Soft Computing (2020) 24:1045–1061
https://doi.org/10.1007/s00500-019-03942-3

METHODOLOGIES AND APPLICATION

A variable-level automated defect identification model based on machine learning

Yuwei Zhang[1] · Ying Xing[2] · Yunzhan Gong[1] · Dahai Jin[1] · Honghui Li[3] · Feng Liu[3]

Published online: 23 March 2019

Abstract
Static analysis tools, automatically detecting potential source code defects at an early phase during the software development process, are diffusely applied in safety-critical software fields. However, alarms reported by the tools need to be inspected manually by developers, which is inevitable and costly, whereas a large proportion of them are found to be false positives. Aiming at automatically classifying the reported alarms into true defects and false positives, we propose a defect identification model based on machine learning. We design a set of novel features at variable level, called variable characteristics, for building the classification model, which is more fine-grained than the existing traditional features. We select 13 base classifiers and two ensemble learning methods for model building based on our proposed approach, and the reported alarms classified as unactionable (false positives) are pruned for the purpose of mitigating the effort of manual inspection. In this paper, we firstly evaluate the approach on four open-source C projects, and the classification results show that the proposed model achieves high performance and reliability in practice. Then, we conduct a baseline experiment to evaluate the effectiveness of our proposed model in contrast to traditional features, indicating that features at variable level improve the performance significantly in defect identification. Additionally, we use machine learning techniques to rank the variable characteristics in order to identify the contribution of each feature to our proposed model.

Keywords Machine learning · Static analysis · Automated defect identification · Alarm classification · Model evaluation

1 Introduction

Software testing based on defect pattern (Quinlan et al. 2007) is a source code static analysis technology developed in this century. For its high efficiency and accuracy, various static analysis tools, such as Coverity (Bessey et al. 2010), PREfix (Bush et al. 2000), Defect Testing System (DTS) (Yang et al. 2008) and FindBugs (Ayewah and Pugh 2010), have been widely applied in automatically detecting potential source code defects at an early software development phase.

Communicated by V. Loia.

✉ Yuwei Zhang
hyun@bupt.edu.cn

[1] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China

[2] Automation School, Beijing University of Posts and Telecommunications, Beijing, China

[3] School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China

Although static analysis tools have proven themselves to be useful and significant in some domains, several researches (Johnson et al. 2013; Kumar and Nori 2013; Beller et al. 2016; Christakis and Bird 2016) demonstrate that such tools are faced with challenges in practice. One of the most crucial challenges involves false positives, which is a common problem of software testing based on defect pattern. Because static analysis technology cannot obtain the dynamic execution information, static analysis tools are required to speculate on how the program will behave actually (Ruthruff et al. 2008). As a result, a large scale of alarms reported by the tools are found to be false positives, which is inevitable (Dillig et al. 2012). Therefore, manual inspection of the reported alarms would be a costly and unavoidable work for developers.

To mitigate the effort of manual inspection, efficient defect identification techniques for handling static analysis alarms have been put forward by numerous studies and summarized in a few literature reviews (Heckman and Williams 2011; Muske and Serebrenik 2016). One of the promising approaches addressing the problem is to come up with a set of artifact characteristics for classifying alarms as actionable

Table 1 Evaluated projects

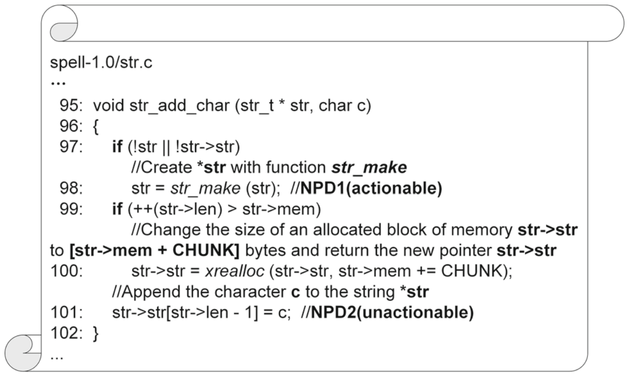
Project	Download website	Description	Size (KLOC)	# of IPs	NPD rate (%)
antiword-0.37	https://launchpad.net/ubuntu/+source/antiword/0.37-11	A free MS Word reader for Linux	20	90	52
spell-1.0	https://ftp.gnu.org/gnu/spell/	A spell checking program	2	74	84
sphinxbase-0.3	https://sourceforge.net/projects/cmusphinx/files/sphinxbase/	A speech recognition toolkit	23	240	59
uucp-1.07	http://www.airs.com/ian/uucp.html	A complete UUCP package	53	943	76

and unactionable, probability-based ranking of each alarm being true, and clustering by the similarity of alarms. The artifact characteristics are based on semantic, structure or historical information about the source codes and reported alarms.

The objective of our research is to build a binary classification model based on machine learning methods for automatically identifying the reported alarms. The reported alarms are classified either as actionable (true defects) or unactionable (false positives), and the unactionable ones are pruned and not reported to developers, thus reducing the workload of manual inspection. Despite the fact that numerous machine learning methods have been applied in the classification of alarms with positive results, no individual learner can always achieve perfect performance due to the limitation of each machine learning algorithm. Therefore, ensemble learning methods, combining multiple classifiers with strategies, have become a better choice for classification tasks. We select 13 base classifiers in our research for building defect identification models. They are selected from five categories built in Weka in order to reflect the diversity of machine learning algorithms and the fairness of evaluation analysis. And then, two ensemble learning methods are introduced to improve the classification performance compared to base classifiers.

In this paper, we discuss the potential defects from four open-source C projects (listed in Table 1 with full description) detected by DTS, a tool to catch defects in source code using static testing techniques (Yang et al. 2008). The amount of reported alarms from each project, called inspection points (IPs) in DTS, is listed in the fifth column of Table 1. Since the characteristics of different defect patterns are specific, the model proposed in this paper is merely for null pointer dereference (NPD) that occupies the majority of the IPs reported by DTS, which can be seen in the *NPD Rate* column.

In order to establish a defect-specific model, we need to firstly draw the discrepancy between true defects and false positives from the source codes related to the reported alarms. The existing approaches have manually designed various features to classify the alarms, such as software metrics features, source code history and churn features based on file or module level, and alarms-based features. However, these features



```
spell-1.0/str.c
...
 95:  void str_add_char (str_t * str, char c)
 96:  {
 97:      if (!str || !str->str)
              //Create *str with function str_make
 98:          str = str_make (str);  //NPD1(actionable)
 99:      if (++(str->len) > str->mem)
              //Change the size of an allocated block of memory str->str
to [str->mem + CHUNK] bytes and return the new pointer str->str
100:          str->str = xrealloc (str->str, str->mem += CHUNK);
          //Append the character c to the string *str
101:      str->str[str->len - 1] = c;  //NPD2(unactionable)
102:  }
...
```

Fig. 1 A motivating example

lack precision in representing the distinct semantics of alarms leading to a large amount of false positives. For example, Fig. 1 shows a C language function `str_add_char` from `spell` with two alarms reported by DTS. Alarm NPD1 is determined as actionable after manual inspection because of dereferencing of a parameter `str` that may be null, while alarm NPD2 is determined as unactionable. The feature vectors of these two alarms are identical under traditional features, because these two alarms have the same characteristics in terms of **if** statements, assignment statements and lines of code, etc. However, the manual inspection results of these two alarms are opposite. Therefore, false positives may occur when we use traditional features to classify the reported alarms.

To bridge the gap between the reported alarms' semantics and features used for defect identification, a set of novel features at variable level, named variable characteristics (VCs), are raised in this paper, which is based on the related information of variables that cause defects. Each reported alarm can be transformed into one feature vector with designed VCs via a mapping function. Then, a predictive model can be trained using machine learning methods in an ensemble way to automatically identify new reported alarms either as actionable or unactionable. The contributions of this paper are fourfold:

– Two ensemble learning methods and 13 base classifiers are selected to build the automated defect identification models in order to mitigate the effort of manual inspection.

- A set of novel artifact characteristics at variable level, named variable characteristics, are designed to build the proposed model.
- Experiments are conducted on four open-source C projects to evaluate the performance of our approach in the case of identifying alarms reported from the same project.
- Variable characteristics are ranked by three different single attribute evaluators to identify the impact of these features on our proposed model.

The rest of this paper is organized as follows. We survey related work in Sect. 2. Sections 3 and 4 describe variable characteristics and model building process, respectively. We provide the experimental setup in Sect. 5. Section 6 shows the analyzing results of model evaluation. We conclude this paper and present the future work in Sect. 7.

2 Related work

There have been an increasing number of approaches for handling static analysis alarms, and the approaches are categorized in a few literature reviews (Heckman and Williams 2011; Muske and Serebrenik 2016). One of the promising approaches is in a position to simplify the inspection effort by designing a set of features from the related information of source codes and reported alarms for clustering, ranking schemes and classification tasks.

