Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 924923, 8 pages
doi:10.1155/2012/924923

Research Article

Can Faulty Modules Be Predicted by Warning Messages of Static Code Analyzer?

Osamu Mizuno and Michi Nakai

Kyoto Institute of Technology, Matsugasaki Goshokaido-cho, Sakyo-ku, Kyoto 606-8585, Japan

Correspondence should be addressed to Osamu Mizuno, o-mizuno@kit.ac.jp

Received 5 January 2012; Accepted 24 February 2012

Academic Editor: Chin-Yu Huang

Copyright © 2012 O. Mizuno and M. Nakai. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We have proposed a detection method of fault-prone modules based on the spam filtering technique, "Fault-prone filtering." Fault-prone filtering is a method which uses the text classifier (spam filter) to classify source code modules in software. In this study, we propose an extension to use warning messages of a static code analyzer instead of raw source code. Since such warnings include useful information to detect faults, it is expected to improve the accuracy of fault-prone module prediction. From the result of experiment, it is found that warning messages of a static code analyzer are a good source of fault-prone filtering as the original source code. Moreover, it is discovered that it is more effective than the conventional method (that is, without static code analyzer) to raise the coverage rate of actual faulty modules.

1. Introduction

Recently, machine learning approaches have been widely used for fault-proneness detection [1]. We have introduced a text feature-based approach to detect fault-prone modules [2]. In this approach, we extract text features from the frequency information of words in source code modules. In other words, we construct a large metrics set representing the frequency of words in source code modules. Once the text features are obtained, the Bayesian classifier is constructed from text features. In the fault-prone module detection of new modules, we also extract text features from source code modules, and Bayesian model classifies modules into either fault-prone or nonfault-prone. Since less effort or cost needed to collect text feature metrics than other software metrics, it may be applied to software development projects easily.

On the other hand, since this approach accepts any input with text files, the accuracy of prediction could be improved by selecting appropriate input other than raw source code. We then try to find another input but source code. In this study, we use warning messages of a static code analyzer. Among many static code analyzers, we used PMD in this study. By replacing the input of fault-prone filtering from raw source code to warning messages of PMD, we can get the results of prediction by PMD and fault-prone filtering.

The rest of this paper is organized as follows. Section 2 describes the objective of this research. Section 3 shows a brief summary of the fault-prone filtering technique with PMD. In Section 4, the experiments conducted in this study are described. Section 5 discusses the result of the experiments. Finally, Section 6 concludes this study.

2. Objective

2.1. Fault-Prone Module Filtering. The basic idea of fault-prone filtering is inspired by the spam mail filtering. In the spam e-mail filtering, a spam filter first trains both spam and ham e-mail messages from the training data set. Then, an incoming e-mail is classified into either ham or spam by the spam filter.

This framework is based on the fact that spam e-mail usually includes particular patterns of words or sentences. From the viewpoint of source code, a similar situation usually occurs in faulty software modules. That is similar faults may occur in similar contexts. We thus guessed that similar to spam e-mail messages, faulty software modules

have similar patterns of words or sentences. To obtain such features, we adopted a spam filter in fault-prone module prediction.

In other words, we try to introduce a new metric as a fault-prone predictor. The metric is "frequency of particular words." In detail, we do not treat a single word, but use combinations of words for the prediction. Thus, the frequency of a certain length of words is the only metric used in our approach.

From a viewpoint of effort, conventional fault-prone detection techniques require relatively much effort for application because they have to measure various metrics. Of course, metrics are useful for understanding the property of source code quantitatively. However, measuring metrics usually needs extra effort and translating the values of metrics into meaningful result also needs additional effort. Thus, easy-to-use technique that does not require much effort will be useful in software development.

We then try to apply a spam filter to identification of fault-prone modules. We named this approach as "fault-prone filtering." That is, a learner first trains both faulty and nonfaulty modules. Then, a new module can be classified into fault-prone or notfault-prone using a classifier. In this study, we define a software module as a Java class file.

Essentially, the fault-prone filtering does the text classification on the source codes. Of course, the text classification can be applied to the text information other than the source codes. We guessed that there is the other input for the text classification to achieve higher prediction accuracy. We then started seeking such information.

