Mandelbug, since in Linux, Mandelbugs are the majority. For Apache HTTPD and Axis, SVM also nearly predicts every defect as Bohrbug, since Bohrbugs are the majority. Thus, although the average accuracy of SVM is higher than $USES^B$, it does not mean that SVM performs better.

E. RQ2: USES vs. IG, GR, and OneR

Table V presents the experiment results of our $USES^B$ compared to information gain with naive Bayes multinomial (IG). We notice that the differences in F-measures and accuracy are substantial. $USES^B$ improves the average Mandelbug F-measure, Bohrbug F-measure, and accuracy of IG by 12.3%, 4.0%, and 5.3%, respectively. Table VI presents the experiment results of our $USES^B$ compared to gain ratio with naive Bayes multinomial (GR). We notice that the differences in F-measures and accuracy are substantial. On average across the 4 datasets, $USES^B$ improves the average Mandelbug F-measure, Bohrbug F-measure, and accuracy of GR by 12.9%, 4.2%, and 5.3%, respectively. Table VII presents the experiment results of our $USES^B$ compared to one rule with naive Bayes multinomial (OneR). We notice that the differences in F-measures and accuracy are substantial. On average across the 4 datasets, $USES^B$ improves the average Mandelbug F-measure, Bohrbug F-measure, and accuracy of OneR by 68.3%, 4.2%, and 4.6%, respectively.

F. RQ3: Effect of Various Textual Features

Table VIII presents the experiment results of using various textual features, i.e., $Text^{All}$, $Text^{Desc}$, and $Text^{Sum}$. We notice that $Text^{All}$ achieves the best performance as compared to $Text^{Desc}$, and $Text^{Sum}$.

G. RQ4: Effect of Varying the Number of Selected Features

Figure 5 presents the Mandelbug F-measure of $USES^B$ for various numbers of selected features (i.e., words) for Linux, Mysql, Apache HTTPD, and Axis. Notice that for small numbers of features (e.g., 5%, 10%, and 15% of the number of distinct words), the Mandelbug F-measure scores are unstable. For example, for Apache HTTPD, its Mandelbug F-measure is 0.4 when 5% of the number of distinct words are selected, it becomes 0.49 when 10% of the number of distinct words are selected, and it becomes 0.353 when 15% of the number of distinct words are selected. For a larger number of features (e.g., 20%, 25%, and 30% of the number of distinct words), the Mandelbug F-measure scores are more stable. For example, for Axis, its Mandelbug F-measure is 0.298 when 20% of the number of distinct words are selected, it becomes 0.298 when 25% of the number of distinct words are selected, and it becomes 0.273 when 30% of the number of distinct words are selected.

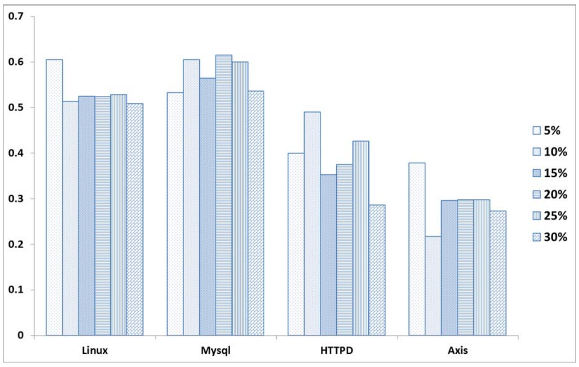


Fig. 5. Mandelbug F-measures for Different Numbers of Selected Features (5% to 30% of the Number of Distinct Words in the Training Data).

VII. Threats to Validity

Threats to internal validity relate to errors in our experiments. We use the same datasets provide by Cotroneo et al. [6]. We have double checked our experiments and removed the duplicated defects from the datasets, still there could be errors that we do not notice.

TABLE VII. Our Proposed Tool USES with Naive Bayes Multinomial(NBM) VS. One Rule with NBM (OneR).