Clustering Static analysis alarms are clustered by the dependence among them. Since the grouped alarms are dominated by the alarm on which they are depending, not only the number of alarms that need to be inspected is reduced, but also the superfluous inspection effort is eliminated (Le and Soffa 2010; Lee et al. 2012; Zhang et al. 2013; Podelski et al. 2016; Muske et al. 2018). Podelski et al. (2016) proposed a set of semantic-based features for each alarm, and the alarms of the same feature values were grouped. Le and Soffa (2010) constructed a correlation graph by collecting the data about the characteristics of fault correlations, and this graph could integrate fault correlations on different paths and among multiple faults. Zhang et al. (2013) presented a sound alarm correlation algorithm based on trace semantic to automate alarm identification. Muske et al. (2018) described a novel technique that reduces alarms by repositioning, which uses the information of control flow to group the related alarms.

Ranking One ranking approach is to make a priority of the alarms that have a high probability to be true defects, and artifact characteristics are used to compute the likelihood of each alarm being actionable. Jung et al. (2005) made use of syntactic alarm context as input of Bayesian to compute the probability of each alarm to be true and ranked the alarms based on the probability before reporting. Kim and Ernst (2007a, b) put forward a warning prioritization algorithm based on the software change history features that were mined from the source code repository, while their underlying intuition was that alarms eliminated by fix-changes are important. On the similar lines, Williams and Hollingsworth (2005) raised a method to utilize the source code change history of a software project to drive and help to refine the search for defects. Addressing the weakness of other ranking schemes, that is, rankings should be adaptive as reports are inspected, Kremenek et al. (2004) took advantage of correlation behavior among reports and user feedback for alarm ranking. Compared with the clustering approach, this approach needs to inspect all the ranked alarms.

Classification The classification approach can identify whether the alarms as actionable or unactionable. The unactionable alarms are not reported to the users for these alarms are more likely to be false positives. Ayewah et al. (2007) discussed the kinds of generated alarms and classified the types of alarms into false positives, trivial bugs and serious bugs. Ruthruff et al. (2008) proposed a logistic regression model based on 33 features extracted from the alarms themselves to predict actionable alarms found by FindBugs, and a screening methodology was used to quickly discard features with low predictive power in order to build cost-effectively predictive models. Reynolds et al. (2017) used a set of descriptive attributes to standardize the patterns of false positives. Several studies (Brun and Ernst 2004; Yi et al. 2007; Heckman and Williams 2009; Liang et al. 2010; Yuksel and Sözer 2013; Hanam et al. 2014; Yoon et al. 2014; Flynn et al. 2018) have utilized machine learning classification models to abstract the difference between the actionable alarms and the unactionable alarms for automatically identifying defects. Brun and Ernst (2004) presented a machine learning-based technique that builds models of program properties and used the built models to classify the program properties that may lead to latent defects. Heckman and Williams (2009) evaluated 15 machine learning algorithms based on distinct sets of alarms characteristics out of 51 candidate characteristics, which is one of the most comprehensive studies in predicting actionable alarms and achieves high performance. Additionally, they proposed a benchmark in Heckman and Williams (2008) for evaluation and comparison of the automated defect identification models. Liang et al. (2010) constructed a training set automatically to compute the learning weights effectively for different features, and then, the reported alarms were ranked and classified by computing the scores using these learning weights. Hanam et al. (2014) put forward a method for differentiating actionable and unactionable alarms by finding similar code patterns that are based on the code surrounding each static analysis alarm. Flynn et al. (2018) developed and

Table 2 Semantic metrics

Assignment statements	
VC	Description
AS_C	Related variable assigned by a **constant** value
AS_V	Related variable assigned by a **variable** value
AS_NULL	Related variable assigned by a **null** value
AS_LFUNC	Related variable assigned by a **library function** return value
AS_UFUNC	Related variable assigned by an **user-defined function** return value
AS_IF	Related variable assigned in an **if** statement
AS_FOR	Related variable assigned in a **for** statement
AS_WHILE	Related variable assigned in a **while** statement
Reference statements	
VC	Description
RS_V	Related variable referenced by a **variable**
RS_LFUNC	Related variable referenced by a **library function**
RS_UFUNC	Related variable referenced by an **user-defined function**
RS_IF	Related variable referenced in an **if** statement
RS_FOR	Related variable referenced in a **for** statement
RS_WHILE	Related variable referenced in a **while** statement
Control-flow statements	
VC	Description
CS_IF	Related variable occurs in an **if** statement
CS_FOR	Related variable occurs in a **for** statement
CS_WHILE	Related variable occurs in a **while** statement

tested four classification models for static analysis alarms mapped to CERT rules, using a novel combination of multiple static analysis tools and 28 features extracted from the alarms.

To the best of our knowledge, no research gives a set of artifact characteristics at variable level. In this paper, alarms are reported with detailed description after detected by DTS, including the variable information of each static analysis alarm. Information from source codes and reported alarms is extracted to represent variable characteristics, which will be described in Sect.3.

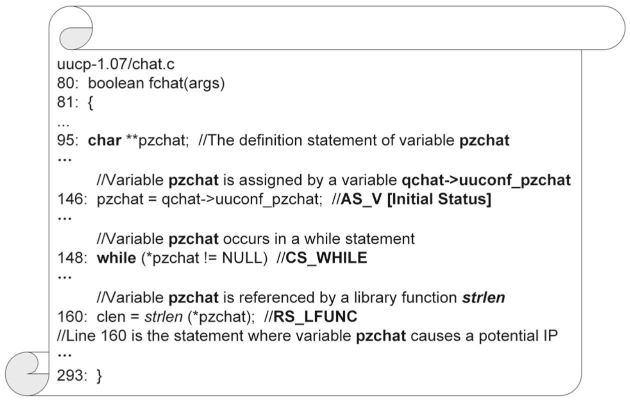
3 Variable characteristics

There are a growing number of features designed in the existing studies (Heckman and Williams 2009; Podelski et al. 2016; Hanam et al. 2014; Yuksel and Sözer 2013; Yoon et al. 2014) to classify the alarms. In this paper, a set of novel artifact characteristics, called VCs, are designed for each reported alarm, which are based on the related information of variables that cause potential defects. These VCs are derived from three sources: the information of data flow and conditional predicate of related variable in the source codes, the lines of code (LOC) metrics, and the defect pattern definition in DTS. Details of the VCs are shown in the following three subsections.

3.1 Semantic metrics

For our paper, we firstly utilize abstract syntax tree (AST) to extract source code semantics. When analyzing source code files, we pour attention into the statements of data flow and conditional predicate of related variable, which has a great impact on leading to potential NPD defects (Wang et al. 2013). Then, we reduce the number of statements to inspect by generating a backwards program slice. A backwards program slice takes the statement containing the related variable as the seed statement and extracts three types of statements that could have affected the outcome of the seed statement as characteristics, namely assignment statements, reference statements and control-flow statements, respectively. We totally design 17 VCs based on the three types of statements listed in Table 2 with detailed description.

For example, we consider the code in Fig. 2 where the pointer variable `pzchat` (defined at line 95) causes a potential NPD defect at line 160. Line 160 is used as the seed statement for computing a backwards program slice, which



```
uucp-1.07/chat.c
80:  boolean fchat(args)
81:  {
...
95:  char **pzchat;  //The definition statement of variable pzchat
...
       //Variable pzchat is assigned by a variable qchat->uuconf_pzchat
146:  pzchat = qchat->uuconf_pzchat;  //AS_V [Initial Status]
...
       //Variable pzchat occurs in a while statement
148:  while (*pzchat != NULL)  //CS_WHILE
...
       //Variable pzchat is referenced by a library function strlen
160:  clen = strlen (*pzchat);  //RS_LFUNC
//Line 160 is the statement where variable pzchat causes a potential IP
...
293:  }
```

Fig. 2 An example of variable characteristics extraction

Table 3 Mapping between initial status cases and integers

Case	Integer
Assignment	0
Reference	1
IF	2
FOR	3
WHILE	4

will produce the set of statements at lines {95, 146, 148, 160}. In the code, the related variable `pzchat` is defined at line 95, is assigned by a variable `qchat->uuconf_pzchat` at line 146, occurs in a **while** statement at line 148, and is referenced by a library function `strlen` at line 160. Then, the following VCs are extracted:

- AS_V: line 146
- CS_WHILE: line 148
- RS_LFUNC: line 160

The initial status of the variable after defined is also considered as a variable characteristic called *I_STATE*. As an example, again consider the pointer variable `pzchat` in Fig. 2, which occurs in an assignment statement after it is defined. Therefore, we take the attitude that the initial status of such variable is **Assignment**. Furthermore, five initial cases are considered and mapped into integers to construct feature vectors as input for the classification model, which is given in Table 3.