2.2. Static Code Analysis. The static code analysis is a method of analyzing without actually running software and finding the problem and faults in a software. By analyzing a source code structurally, we can find potential faults, violation of coding conventions, and so on. The static code analysis thus can assure the safety of software, reliability, and quality. It also reduces the cost of maintenance. In recent years, the importance of static code analysis has been emerging since finding potential faults or security hole is required at an early stage of the development. There are many kinds of tools for the static code analysis available [3]. Among them, we used the PMD (the meaning of PMD is not determined. "We have been trying to find the meaning of the letters PMD—because frankly, we do not really know. We just think the letters sound good together" [4]), since it can be applicable to the source code directly.

The PMD is one of static code analysis tools [5]. It is an open-source software and written in Java, and it is used for analyzing programs written in Java. PMD can find the code pieces that may cause the potential faults such as an unused variable and an empty catch block by analyzing the source code of Java. To do so, PMD has a variety of rule sets. According to the rule sets to be used, a broad range of purposes from the inspection of coding conventions to find potential faults can be used.

2.3. Characteristics of the Warning Messages of the Static Code Analyzer. Warning messages of a static code analyzer include rich information about potential faults in source codes. Figure 1 shows an example of warning messages. Usually, the number of warning messages generated by the static code analyzer becomes large in proportion to the length of source code. Since most of the messages are not harmful or trivial, warning messages are often ignored. It can be considered that these warning messages are quality aspects of the source code. Thus, we consider that the warning messages have less noise for fault-prone module prediction.

As mentioned in Section 2.1, applying the text information to the text classifier is an easy task. We thus implement the fault-prone filtering technique to use the warning messages of the static code analyzer. We then conduct experiments to confirm the effects of the warning messages to the performance of the fault-prone filtering approach.

2.4. Research Questions. In this study, we aim at answering the following research questions:

RQ1: "can fault-prone modules be predicted by applying a text filter to the warning messages of a static code analyzer?"

RQ2: "if RQ1 is true, is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?"

RQ1 tries to find a possibility to apply the warning messages to the fault-prone filtering technique. RQ2 investigates the prediction performance.

3. Fault-Prone Filtering with PMD

3.1. Applying PMD to Source Code. We used 10 rule sets of PMD in a standard rule sets: Basic, Braces, Code Size, Coupling, Design, Naming, Optimizations, Strict Exception, Strings, and Unused Code. These rule sets are frequently used for investigation of the quality of software. We apply PMD with 10 rule sets to all source code modules and get warning messages of PMD.

3.2. Classification Techniques. In this study, we used CRM114 (the controllable regex mutilator) spam filtering software [6] for its versatility and accuracy. Since CRM114 is implemented as a language to classify text files for general purpose, applying source code modules is easy. Furthermore, the classification techniques implemented in CRM114 are based mainly on Markov random field model instead of the naive Bayesian classifier.

In this experiment, we used the orthogonal sparse bigrams Markov model built in CRM114.

Orthogonal Sparse Bigrams Markov model (OSB)

Basically, CRM114 uses sparse binary polynomial Hash Markov model (SBPH). It is an extension of the Bayesian classification, and it maps features in the input text into a markov random field [7]. In this model, tokens are constructed from combinations of n words (n-grams) in a text file. Tokens are then

The class "ISynchronizerTest" has a Cyclomatic Complexity of 8 (Highest = 32).
This class has too many methods, consider refactoring it.
Avoid excessively long variable names like NUMBER_OF_PARTNERS
The field name indicates a constant but its modifiers do not
Variables should start with a lowercase character
Variables that are not final should not contain underscores (except for
underscores in standard prefix/suffix).
Document empty constructor
Parameter "name" is not assigned and could be declared final
Avoid variables with short names like b1
Avoid variables with short names like b2
Parameter "b1" is not assigned and could be declared final
Parameter "b2" is not assigned and could be declared final
Parameter "message" is not assigned and could be declared final
Avoid using for statements without curly braces
Local variable "body" could be declared final
Parameter "monitor" is not assigned and could be declared final
Parameter "resource" is not assigned and could be declared final
Avoid using for statements without curly braces
...