Evaluation	**Techniques**	**Linux**	**Mysql**	**HTTPD**	**Axis**	**Average.**
Mandelbug F-measure	$USES^B$	0.524	0.615	0.375	0.298	0.453
	OneR	0.498	0.547	0.302	0.293	0.091
	Impro.	**5.2%**	**12.4%**	**28.0%**	**227.5%**	**68.3%**
Bohrbug F-measure	$USES^B$	0.587	0.758	0.872	0.906	0.781
	OneR	0.548	0.7	0.88	0.887	0.754
	Impro.	**7.1%**	**8.3%**	**-0.9%**	**2.1%**	**4.2%**
Accuracy	$USES^B$	55.8%	70.3%	78.7%	83.4%	72.1%
	OneR	52.4%	63.9%	79.4%	80.0%	68.9%
	Impro.	**6.4%**	**10.1%**	**-0.9%**	**4.4%**	**4.6%**

TABLE VIII. Various Text Features Used for Classification.

Evaluation	**Features**	**Linux**	**Mysql**	**HTTPD**	**Axis**	**Average.**
Mandelbug F-measure	$Text^{All}$	0.524	0.615	0.375	0.298	0.453
	$Text^{Desc}$	0.478	0.595	0.383	0.292	0.437
	$Text^{Sum}$	0.690	0.383	0.200	0.00	0.318
Bohrbug F-measure	$Text^{All}$	0.587	0.758	0.872	0.906	0.781
	$Text^{Desc}$	0.551	0.740	0.877	0.903	0.768
	$Text^{Sum}$	0.313	0.754	0.905	0.961	0.733
Accuracy	$Text^{All}$	55.8%	70.3%	78.7%	83.4%	72.1%
	$Text^{Desc}$	51.7%	68.3%	79.4%	82.9%	70.6%
	$Text^{Sum}$	57.3%	64.9%	83.0%	92.5%	74.4%

Threats to external validity relate to the generalizability of our results. We have analyzed 809 bug reports from 4 open source software projects. In the future, we plan to reduce this threat further by analyzing more bug reports from open source and commercial software projects .

Threats to construct validity refer to the suitability of our evaluation metrics. We use Mandelbug F-measure as the main evaluation metric. F-measure has also been used by past software engineering studies to evaluate the effectiveness of a prediction technique [15], [19]–[21]. Thus, we believe there is little threat to construct validity.

VIII. Related Work

In this section, we discuss some related works on characterizing or predicting the types of bugs in Section VIII-A. Next, we briefly review some works on text mining for software engineering in Section VIII-B.

A. Characterization and Prediction of Bugs

There have been a number of studies on characterizing or predicting the types of bugs [6], [23]–[27]. Gegick et al. propose the usage of text mining techniques to identify whether a bug is a security bug or not [23]. Zaman et al. perform an empirical study on security bugs and performance bugs in Firefox [25]. They find security bugs need more time to be fixed, while performance bugs are not that different from other bugs, in terms of bug fix time, but more files need to be changed to fix them. Arshad et al. extract configuration bugs from GlassFish and JBoss, and they characterize configuration bugs from several dimensions, i.e., problem-type, problem-time, problem-manifestation, and problem-culprit [24]. Based on their findings, they also develop a tool named ConfGauge which injects parameter-based configuration issues into software systems. Cotroneo et al. perform an empirical study on Borhbug and Mandelbug in open-source software projects, and they conclude that Mandelbugs need longer time to fix, and require specific strategies to be dealt with [6]. Xia et al. perform an empirical study on bugs in software build systems such as Ant, Maven, CMake and QMake, and they find that 21.35% of the build system bugs are related to external interface problems [26]. Thung et al. propose a method to automatically categorize bug reports into 3 families: control and data flow, structural, and non-functional [27]. Xia et al. propose the usage of data mining and feature selection techniques to identify configuration bugs [28]. Our work complements the above studies; we classify a bug as a Bohrbug or a Mandelbug.