3.2 LOC metrics

Heckman and Williams (2009) collected LOC metrics at three different levels of granularity, that is, method, file and package, respectively. Podelski et al. (2016) have utilized the LOC metrics to classify bugs. For LOC metrics containing different levels of granularity, the LOC metrics are collected at variable and method levels in our model. The LOC metrics designed in this paper are listed below:

- IP_LOC: the number of source code lines counted from the definition statement of the related variable to the statement that contains the variable causing a potential NPD defect. As shown in Fig. 2, the pointer variable `pzchat` is defined at line 95 and line 160 is the statement where `pzchat` causes a potential NPD defect, and thus, the value of *IP_LOC* is 66.
- METHOD_LOC: the number of source code lines within the method containing the variable. As shown in Fig. 2, the pointer variable `pzchat` is contained in the method `fchat`, and thus, the value of *METHOD_LOC* is 214 (counted from line 80 to line 293).

3.3 Defect pattern definition metric

In DTS, NPD is further defined as five categories in terms of analyzing C programming language projects, which is considered as a variable characteristic *CLASS*. In addition, a mapping between categories and integers is built to construct feature vectors as input for the classification model, which is shown below:

- NPD: dereferencing of a local pointer that may be null and mapped into the value of **0**.
- NPD_CHECK: dereferencing of a checked parameter or global pointer that may be null and mapped into the value of **1**.
- NPD_EXP: dereferencing of an expression that may be null and mapped into the value of **2**.
- NPD_PARAM: dereferencing of a functional returned parameter that may be null and mapped into the value of **3**.
- NPD_PRE: dereferencing of a parameter or global pointer that may be null and mapped into the value of **4**.

4 Model building process

There is a strategy for classification tasks using machine learning outlined by Witten et al. (2016), and Fig. 3 illustrates a complete procedure of building a classification model for automatically identifying defects based on the strategy. For the following four subsections, we describe the model building process proposed in this paper.

4.1 DTS's architecture

DTS is a defect pattern-driven tool. Each defect pattern is defined using a defect pattern state machine (DPSM), which

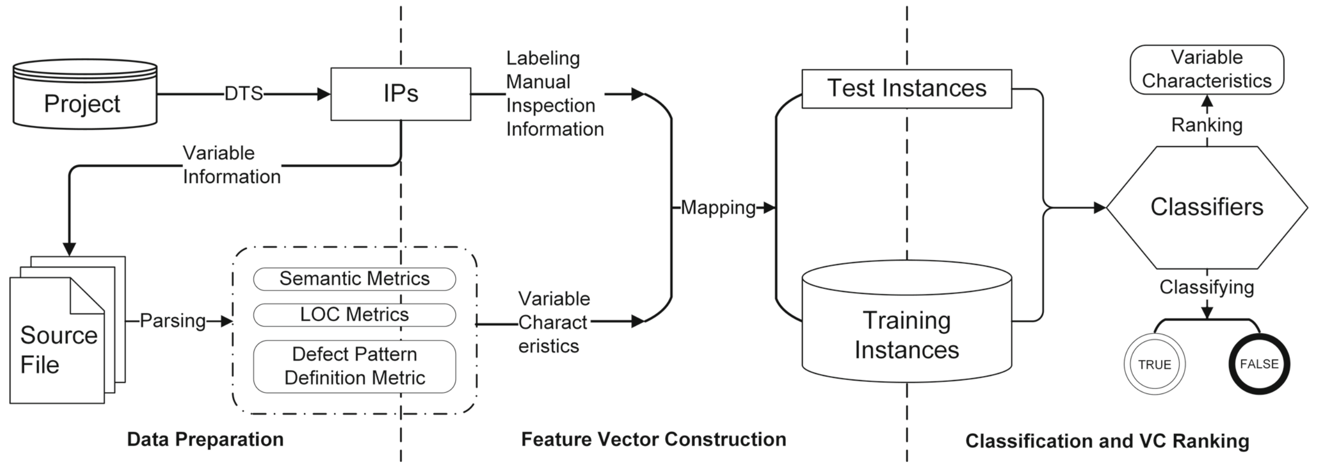


Fig. 3 Model building process

is stored in an xml file. A DPSM can be represented as a triple (S, T, C), in which:

- S is a state set and $S = \{S_{\text{start}}, S_{\text{error}}, S_{\text{end}}, S_{\text{other}}\}$, where S_{start} denotes the initial state, S_{error} denotes the error state, S_{end} denotes the end state and S_{other} denotes other intermediate states,
- T is a state transition set, which is defined as $T : S \times C \rightarrow S$,
- C is a transition condition set, which denotes the state transition conditions.

DTS's architecture is shown in Fig. 4. Firstly, DTS transforms the source code file into a program model that indicates the analyzed codes with a set of data structures. As the AST is built for the analyzed codes, the tool constructs a symbol table alongside it. The symbol table is linked to each identifier in the analyzed codes with its type and a pointer to its declaration or definition.

DTS proposes the interval computation technique in control-flow and data-flow analysis, of which the purpose is to compute the state of DPSM. Defect patterns tell the defect patterns analysis engine how to model the environment and the effects of library and system call. If a DPSM is transited to an error state, then a defect is reported by DTS. Since the static analysis tools would produce the false positives, the IPs reported by DTS are reviewed by our testing team.

4.2 Data preparation

The four evaluated projects are originally analyzed statically by DTS, and then, the reported IPs are inspected manually by the developers in order to obtain the result of each IP being either actionable or unactionable. After that, the information of the variable causing a potential NPD defect is received from the reporting log of each IP and is parsed based on the source code files, which is fully explained in Sect. 3.

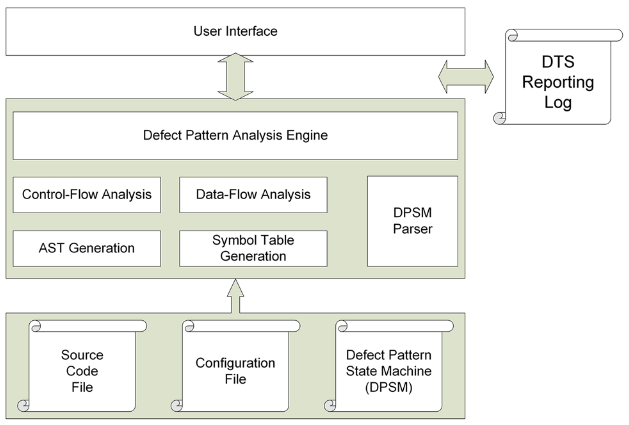


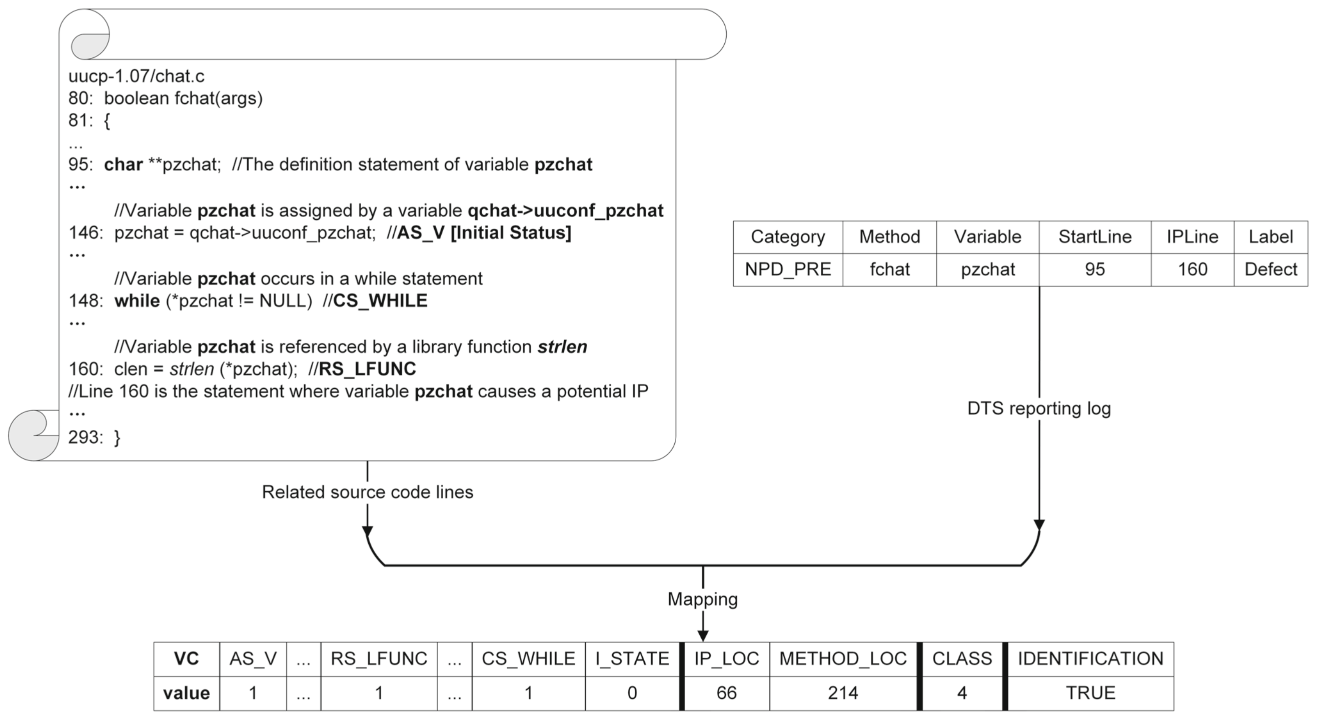
Fig. 4 DTS's architecture

4.3 Feature vector construction

As is described in Sect. 3, the feature vector is a mapping between each reported IP and the VCs we designed, and the construction process shown in Fig. 3 can be represented according to the following mapping function:

$$\mathbb{IP} \longmapsto \mathbb{FV} = (\mathbb{VC}, R(\mathbb{IP})) \tag{1}$$

where $\mathbb{IP}$ is the inspection point reported by DTS, $\mathbb{FV}$ is the mapped feature vector via Eq. (1), $\mathbb{VC}$ is a list of integer numbers denoting the value of the 21 variable characteristics calculated from $\mathbb{IP}$, and $R(\mathbb{IP})$ is the manual inspection result of $\mathbb{IP}$. The identification rule of $R(\mathbb{IP})$ is defined below:



```
uucp-1.07/chat.c
80:  boolean fchat(args)
81:  {
...
95:  char **pzchat;  //The definition statement of variable pzchat
...
        //Variable pzchat is assigned by a variable qchat->uuconf_pzchat
146:  pzchat = qchat->uuconf_pzchat;  //AS_V [Initial Status]
...
        //Variable pzchat occurs in a while statement
148:  while (*pzchat != NULL)  //CS_WHILE
...
        //Variable pzchat is referenced by a library function strlen
160:  clen = strlen (*pzchat);  //RS_LFUNC
//Line 160 is the statement where variable pzchat causes a potential IP
...
293:  }
```

Category	Method	Variable	StartLine	IPLine	Label
NPD_PRE	fchat	pzchat	95	160	Defect

VC	AS_V	...	RS_LFUNC	...	CS_WHILE	I_STATE	IP_LOC	METHOD_LOC	CLASS	IDENTIFICATION
value	1	...	1	...	1	0	66	214	4	TRUE

Fig. 5 Feature vector construction

$$R(\mathbb{IP}) = \begin{cases} \textbf{TRUE}, & \text{if } \mathbb{IP} \text{ is actionable;} \\ \textbf{FALSE}, & \text{otherwise.} \end{cases} \tag{2}$$

Figure 5 demonstrates an example of constructing a feature vector from the code in Fig. 2. The upper left corner of the figure is the source code lines of the related variable obtained from the IP's reporting log in the upper right corner. The ellipsis in the sample feature vector represent the variable characteristics that value 0. Detailed mapping process can be referred in Sect. 3.

4.4 Classification and VC ranking

To classify new IPs from the same source code project, we need to keep track of the IPs that have been classified by our testing team, along with the feature vector for each IP. This firstly requires a training phase that our testing team inspect a number of IPs and classify them as actionable or unactionable manually. Then, we use the inspected IPs to build and train defect identification models based on machine learning methods to differentiate the actionable and unactionable IPs. Finally, we allow the models to automatically classify the rest IPs in the project, thus reducing our burden of manual inspection.

Moreover, we use machine learning techniques to rank the variable characteristics in order to find out how much proportion of classification performance these features can contribute to our proposed model in Sect. 6.4.

5 Experimental setup

To evaluate our proposed approach, Weka (Witten et al. 2016), an open-source software developed by the Machine Learning Group at the University of Waikato in New Zealand, is used for model building. As shown in Fig. 6, the framework consists of three major steps. Firstly, 13 base classifiers built in Weka are selected to build individual classification model, respectively. Secondly, two ensemble learning methods are trained based on the output of the 13 base classifiers. Finally, we evaluate all of the classification models including base classifiers and ensemble learning methods by carrying out tenfold cross-validation on four open-source projects. We use the default parameters for all classifiers in Weka. Our experiments are all run on a 3.7 GHz Intel Core i3-6100 machine with 4 GB RAM.

5.1 Base classifiers

Weka contains a series of machine learning algorithms for classification tasks with understandable output results to developers. We select 13 classification algorithms from five categories built in Weka, which are fully described in Table 4. The selection of these classifiers is based on their popularity and diversity (Ruthruff et al. 2008; Heckman and Williams 2009; Yuksel and Sözer 2013; Hanam et al. 2014; Yoon et al. 2014; Flynn et al. 2018).

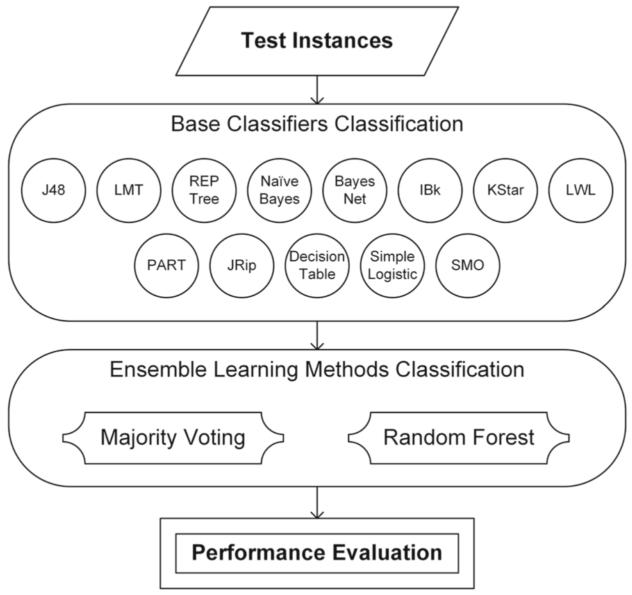


Fig. 6 Framework of alarms classification using ensemble learning methods

5.2 Ensemble learning methods

Ensemble learning methods, also called multi-classifier systems (MCSs), use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone. A machine learning ensemble trains each base classifier firstly by individual machine learning algorithm, and then, these base classifiers are integrated into one MCS by strategy. In general, ensemble learning methods can often perform better than base classifiers (Dietterich 2000). For the purpose of improving the performance of defect identification, we choose two different kinds of MCSs for building models in this paper: One is that all base classifiers in the MCS are not one single type (majority voting), and the other is that all base classifiers are of the same type (random forest). The two MCSs are described below.

5.2.1 Majority voting

Majority voting (MV) is the most common combination strategy used in ensemble learning. The voting method derives from the hypothesis that the decision of a group is superior to that of the individuals. The flowchart of MV is presented in Fig. 7. For binary classification model proposed in this paper, the ensemble consists of 13 base classifiers $\{h_1, h_2, \ldots, h_{13}\}$, and base classifier h_i predicts a label for one test instance x from the set of class label $\{c_1, c_2\}$, where c_1 denotes **TRUE** and c_2 denotes **FALSE**. If x is identified as the same class by most base classifiers, x is labeled to this class. Since the number of base classifiers is odd, the two classes cannot obtain the same voting value for x. Thus, x is always able to be labeled to one certain class. The rule of class identification for majority voting is shown in Eq. (3), where $h_i^j(x)$ denotes the prediction output of base classifier h_i on class label c_j.

$$H(x) = \begin{cases} c_1, & \text{if } \sum_{i=1}^{13} h_i^1(x) > 0.5 \sum_{j=1}^{2} \sum_{i=1}^{13} h_i^j(x); \\ c_2, & \text{otherwise.} \end{cases} \tag{3}$$

Table 4 The selected 13 base classifiers

Category	Classifier	Description
Decision tree	J48	Generating a pruned or unpruned C4.5 decision tree
	LMT	Building ‘logistic model trees,’ which are classification trees with logistic regression functions at the leaves
	REPTree	Building a decision tree using information gain and pruning it using reduced-error pruning
Bayes classifier	NaiveBayes	A Naive Bayes classifier using estimator classes
	BayesNet	Bayes Network learning using various search algorithms and quality measures
Instance-based algorithm	IBk	A K-nearest neighbors classifier
	KStar	An instance-based classifier using an entropy-based distance function
	LWL	Locally weighted learning using an instance-based algorithm to assign instance weights
Rule-based algorithm	PART	Generating a PART decision list using separate-and-conquer
	JRip	A propositional rule learner repeating incremental pruning to produce error reduction
	DecisionTable	Building and using a simple decision table majority classifier
Function-based model	SimpleLogistic	Building linear logistic regression models
	SMO	Implementing sequential minimal optimization algorithm for training a support vector classifier