Figure 1: A part of warning messages by PMD from a source code module of Eclipse.

mapped into a Markov random field to calculate the probability.

OSB is a simplified version of SBPH. It considers tokens as combinations of exactly 2 words created in the SBPH model. This simplification decreases both memory consumption of learning and time of classification. Furthermore, it is reported that OSB usually achieves higher accuracy than a simple word tokenization [8].

3.3. Tokenization of Inputs. In order to perform fault-prone filtering approach, inputs of fault-prone filter must be tokenized. In this study, in order to use the warning messages of PMD as input of filtering, the messages need to be tokenized. Warning messages of PMD contains English text in natural language and a part of Java code. In order to separate them, we classified them into the following kind of strings:

(i) strings that consist of alphabets and numbers;

(ii) all kinds of brackets, semicolons, commas;

(iii) operators of Java and dot;

(iv) other strings (natural language message).

Furthermore, warning messages of PMD have file names and line numbers on the top of each line. In usual, they provide useful information for debug, but for learning and classification, they may mislead the learning of faulty modules. For example, once we learn a line number of a faulty module, the same line number of the other file is wrongly considered as faulty token.

3.4. Example of Filtering. Here, we explain briefly how these classifiers work. We will show how to tokenize and classify the faulty modules in our filtering approach.

```
1:  if  x
2:  if      ==
3:  if          1
4:  if              return
```

Figure 2: Example of tokens for OSB using the source code.

```
1:  underscores  in
2:  underscores     standard
3:  underscores         prefix/suffix)
```

Figure 3: Example of tokens for OSB using the warning messages.

3.4.1. Tokenization. In OSB, tokens are generated so that these tokens include exactly 2 words. For example, a sentence "if (x == 1) return;" is tokenized as shown in Figure 2 By definition, the number of tokens drastically decreases compared to SBPH. As for the warning messages, an example of a sentence "underscores in standard prefix/suffix)." is shown in Figure 3

3.4.2. Classification. Let T_{FP} and T_{NFP} be sets of tokens included in the fault-prone (FP) and the nonfault-prone (NFP) corpuses, respectively. The probability of fault-proneness is equivalent to the probability that a given set of tokens T_x is included in either T_{FP} or T_{NFP}. In OSB, the probability that a new module m_{new} is faulty, $P(T_{\mathrm{FP}}|T_{m_{\mathrm{new}}})$, with a given set of token $T_{m_{\mathrm{new}}}$ from a new source code module m_{new} is calculated by the following Bayesian formula:

$$\frac{P(T_{m_{\mathrm{new}}}T_{\mathrm{FP}})P(T_{\mathrm{FP}})}{P(T_{m_{\mathrm{new}}}T_{\mathrm{FP}})P(T_{\mathrm{FP}})+P(T_{m_{\mathrm{new}}}T_{\mathrm{NFP}})P(T_{\mathrm{NFP}})}. \quad (1)$$

Table 1: Target project: Eclipse BIRT plugin.

Name	Eclipse BIRT plugin
Language	Java
Revision control	cvs
Type of faults	Bugs
Status of faults	Resolved; Verified; closed
Resolution of faults	Fixed
Severity	Blocker; critical; major; normal
Priority of faults	All
Total number of faults	4708

Table 2: The number of modules in Eclipse BIRT.

	Number of modules (files)
Nonfaulty	42,503
Faulty	27,641
Total	70,144

Intuitively speaking, this probability denotes that the new code is classified into FP. According to $P(T_{\mathrm{FP}} \mid T_{m_{\mathrm{new}}})$ and predefined threshold t_{FP}, classification is performed.

4. Experiment

4.1. The Outline of the Experiment. In this experiment, warning messages of PMD are used for fault-prone filtering as an input instead of a source code module, and Fault-prone module is predicted. And it is the purpose to evaluate the predictive accuracy of the proposed method. Therefore, two experiments using raw source code modules and the warning messages by the PMD as inputs are conducted. We then compare these results to each other.