B. Text Mining for Software Engineering

There have been a number of studies on text mining for software engineering [20], [29]–[33]. The survey here is by no means complete. Sun et al. propose a text mining technique and extend BM25F to accurately detect duplicated bug reports [29], [30]. Later, Nguyen et al. combine topic model and BM25F to achieve a better performance in detecting duplicated bug reports [31]. Wu et al. propose Relink which leverages information retrieval techniques to recover links between bugs and their corresponding changes [20]. Marcus and Maletic use Latent Semantic Indexing (LSI) to recover traceability links between documentation to source code [32]. Zhou et al. propose an approach that takes in a bug report and return source code files that are likely to be relevant to the input bug report [34]. Haiduc et al. use automated text summarization to produce succinct and informative text to comprehend software code [33]

IX. Conclusion and Future Work

In this paper, we propose a text mining solution which categorizes defects into *fault trigger* categories by analyzing the natural-language description of bug reports. Considering Mandelbugs are more complex, and need more time to fix, we propose a novel fuzzy set based feature selection algorithm named *USES*, which selects the features (i.e., terms) which have high ability to distinguish Mandelbugs from Bohrbugs. *USES* first caches a set of terms based on their fuzzy affinity scores to Bohrbug or Mandelbug. Next, it iterates many times, and in each iteration, it selects a subset of terms, builds a classifier on these terms, and evaluates it using a subset of a training data. At the end of the iterations, *USES* selects the classifier and the terms which could achieve the best Mandelbug F-measure scores on thetraining data. We evaluate our solution on 4 datasets including Linux, Mysql, Apache HTTPD, and AXIS containing a total of 809 bug reports. We show that *USES* with naive Bayes multinomial achieves the best performance over many baseline approaches. On average across the 4 projects, *USES* improves Mandelbug F-measure scores of information gain with naive Bayes multinomial, which is the best performing baseline, by 12.3%.

In the future, we plan to evaluate our proposed tool with more defects from more software projects, and develop a better technique which could further improve the performance of defect categorization based on fault triggering conditions. We also plan to build a model leveraging transfer learning [35] to predict Mandelbugs by using data from other projects.

Acknowledgment

This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (2014BAH24F02).

References

[1] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of systems and software*, vol. 9, no. 3, pp. 191–195, 1989.

[2] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006.

[3] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM Sigplan Notices*, vol. 43, no. 3. ACM, 2008, pp. 329–339.

[4] J. Gray, "Why do computers stop and what can be done about it?" in *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA, 1986, pp. 3–12.

[5] M. Grottke and K. S. Trivedi, "A classification of software faults," *Journal of Reliability Engineering Association of Japan*, 2005.

[6] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 178–187.

[7] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752. Citeseer, 1998, pp. 41–48.

[8] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[9] M. J. Orr *et al.*, "Introduction to radial basis function networks," 1996.

[10] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[11] H. He and E. A. Garcia, "Learning from imbalanced data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 9, pp. 1263–1284, 2009.

[12] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 249–258.

[13] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 134–143.

[14] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.

[15] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 386–396.

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

[17] F. Thung, D. Lo, and J. L. Lawall, "Automated library recommendation," in *WCRE*, 2013, pp. 182–191.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[19] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 63.

[20] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes." in *SIGSOFT FSE*, 2011, pp. 15–25.

[21] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[22] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[23] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 11–20.

[24] F. A. Arshad, R. J. Krause, and S. Bagchi, "Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss," pp. 198–207, 2013.

[25] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[26] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *Quality Software (QSIC), 2013 13th International Conference on*. IEEE, 2013, pp. 200–203.

[27] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[28] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *38th Annual International Computers, Software & Applications Conference (COMPSAC)*. IEEE, 2014.

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

[30] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.

[31] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 70–79.

[32] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 125–135.

[33] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2. IEEE, 2010, pp. 223–226.

[34] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, 2012, pp. 14–24.

[35] S. J. Pan and Q. Yang, "A survey on transfer learning," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 1345–1359, 2010.