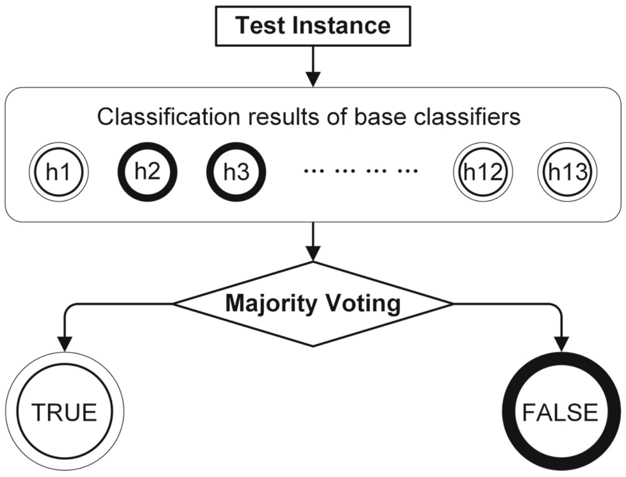


Fig. 7 Flowchart of majority voting

5.2.2 Random forest

Random forest (RF) is an ensemble learning method where each two base classifiers have no strong dependency and can be generated simultaneously and parallel. Differing from traditional decision tree algorithms, k attributes are randomly selected at each node of an individual tree in the forest during the procedure of building a random forest model. Breiman (2001) suggested that $k = \log_2 d$, where d is the size of the attribute set.

5.3 Experimental design

For within-project defect identification, datasets from the same project are split into the training set and the test set. When building models, we carry out tenfold cross-validation to evaluate the effectiveness of classification. In cross-validation, datasets are randomly split into ten approximately equal subsets, and nine of the subsets are used to train a model and the last subset to test. The process is repeated ten times in order that each of the ten subsets would be tested once. We repeat the tenfold cross-validation 100 times for each model, as randomness would occur inevitably in splitting datasets (Arcuri and Briand 2011).

Furthermore, attribute selection in Weka (Witten et al. 2016), selecting a subset of attributes using attribute evaluator with one search method, is of great importance to avoid reducing classifier performance because of redundant and irrelevant attributes. In this paper, the ranking of variable characteristics is what we concern about to show the merit of each VC to our proposed model. Variable characteristics designed in this paper are evaluated using three single attribute evaluators of Weka with *Ranker* search method (Witten et al. 2016).

5.4 Evaluation metrics

To assess the performance of models trained in this paper, the following metrics are adopted to evaluate defect identification techniques.

5.4.1 Accuracy

Accuracy is one of the most shared evaluation metrics for classification tasks. According to individual model M built on dataset D, the definition of accuracy is shown in Eq. (4), where S is the size of D, and $M(x_i)$ and y_i represent the results of the model prediction and the manual inspection, respectively.

$$\text{accuracy}(M; D) = \frac{1}{S}\sum_{i=1}^{S}\mathbb{I}(M(x_i) = y_i) \tag{4}$$

5.4.2 Kappa statistic

Kappa statistic is a coefficient for consistency test, that is, to determine whether the model prediction results and the actual results are consistent. Kappa coefficient values from 0 to 1. When the coefficient is larger than 0.6, the model prediction result is reliable.

5.4.3 Indicators derived from confusion matrix

For a binary classification problem, instances can be divided into true positive, false positive, true negative and false negative combining by the predicted results of the model and actual results of the manual inspection, which consist of the confusion matrix. The three metrics, precision (P), recall (R) and F-measure ($F1$), are derived from the confusion matrix. Here is a brief introduction:

$$P = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} \tag{5}$$

$$R = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} \tag{6}$$

$$F1 = \frac{2 * P * R}{P + R} \tag{7}$$

Precision and recall are a pair of contradictory metrics. In general, higher precision means that more true defects can be classified as positive, while a growing rate of recall can reveal more true defects. F-measure that takes consideration of both precision and recall is a harmonic mean value.

5.4.4 ROC curve and area under ROC curve

Receiver operating characteristic (ROC) curve that is widely used in machine learning is an effective tool to visualize the generalization performance of classifiers. However, ROC curve has a drawback when the curves of two classifiers are intersected, and we can hardly predicate which classifier is superior. Thus, area under ROC curve (AUC), summing up the area of each part under ROC curve, is introduced as an evaluation metric. Assuming that the ROC curve is connected by a series of points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ in sequence, AUC can be calculated as shown in Eq. (8):

$$AUC = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1} - x_i)\cdot(y_{i+1} - y_i) \tag{8}$$

6 Experimental results and analysis

6.1 Ground truth

To evaluate our proposed models, we should firstly classify the reported NPD IPs as actionable and unactionable accurately. As is described in Sect. 4.4, our method in this paper is looking through the reported NPD IPs manually and classifying them into the two classes. The result of manual inspection is given in Table 5. The *TRUE* column indicates the number of NPD IPs that are classified as actionable, and the number of unactionable NPD IPs is listed in the *FALSE* column.

We classify 890 NPD IPs in total and use these IPs in our experiments. A total of 582 of these are actionable, and 308 are unactionable. The machine learning methods mentioned in Sects. 5.1 and 5.2 are used to make a classification of these IPs, and we repeat the experiment designed in Sect. 5.3 100 times on each classifier built for each project so as to evaluate the performance of models based on our designed VCs. Furthermore, we calculate a weighted average of the evaluation metrics except accuracy and kappa statistic referred in Sect. 5.4 across both classes (**TRUE** and **FALSE**), which is according to the number of NPD IPs in each class. The weighted average (WA) is shown in Eq. (9), where $[M]_T$ denotes the metric value of precision, recall, F-measure or AUC for class **TRUE**, $[M]_F$ denotes the metric value of precision, recall, F-measure or AUC for class **FALSE**, A_{IP} denotes the number of actionable NPD IPs, and U_{IP} denotes the number of unactionable NPD IPs.

Table 5 The manual inspection result on evaluated projects

Project	TRUE	FALSE
antiword	27	12
spell	40	20
sphinxbase	42	81
uucp	473	195
Total	582	308

Table 6 Accuracy and kappa statistic results on evaluated projects

Metric	Project	J48	LMT	REPTree	NaiveBayes	BayesNet	IBk	KStar	LWL	PART	JRip	DecisionTable	SimpleLogistic	SMO	MV	RF
Accuracy	antiword	0.8272	0.8903	0.7979	0.8567	0.8277	0.8995	0.8782	0.8374	0.8423	0.8408	0.8572	0.8908	**0.9133**	0.8679	0.8874
	spell	0.8450	0.8522	0.8058	0.8507	0.7102	0.8652	0.8508	0.7762	**0.8940**	0.8668	0.7648	0.8520	0.7928	0.8462	0.8683
	sphinxbase	0.8216	0.8250	0.8177	0.5619	0.8292	0.8202	0.8511	0.8211	0.8211	0.8146	0.8170	0.7945	0.6787	0.8541	**0.8854**
	uucp	0.8708	0.8825	0.8440	0.7649	0.8306	0.8738	0.8967	0.7862	0.8600	0.8522	0.8334	0.8030	0.7704	0.8766	**0.9032**
Kappa	antiword	0.5709	0.7372	0.4899	0.6723	0.5835	0.7640	0.7177	0.6001	0.6185	0.6105	0.6550	0.7383	**0.7941**	0.6796	0.7345
	spell	0.6455	0.6448	0.5422	0.6530	0.2236	0.6800	0.6411	0.4715	**0.7596**	0.6991	0.4457	0.6429	0.4549	0.6260	0.6882
	sphinxbase	0.5902	0.5999	0.5859	0.2427	0.6180	0.5870	0.6564	0.6023	0.5963	0.5787	0.5941	0.5286	0.1509	0.6730	**0.7411**
	uucp	0.6784	0.7146	0.6004	0.3805	0.5582	0.6949	0.7456	0.3742	0.6611	0.6255	0.5528	0.4654	0.3060	0.6734	**0.7564**