4.2. Target Project. In this experiment, we use the source code module of an open source project, Eclipse BIRT (business intelligence and reporting tools). The source code module is obtained from this project by the SZZ (Śliwerski et al.) algorithm [9]. The summary of Eclipse BIRT project is shown in Table 1. All software modules in this project are used for both learning and classification by the procedure called training only errors (TOE). The number of modules is shown in Table 2.

4.3. Procedure of Filtering (Training on Errors). Experiment 1 performs the original fault-prone module prediction using the raw source code and OSB classifier by the following procedures:

(1) apply the FP classifier to a newly created software module (say, method in Java, function in C, and so on), M_i, and obtain the probability to be fault-prone;

(2) by the predetermined threshold t_{FP} $(0 < t_{\mathrm{FP}} < 1)$, classify the module M_i into FP or NFP;

(3) when the actual fault-proneness of M_i is revealed by fault report, investigate whether the predicted result for M_i was correct or not;

(4) if the predicted result was correct, go to step 1; otherwise, apply FP trainer to M_i to learn actual fault-proneness and go to step 1.

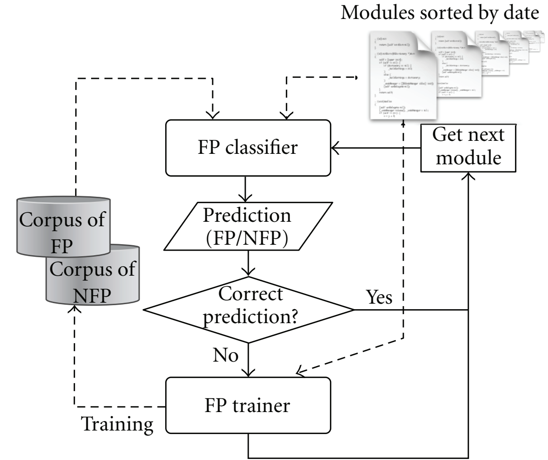

Figure 4: Outline of fault-prone filtering by training on errors.

This procedure is called “training on errors (TOE)” procedure because training process is invoked only when classification errors happen. The TOE procedure is quite similar to actual classification procedure in practice. For example, in actual e-mail filtering, e-mail messages are classified when they arrive. If some of them are misclassified, actual results (spam or nonspam) should be trained.

Figure 4 shows an outline of this approach. At this point, we consider that the fault-prone filtering can be applied to the sets of software modules which are developed in the same (or similar) project.

Experiment 2 is an extension of Experiment 1 by appending additional steps as the first step as follows:

(1) obtain warning messages W_i of PMD by applying PMD to a newly created software module M_i;

(2) apply the FP classifier to the warning messages, W_i, and obtain the probability to be fault-prone;

(3) by the predetermined threshold t_{FP} $(0 < t_{\mathrm{FP}} < 1)$, classify the warning messages W_i into FP or NFP;

(4) when the actual fault-proneness of M_i is revealed by fault report, investigate whether the predicted result for W_i was correct or not;

(5) if the predicted result was correct, go to step (1); otherwise, apply FP trainer to W_i to learn actual fault-proneness and go to step (1).

Table 3: Classification result matrix.

		Prediction	
		Nonfault-prone	Fault-prone
Actual	Nonfaulty	True negative (tn)	False positive (fp)
	Faulty	False negative (fn)	True positive (tp)

4.4. Procedure of TOE Experiment. In the experiment, we have to simulate actual TOE procedure in the experimental environment. To do so, we first prepare a list of all modules found in Section 4.2. The list is sorted by the last modified date (d_i) of each module so that the first element of the list is the oldest module. We then start simulated experiment in the procedure shown in Algorithm 1. During the simulation, modules are classified by the order of date. If the predicted result s_i^p differs from actual status s_i^a, the training procedure is invoked.

4.5. Evaluation Measures. Table 3 shows a classification result matrix. True negative (tn) shows the number of modules that are classified as nonfault-prone, and are actually nonfaulty. False positive (fp) shows the number of modules that are classified as fault-prone, but are actually nonfaulty. On the contrary, false negative shows the number of modules that are classified as nonfault-prone, but are actually faulty. Finally, true positive shows the number of modules that are classified as fault-prone which are actually faulty.

In order to evaluate the results, we prepare two measures: recall and precision. Recall is the ratio of modules correctly classified as fault-prone to the number of entire faulty modules Recall is defined as $tp/(tp + fn)$. Precision is the ratio of modules correctly classified as fault-prone to the number of entire modules classified fault-prone. Precision is defined as $tp/(tp+fp)$. Accuracy is the ratio of correctly classified modules to the entire modules. Accuracy is defined as $(tp + tn)/(tn + tp + fp + fn)$. Since recall and precision are in the trade-off, F_1-measure is used to combine recall and precision [10]. F_1-measure is defined as $(2\times\text{recall}\times\text{precision})/(\text{recall}+\text{precision})$. In this definition, recall and precision are evenly weighed.

From the viewpoint of the quality assurance, it is recommended to achieve higher recall, since the coverage of actual faults is of importance. On the other hand, from the viewpoint of the project management, it is recommended to focus on the precision, since the cost of the software unit test is deeply related to the number of modules to be tested. In this study, we mainly focus on the recall from the viewpoint of the quality assurance.

4.6. Result of Experiments. Tables 4 and 5 show the result of experiment using the original approach without PMD and the approach with PMD, respectively. Table 6 summarizes the evaluation measures for these experiments.

From Table 6, we can see that the approach with PMD has almost the same capability to predict fault-prone modules as the approach without PMD. For example, F_1 for the approach without PMD is 0.779, and for the approach with PMD is 0.710. The result shows that the original approach without PMD is relatively better than the approach with PMD in precision, accuracy, and F_1 measures. The recall of the approach with PMD is better than the approach without PMD.

Table 4: Result of prediction in Experiment 1 (without PMD).

		Prediction	
		Nonfault-prone	Fault-prone
Actual	Nonfaulty	30,521	11,982
	Faulty	2,360	25,281
Total		32,821	37,263

Table 5: Result of prediction in Experiment 2 (with PMD).

		Prediction	
		Nonfault-prone	Fault-prone
Actual	Nonfaulty	22,982	19,521
	Faulty	1,674	25,967
Total		24,656	45,488

Table 6: Evaluation measures of the results of experiments.

	Precision	Recall	Accuracy	F_1
Experiment 1 (without PMD)	0.678	0.915	0.796	0.779
Experiment 2 (with PMD)	0.571	0.939	0.698	0.710

Figures 5 and 6 show the result of TOE history for the approaches without and with PMD, respectively. From this graph, we can see that evaluation measures first to decrease at the beginning of TOE procedure, then increase and become stable after learning and classification of 15,000 modules.

5. Discussions

At first, we discuss the advantage of the approach with PMD. From Table 6, we can see that the result of Experiment 2 has higher recall and lower precision than that of Experiment 1. Generally speaking, the recall is an important measure for the fault-prone module prediction because it implies how many actual faults can be detected by the prediction. Therefore, higher recall can be an advantage of the approach with PMD. However, the difference of the recalls between two experiments is rather small.

When we focus on the graphs of TOE histories shown in Figures 5 and 6, the difference between two experiments can be seen clearly. The transition of recall in Experiment 2 keeps higher than that of Experiment 1 from an early stage of the experiment. That is the recall of Experiment 2 reaches 0.90 at 10,000 modules learning. From this fact, we can say that the approach with PMD is efficient especially at an early stage of development. It can be considered as another advantage of the approach with PMD.