Table 7 Weighted precision, recall and *F*-measure results on evaluated projects

Metric	Project	J48	LMT	REPTree	NaiveBayes	BayesNet	IBk	KStar	LWL	PART	JRip	DecisionTable	SimpleLogistic	SMO	MV	RF
Precision	antiword	0.8238	0.8909	0.7936	0.8628	0.8245	0.8999	0.8805	0.8344	0.8413	0.8392	0.8557	0.8914	**0.9134**	0.8667	0.8877
	spell	0.8443	0.8556	0.8043	0.8488	0.7089	0.8664	0.8534	0.7716	**0.8947**	0.8680	0.7613	0.8565	0.8294	0.8511	0.8693
	sphinxbase	0.8193	0.8231	0.8158	0.7656	0.8283	0.8175	0.8498	0.8211	0.8200	0.8126	0.8175	0.7909	0.6559	0.8533	**0.8848**
	uucp	0.8686	0.8822	0.8403	0.7516	0.8258	0.8739	0.8954	0.7945	0.8600	0.8488	0.8324	0.7971	0.7842	0.8801	**0.9023**
Recall	antiword	0.8272	0.8903	0.7979	0.8567	0.8277	0.8995	0.8782	0.8374	0.8423	0.8408	0.8572	0.8908	**0.9133**	0.8679	0.8874
	spell	0.8450	0.8522	0.8058	0.8507	0.7102	0.8652	0.8508	0.7762	**0.8940**	0.8668	0.7648	0.8520	0.7928	0.8462	0.8683
	sphinxbase	0.8216	0.8250	0.8177	0.5619	0.8292	0.8202	0.8511	0.8211	0.8211	0.8146	0.8170	0.7945	0.6787	0.8541	**0.8854**
	uucp	0.8708	0.8825	0.8440	0.7649	0.8306	0.8738	0.8967	0.7862	0.8600	0.8522	0.8334	0.8030	0.7704	0.8766	**0.9032**
F-measure	antiword	0.8255	0.8906	0.7958	0.8597	0.8261	0.8997	0.8793	0.8359	0.8418	0.8400	0.8564	0.8911	**0.9134**	0.8673	0.8876
	spell	0.8446	0.8539	0.8051	0.8497	0.7094	0.8658	0.8521	0.7739	**0.8943**	0.8674	0.7631	0.8542	0.8107	0.8486	0.8688
	sphinxbase	0.8205	0.8240	0.8168	0.6481	0.8287	0.8188	0.8504	0.8211	0.8205	0.8136	0.8172	0.7927	0.6670	0.8537	**0.8851**
	uucp	0.8697	0.8823	0.8422	0.7582	0.8282	0.8738	0.8960	0.7903	0.8600	0.8505	0.8329	0.8000	0.7773	0.8783	**0.9028**

$$WA = \frac{[M]_T * A_{\mathrm{IP}} + [M]_F * U_{\mathrm{IP}}}{A_{\mathrm{IP}} + U_{\mathrm{IP}}} \tag{9}$$

6.2 Evaluation metrics analysis

6.2.1 Accuracy and kappa statistic

The results of accuracy and kappa statistic are listed in Table 6, where the best result values of accuracy and kappa statistic on each evaluated project are highlighted in bold. Each column represents the machine learning methods given to the project (by row). The machine learning methods are divided by double vertical lines according to their categories as is described in Sects. 5.1 and 5.2. Overall, the average accuracy and kappa statistic (across all projects and classifiers) are 83.36% and 0.5986, respectively. According to the statistics from Table 6, half of the models' accuracy surpass 85% and the proportion of the models with kappa statistic above 0.6 is 60%, indicating that our approach based on machine learning is precise and credible enough to provide a reliable prediction result. As shown in Table 6, we can find out that the best model based on accuracy and kappa statistic for the four projects is specific. The best model for `antiword` is SMO. SMO is a support vector machine classifier and has an advantage in solving the classification of small dataset, and thus, it performs better than other machine learning methods for `antiword`. PART, a rule-based learner, is the best model for `spell`, and RF is the best model for both `sphinxbase` and `uucp`. Furthermore, the two MCSs enhance the classification accuracy and kappa statistic in most cases compared to the 13 base classifiers.

6.2.2 Weighted precision, recall and *F*-measure

Table 7 shows the results of weighted precision, recall and *F*-measure for the models, which are calculated as described in Eq. (9). The values highlighted in bold are the highest precision, recall and *F*-measure on each evaluated project. In general, the average weighted precision (across all projects and classifiers) is 0.8367, the average weighted recall is 0.8336, and the average weighted *F*-measure is 0.8349. Statistics from Table 7 shows that nearly half of the models' three indicators (weighted precision, recall and *F*-measure) exceed 0.85, indicating the high performance of our proposed approach. According to Table 7, the best model for each evaluated project is the same as discussed in the analysis of accuracy and kappa statistic. In most instances, the performance of the two MCSs surpasses the 13 base classifiers.

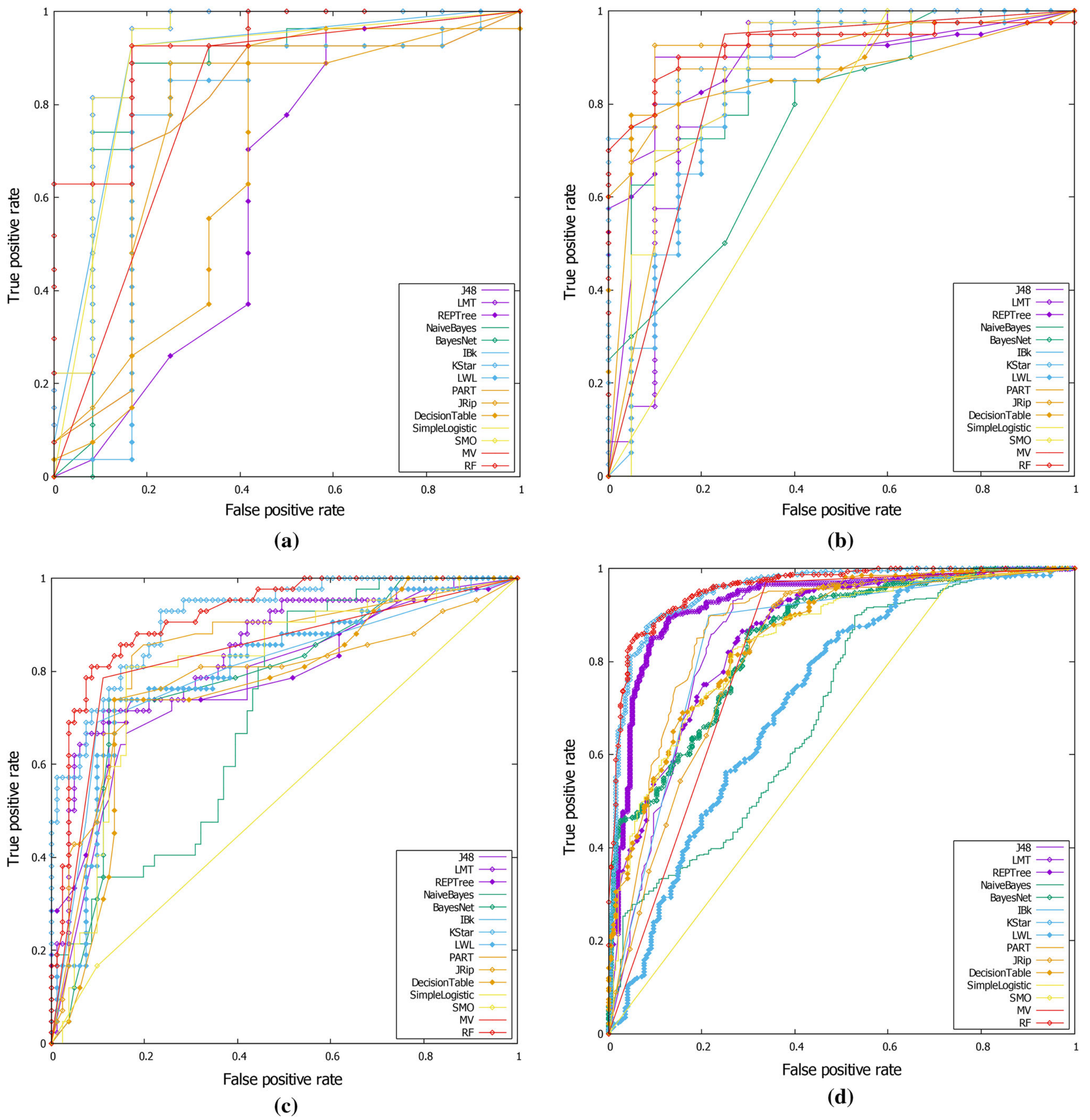


Fig. 8 ROC curves on evaluated projects: **a** antiword, **b** spell, **c** sphinxbase, **d** uucp

6.2.3 ROC curve and weighted AUC

ROC curves for each project of different classifiers are shown in Fig. 8, and Table 8 presents the classification performance of weighted AUC, where the best weighted AUC values of the classifiers built on each project are highlighted in bold. Generally, our proposed approach achieves a weighted AUC of 0.8273 on the average (across all projects and classifiers). Based on the statistics in Table 8, one-third of the models obtain a weighted AUC value that over 0.85. To be specific in all machine learning methods, RF measures up the best performance for all of the evaluated projects. However, the weighted AUC of MV is inferior to the base classifiers in most cases. Consequently, there is little or nothing we can judge on the AUC performance between MV and the base classifiers. Generally speaking, two ensemble learning methods perform well especially RF.