We discuss the reasons of the result that the approach with PMD does not shows a good evaluation measures at

t_{FP} : Threshold of probability to determine FP and NFP
s_i^p : Predicted fault status (FP or NFP) of M_i
for each M_i in list of modules sorted by d_i's
  $prob$ = fpclassify(M_i)
  if $prob > t_{FP}$ then s_i^p = FP
    else s_i^p = NFP
  endif
  if $s_i^a \neq s_i^p$ then fptrain(M_i, s_i^a)
  endif
endfor
fpclassify(M) :
  if Experiment 1 then
    Generate a set of tokens T_M from source code M.
    Calculate probability $P(T_{FP} \mid T_M)$
      using corpuses T_{FP} and T_{NFP}.
    Return $P(T_{FP} \mid T_M)$.
  if Experiment 2 then
    Generate a set of tokens T_W
      from warning messages W
      by applying PMD to the source code M.
    Calculate probability $P(T_{FP} \mid T_W)$
      using corpuses T_{FP} and T_{NFP}.
    Return $P(T_{FP} \mid T_W)$
fptrain(M, s^a) :
  if Experiment 1 then
    Generate a set of tokens T_M from M.
    Store tokens T_M to the corpus Ts^a.
  if Experiment 2 then
    Generate a set of tokens T_W from W
      by applying PMD to M.
    Store tokens T_W to the corpus Ts^a.

Algorithm 1: Procedure of TOE experiment.

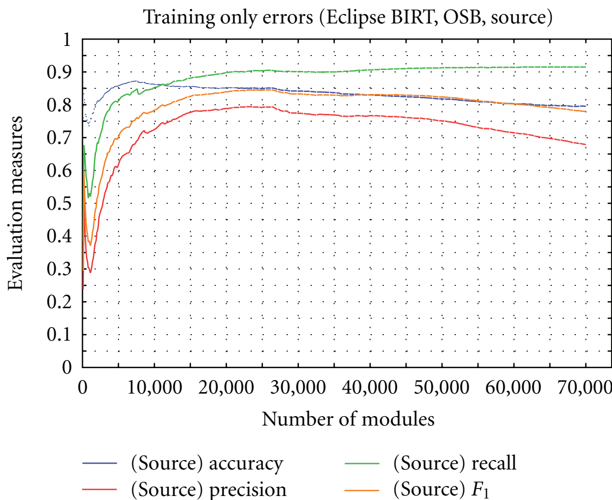

Figure 5: History of training on errors procedure in Experiment 1 (without PMD).

the end of the experiment. First, the selection of rule sets used in PMD may affect the result of experiment. Although we used 10 rule sets according to the past study, the selection of rule sets should be considered more carefully. For future research, we will investigate the effects of rule set selection to the accuracy of fault-prone filtering. Second, we need to apply this approach to more projects. We have conducted experiments on Eclipse BIRT.

Here, we investigate the details of our prediction. Table 7 shows a part of the probabilities for each token in the corpus for faulty modules. This table shows tokens with highest probabilities. The probability $P(T_x|T_{FP})$ shows the conditional probability that a token T_x exists in the faulty corpus. Although these probabilities do not mean immediately that these tokens make a module fault-prone, we guess that the investigation of these probabilities helps improving accuracy.

We can see that specific identifier such as “copyInstance” and specific literals such as “994,” “654,” and “715” appear frequently. It can be guessed that these literals denote line number in a particular source code. These literals are effective to predict the fault-proneness of the specific source code modules, but it can be a noise for the most other modules. In order to improve the overall accuracy of the classifier, eliminating literals that describe a specific source code should be taken into account.

Finally, we answer the research questions here. We have the following research questions in Section 2.4.

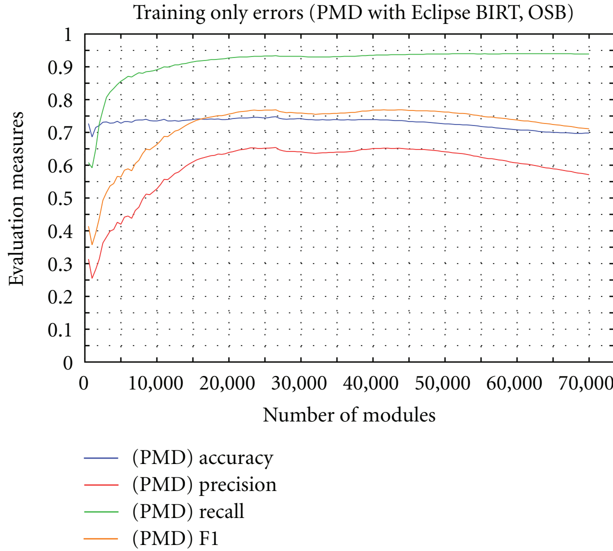

Figure 6: History of training on errors procedure in Experiment 2 (with PMD).