Table 8 Weighted AUC result on evaluated projects

Metric	Project	J48	LMT	REPTree	NaiveBayes	BayesNet	IBk	KStar	LWL	PART	JRip	DecisionTable	SimpleLogistic	SMO	MV	RF
AUC	antiword	0.7883	0.8859	0.7184	0.8472	0.8473	0.8818	0.9168	0.8146	0.7881	0.7586	0.7596	0.8860	0.8911	0.8294	**0.9303**
	spell	0.8445	0.8441	0.8220	0.8905	0.7375	0.8228	0.9417	0.7718	0.8817	0.8717	0.8810	0.8478	0.6945	0.7904	**0.9482**
	sphinxbase	0.7839	0.8459	0.7757	0.7315	0.7817	0.7863	0.9331	0.8043	0.8159	0.7614	0.8015	0.8318	0.5635	0.8339	**0.9379**
	uucp	0.8729	0.9159	0.8495	0.6928	0.8503	0.8472	0.9544	0.7444	0.8793	0.8188	0.8597	0.8431	0.6225	0.8078	**0.9573**

6.3 Baseline comparison analysis

6.3.1 A baseline for comparison

To evaluate the performance of semantic metrics we raised in Sect. 3.1 in defect identification, we compare semantic metrics with traditional metrics. Our baseline of traditional metrics consists of three VCs, that is, *IP_LOC*, *METHOD_LOC* and *CLASS* (described in Sects. 3.2 and 3.3).

6.3.2 Evaluation setup

The project for the baseline comparison is `uucp`, which is large enough to contain many IPs. We select six (J48, NaiveBayes, KStar, PART, SimpleLogistic and RF) out of 15 machine learning methods to run our experiment designed in Sect. 5.3, for their large variance and high evaluation performance according our analysis in Sect. 6.2. The six metrics described in Sect. 5.4 are used to evaluate the baseline comparison.

6.3.3 Result analysis

As is shown in Table 9, each column represents the machine learning methods given to the project `uucp`, and the evaluation results of each metric are listed by row. The machine learning methods are divided by double vertical lines according to their categories. Generally, we can observe a significant improvement of the six evaluation metrics from baseline to our approach proposed in this paper in the *Average* column. In all cases, our approach is more accurate than the baseline; that is, fewer IPs are classified incorrectly, which shows that our models minimize the number of unactionable IPs (false positives) developers need to inspect. Since the reported IPs classified as unactionable would be pruned for reducing the workload of manual inspection, the increment in precision, recall and *F*-measure indicates that our models can reveal more true defects as well as lower the rate of false negative. Overall, our proposed model with semantic metrics at variable level improves the performance in defect identification.

6.4 VC ranking analysis

Single attribute evaluator with *Ranker* evaluates each VC individually and returns an ordered list of VCs with merit values (Witten et al. 2016). Based on the merit values calculated from each evaluator for each project, the top 5 out of 21 designed VCs for each case are presented in Table 10. Among the ranked VCs in Table 10, the merit values range from 0.0356 to 0.6829. Overall, 16 out of 21 designed VCs are ranked into the top 5 at least one time, which demonstrates that over three quarters of the designed VCs have a

Table 9 The comparison of evaluation metrics results on uucp

		J48	NaiveBayes	KStar	PART	SimpleLogistic	RF	Average
Accuracy	Baseline	0.8205	0.6932	0.7804	0.7822	0.7075	0.8378	0.7703
	Our approach	0.8708	0.7649	0.8967	0.8600	0.8030	0.9032	0.8498
Kappa	Baseline	0.5460	0.1001	0.4493	0.4282	0.0012	0.5976	0.3533
	Our approach	0.6784	0.3805	0.7456	0.6611	0.4654	0.7564	0.6146
Precision	Baseline	0.8152	0.6429	0.7739	0.7741	0.5022	0.8347	0.7238
	Our approach	0.8686	0.7516	0.8954	0.8600	0.7971	0.9023	0.8458
Recall	Baseline	0.8205	0.6932	0.7804	0.7822	0.7075	0.8378	0.7703
	Our approach	0.8708	0.7649	0.8967	0.8600	0.8030	0.9032	0.8498
F-measure	Baseline	0.8178	0.6671	0.7771	0.7781	0.5874	0.8362	0.7440
	Our approach	0.8697	0.7582	0.8960	0.8600	0.8000	0.9028	0.8478
AUC	Baseline	0.7955	0.6105	0.8295	0.7983	0.5014	0.8990	0.7390
	Our approach	0.8729	0.6928	0.9544	0.8793	0.8431	0.9573	0.8667

Table 10 The top 5 VC ranking

Ranking		1st		2nd		3rd		4th		5th	
Project	Evaluator	VC	Merit	VC	Merit	VC	Merit	VC	Merit	VC	Merit
antiword	Correlation	IP_LOC	0.6829	METHOD_LOC	0.6607	RS_LFUNC	0.6158	RS_V	0.5804	AS_NULL	0.5058
	InfoGain	RS_V	0.4130	IP_LOC	0.4130	METHOD_LOC	0.4130	RS_LFUNC	0.3230	AS_NULL	0.2100
	ReliefF	AS_NULL	0.3205	CLASS	0.3103	IP_LOC	0.2488	METHOD_LOC	0.2332	RS_LFUNC	0.2226
spell	Correlation	AS_V	0.4468	RS_FOR	0.4264	I_STATE	0.4254	RS_V	0.3589	CLASS	0.3351
	InfoGain	METHOD_LOC	0.4980	AS_V	0.1752	RS_FOR	0.1434	CS_FOR	0.1127	I_STATE	0.1088
	ReliefF	METHOD_LOC	0.1559	AS_UFUNC	0.1433	I_STATE	0.1342	RS_FOR	0.0733	AS_C	0.0617
sphinxbase	Correlation	AS_UFUNC	0.3190	AS_C	0.2574	RS_UFUNC	0.2247	RS_IF	0.2177	AS_V	0.2149
	InfoGain	AS_UFUNC	0.2655	AS_C	0.1095	RS_IF	0.0683	AS_V	0.0627	RS_UFUNC	0.0627
	ReliefF	CLASS	0.0764	AS_C	0.0661	RS_IF	0.0573	RS_UFUNC	0.0553	METHOD_LOC	0.0542
uucp	Correlation	I_STATE	0.3383	CS_IF	0.3067	CS_FOR	0.2758	IP_LOC	0.1863	RS_WHILE	0.1677
	InfoGain	METHOD_LOC	0.2755	I_STATE	0.1512	CS_FOR	0.1332	RS_FOR	0.0929	CS_IF	0.0827
	ReliefF	METHOD_LOC	0.0522	I_STATE	0.0410	RS_UFUNC	0.0407	RS_WHILE	0.0382	AS_V	0.0356

little contribution to the classification performance of our proposed model.

Based on the statistics from Table 10, Fig. 9 presents the number of times that a VC is ranked into the top 5 in one of the three single attribute evaluators for each project with different color. Hereby, *METHOD_LOC*, *I_STATE* and *AS_V* appear to be the most relevant variable characteristics for classification. Moreover, *METHOD_LOC* is contained in every project at least once, which implies that the number of statements within the method containing the alarm is predictive of the actionability of the alarm. Additionally, there are five variable characteristics contained in none of the four evaluated projects, that is, *AS_LFUNC*, *AS_IF*, *AS_FOR*, *AS_WHILE*, *CS_WHILE*, respectively, which may be less important to our proposed model.

6.5 Threats to validity

Three main threats to validity in this paper, that is, external validity, internal validity and construct validity, respectively, are illustrated as follows.

6.5.1 External validity

The principal threat to external validity is that the evaluated projects in this paper may not be of enough generalization for all software projects. As a result, projects exclude in the four projects might yield better or worse performance based on our approach. But additional running of our proposed model on other projects will minimize this threat to validity. Since our model is only evaluated on open-source C projects, its performance on projects written in other programming languages is unknown.

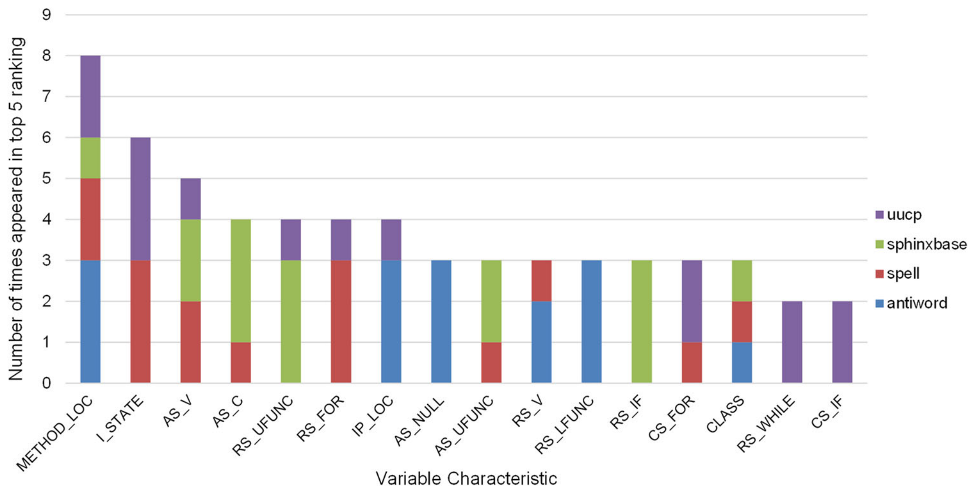


Fig. 9 Variable characteristics ranking results (ordered)

6.5.2 Internal validity

For our paper, dataset preparation is the main concern of internal validity. As is described in Sect. 4, the evaluated projects are detected by DTS and the reported IPs are inspected manually. Oversights of manual inspection could invalidate a few of the model results. Multiple examination by different developers will minimize this threat to internal validity.