Table 7: Probabilities for each token in the corpus for faulty modules. The “□” denotes any words.

$P(T_x \mid T_{FP})$	T_x
0.56229	line 654
0.56229	715 Local
0.56229	“copyInstance” has
0.56229	654 Local
0.56229	instanceof The
0.56229	method copyInstance ()
0.56229	255 Parameter
0.56229	method “copyInstance”
0.56229	489 Avoid
0.56229	line 715
0.56229	copyInstance () has
0.56229	line 994
0.56229	line 474
0.56229	994 The
0.56215	715 □ variable
0.56215	blocks □ “pageNumber”
0.56215	on □ 994
0.56215	bb □ instantiating
0.56215	on □ 654
0.56215	The □ “copyInstance”
0.56215	654 □ variable
0.56215	994 □ String
0.56215	The □ copyInstance ()
0.56215	“copyInstance” □ a
0.56215	on □ 715
0.56215	on □ 474
0.56215	300 □ variables
0.56215	copyInstance () □ an

RQ1: “can fault-prone modules be predicted by applying a text filter to the warning messages of a static code analyzer?”

For this question, we can answer “yes” from the results in Table 5 and Table 6. It is obvious that the approach with PMD has prediction capability of the fault-prone modules at a certain degree.

RQ2: “if RQ1 is true, is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?”

For this question, we can say that the recall of the approach with PMD becomes higher and more stable during the development than the approach without PMD as shown in Table 6 and Figures 5 and 6. From the viewpoint of the quality assurance, it is a preferred property. We then conclude that the proposed approach has better performance to assure the software quality.

6. Conclusion

In this paper, we proposed an approach to predict fault-prone modules using warning messages of PMD and a text filtering technique. For the analysis, we stated two research questions: “can fault-prone modules be predicted by applying a text filter to the warning messages of static code analyzer?” and “is the performance of the fault-prone filtering becomes better with the warning messages of a static code analyzer?” We tried to answer this question by conducting experiments on the open source software. The results of experiments show that the answer to the first question is “yes.” As for the second question, we can find that the recall becomes better than the original approach.

Future work includes investigating which parts of warning messages are really effective for fault-prone module prediction. Selection of rule sets of PMD is an interesting future research.

References

[1] C. Catal and B. Diri, “A systematic review of software fault prediction studies,” *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[2] O. Mizuno and T. Kikuno, “Training on errors experiment to detect fault-prone software modules by spam filter,” in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 405–414, 2007.

[3] N. Rutar, C. B. Almazan, and J. S. Foster, “A comparison of bug finding tools for java,” in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 245–256, IEEE Computer Society, Washington, DC, USA, 2004.

[4] T. Copeland, PMD—What does “PMD” mean?, http://pmd.sourceforge.net/meaning.html .

[5] T. Copeland, *PMD Applied*, Centennial Books, Arexandria, Va, USA, 2005.

[6] W. S. Yerazunis, *CRM114—the Controllable Regex Mutilator*, http://crm114.sourceforge.net/.

[7] S. Chhabra, W. S. Yerazunis, and C. Siefkes, "Spam filtering using a markov random field model with variable weighting schemas," in *Proceedings of the 4th IEEE International Conference on Data Mining, (ICDM '04)*, pp. 347–350, Riverside, Calif, USA, November 2004.

[8] C. Siefkes, F. Assis, S. Chhabra, and W. S. Yerazunis, "Combining winnow and orthogonal sparse bigrams for incremental spam filtering," in *Proceedings of the Conference on Machine Learning/European Conference on Principles and Practice of Knowledge Discovery in Databases (ECML PKDD '04)*, 2004.

[9] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? (on Fridays)," in *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pp. 24–28, St. Louis, Mo, USA, May 2005.

[10] C. J. van Rijsbergen, *Information Retrieval*, Butterworth, Boston, Mass, USA, 2nd edition, 1979.