6.5.3 Construct validity

We cannot generalize our proposed model since variable characteristics designed in this paper are specific for NPD defect, which may not be representative, resulting in a threat to construct validity. However, we consider various shared features including LOC metrics applied in a multitude of the existing research papers, which can contribute to improving the generalization performance of our proposed model.

7 Conclusion and future work

In order to mitigate the effort of manual inspection, this paper presents a machine learning-based model for automatically identifying null pointer dereference (NPD) defects using a set of novel and more fine-grained features. Specifically, features, called variable characteristics (VCs), are extracted from related source codes of the variable leading to a potential NPD defect analyzed by defect testing system (DTS), as well as the DTS reporting log. Then, the designed VCs are leveraged to build models for classifying the reported alarms as actionable and unactionable. Since the unactionable alarms are pruned and only the actionable alarms are to be inspected, the workload of manual inspection is reduced.

Our evaluation results on the four open-source C projects show that the proposed models at variable level are promising and can be a useful approach for automatically classifying the static analysis alarms. Then, we perform a baseline comparison experiment between semantic metrics and traditional metrics, evaluating the effectiveness of our proposed model with semantic metrics in defect identification. Thus, our proposed approach can be applied for automated defect identification when new alarms are reported by static analysis tools. Additionally, we use single attribute evaluator with *Ranker* in Weka to rank the relevance of each VC individually and return an ordered list of ranked VCs.

In the future, we will extend our automated defect identification approach for more defect patterns. Moreover, designing a set of artifact characteristics that may be shared enough for the majority of defect patterns is considered to be of great significance for our work. As is not mentioned in this paper, however, there is a major challenge in raising the accuracy of cross-project defect identification, namely using a given project to train a model to identify the defects from another project without manual inspection. We also plan to leverage our model combined with transfer learning methods to automatically identify defects across projects, which would be promising to increase the accuracy of cross-project defect identification.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Nos. 61702044 and U1736110) and the National Key Research and Development Program of China (No. 2016YFF0204002).

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Arcuri A, Briand LC (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd international conference on software engineering, pp 1–10

Ayewah N, Pugh W (2010) The google findbugs fixit. In: Proceedings of the 19th international symposium on software testing and analysis, pp 241–252

Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou YQ (2007) Evaluating static analysis defect warnings on production software. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, pp 1–8

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: a large-scale evaluation in open source software. In: IEEE 23rd international conference on software analysis, evolution, and reengineering, pp 470–481

Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, McPeak S, Engler DR (2010) A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM 53(2):66–75

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th international conference on software engineering, pp 480–490

Bush WR, Pincus JD, Sielaff DJ (2000) A static analyzer for finding dynamic programming errors. Softw Pract Exp 30(7):775–802

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 332–343

Dietterich TG (2000) Ensemble methods in machine learning. In: Proceedings of the 1st international workshop on multiple classifier systems, pp 1–15

Dillig I, Dillig T, Aiken A (2012) Automated error diagnosis using abductive inference. In: Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, pp 181–192

Flynn L, Snavely W, Svoboda D, VanHoudnos NM, Qin R, Burns J, Zubrow D, Stoddard R, Marce-Santurio G (2018) Prioritizing alerts from multiple static analysis tools, using classification models. In: Proceedings of the 1st international workshop on software qualities and their dependencies, pp 13–20

Hanam Q, Tan L, Holmes R, Lam P (2014) Finding patterns in static analysis alerts: improving actionable alert ranking. In: Proceedings of the 11th working conference on mining software repositories, pp 152–161

Heckman SS, Williams LA (2008) On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: Proceedings of the 2nd international symposium on empirical software engineering and measurement, pp 41–50

Heckman SS, Williams LA (2009) A model building process for identifying actionable static analysis alerts. In: Proceedings of the 2nd international conference on software testing verification and validation, pp 161–170

Heckman SS, Williams LA (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf Softw Technol 53(4):363–387

Johnson B, Song Y, Murphy-Hill ER, Bowdidge RW (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 35th international conference on software engineering, pp 672–681

Jung Y, Kim J, Shin J, Yi K (2005) Taming false alarms from a domain-unaware c analyzer by a Bayesian statistical post analysis. In: Proceedings of the 12th international static analysis symposium, pp 203–217

Kim S, Ernst MD (2007a) Prioritizing warning categories by analyzing software history. In: Proceedings of the 4th international workshop on mining software repositories, pp 27–27

Kim S, Ernst MD (2007b) Which warnings should i fix first? In: Proceedings of the 2007 joint meeting on foundations of software engineering, pp 45–54

Kremenek T, Ashcraft K, Yang JF, Engler DR (2004) Correlation exploitation in error ranking. In: Proceedings of the 12th ACM SIGSOFT international symposium on foundations of software engineering, pp 83–93

Kumar R, Nori AV (2013) The economics of static analysis tools. In: Proceedings of the 2013 joint meeting on foundations of software engineering, pp 707–710

Le W, Soffa ML (2010) Path-based fault correlations. In: Proceedings of the 18th ACM SIGSOFT international symposium on foundations of software engineering, pp 307–316

Lee W, Lee W, Yi K (2012) Sound non-statistical clustering of static analysis alarms. In: Proceedings of the 13th international conference on verification, model checking, and abstract interpretation, pp 299–314

Liang GT, Wu L, Wu Q, Wang QX, Xie T, Mei H (2010) Automatic construction of an effective training set for prioritizing static analysis warnings. In: Proceedings of the 25th IEEE/ACM international conference on automated software engineering, pp 93–102

Muske T, Serebrenik A (2016) Survey of approaches for handling static analysis alarms. In: Proceedings of the 16th IEEE international working conference on source code analysis and manipulation, pp 157–166

Muske T, Talluri R, Serebrenik A (2018) Repositioning of static analysis alarms. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 187–197

Podelski A, Schäf M, Wies T (2016) Classifying bugs with interpolants. In: Proceedings of the 10th international conference on tests and proofs, pp 151–168

Quinlan DJ, Vuduc RW, Misherghi G (2007) Techniques for specifying bug patterns. In: Proceedings of the 5th ACM workshop on parallel and distributed systems: testing, analysis, and debugging, pp 27–35

Reynolds ZP, Jayanth AB, Koc U, Porter AA, Raje RR, Hill JH (2017) Identifying and documenting false positive patterns generated by static code analysis tools. In: 4th IEEE/ACM international workshop on software engineering research and industrial practice, pp 55–61

Ruthruff JR, Penix J, Morgenthaler JD, Elbaum SG, Rothermel G (2008) Predicting accurate and actionable static analysis warnings: an experimental approach. In: Proceedings of the 30th international conference on software engineering, pp 341–350

Wang Q, Jin DH, Gong YZ (2013) Null dereference detection via a backward analysis. In: Proceedings of the 20th Asia-Pacific software engineering conference, vol 1, pp 553–558

Williams CC, Hollingsworth JK (2005) Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans Softw Eng 31(6):466–480

Witten IH, Frank E, Hall MA, Pal CJ (2016) Data mining: practical machine learning tools and techniques. Morgan Kaufmann, Burlington

Yang ZH, Gong YZ, Xiao Q, Wang YW (2008) Dts-a software defects testing system. In: Proceedings of the 8th IEEE international working conference on source code analysis and manipulation, pp 269–270

Yi K, Choi H, Kim J, Kim Y (2007) An empirical study on classification methods for alarms from a bug-finding static c analyzer. Inf Process Lett 102(2–3):118–123

Yoon J, Jin M, Jung Y (2014) Reducing false alarms from an industrial-strength static analyzer by SVM. In: Proceedings of the 21st Asia-Pacific software engineering conference, vol 2, pp 3–6

Yuksel U, Sözer H (2013) Automated classification of static code analysis alerts: a case study. In: Proceedings of the 29th IEEE international conference on software maintenance, pp 532–535

Zhang DL, Jin DH, Gong YZ, Zhang HL (2013) Diagnosis-oriented alarm correlations. In: Proceedings of the 20th Asia-Pacific software engineering conference, vol 1, pp 172–179