An Extension of Fault-Prone Filtering Using Precise Training and a Dynamic Threshold

Hideaki Hata, Osamu Mizuno, Tohru Kikuno
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita
Osaka 565-0871, Japan
{h-hata, o-mizuno, kikuno}@ist.osaka-u.ac.jp

ABSTRACT

Fault-prone module detection in source code is important for assurance of software quality. Most previous fault-prone detection approaches have been based on software metrics. Such approaches, however, have difficulties in collecting the metrics and in constructing mathematical models based on the metrics.

To mitigate such difficulties, we have proposed a novel approach for detecting fault-prone modules using a spam-filtering technique, named Fault-Prone Filtering. In our approach, fault-prone modules are detected in such a way that the source code modules are considered as text files and are applied to the spam filter directly.

In practice, we use the training only errors procedure and apply this procedure to fault-prone. Since no pre-training is required, this procedure can be applied to an actual development field immediately.

This paper describes an extension of the training only errors procedures. We introduce a precise unit of training, "modified lines of code," instead of methods. In addition, we introduce the dynamic threshold for classification. The result of the experiment shows that our extension leads to twice the precision with about the same recall, and improves 15% on the best F_1 measurement.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Measurement, Reliability

Keywords

spam filter, fault-prone modules, text mining

1. INTRODUCTION

Fault-prone code detection is one of the most traditional and important areas in software engineering. Once fault-prone modules are detected at an early stage of the development, developers can take more careful notice of the detected modules. Furthermore, keeping track of fault-prone modules is useful in order to prevent injecting additional faults in them.

Various studies have been done in the detection of the fault-prone modules [3,4,6,8,10,11,14]. Most of these studies used some kind of software metrics, such as program complexity, size of modules, object-oriented metrics, and so on, and constructed mathematical models to calculate fault-proneness.

We have introduced a spam filter based approach, named Fault-Prone Filtering, to detect fault-prone modules [12]. The spam filter is one of the most widely used in text mining applications. Recently, since the usefulness of Bayesian theory for spam filtering has been recognized, most spam filtering tools implement Bayesian theories. Consequently, the accuracy of spam detection has been improving drastically.

Inspired by the spam filtering technique, we tried to apply text-mining techniques to fault-prone detection. In fault-prone filtering, we consider a software module as an e-mail message, and assume that all of the software modules belong to either fault-prone(FP) or not-fault-prone(NFP). We also conducted an experiment based on the training only errors(TOE) procedure that can simulate practical situations. In this procedure, software modules are classified in developed order. Only misclassified software modules are used for training of corpuses for further classification. This procedure reduces the time for training and avoids over-training. However, the result of predictions shows that the recall is very high, but the precision is low. Such an unbalanced result is not appropriate for fault-prone prediction.

We thus introduce the following 2 extensions for fault-prone filtering:

Precise training: In our previous approach, the unit of training was a whole software module. However, faulty parts of a software module are usually very small in source code. Thus, much useless training was performed in the previous way. In this study, we introduce more a precise unit of training, "modified lines of code".

Dynamic threshold: To classify modules according to calculated probabilities, we have to determine a threshold of classification. The threshold was pre-determined and not changed in the TOE. However, we guessed that the threshold varies depending on the characteristics of modules. We thus define the way to change the threshold according to the types of modules.

An experiment is prepared to show the effectiveness of the extended approach using the source code repository of 2 open source software developments. Both FP and NFP modules are then col-

MSR'08, May 10-11, 2008, Leipzig, Germany.

lected from the repository. We conducted the TOE procedure using about 100 thousand software modules and classified them into FP or NFP. The result of the experiment shows that the extended approach leads to twice the precision with about the same recall, and improves 15% on the best F_1.

The rest of this paper is organized as follows: Section 2 describes an overview of previous "fault-prone filtering" approaches as well as its extension for better accuracy. The procedure of collecting FP and NFP modules is then described in Section 3. An experiment to show the effectiveness of our extended approach is shown in Section 4. Section 5 discusses the result obtained in the experiment. Section 6 addresses threats to the validity of this study. Finally, Section 7 summarizes this study and also addresses future work.

2. EXTENDED FAULT-PRONE FILTERING

2.1 Fundamental Idea of Fault-Prone Filtering

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

This framework is based on the fact that spam e-mails usually include particular patterns of words or sentences. From the viewpoint of source code, similar situations usually occur in faulty software modules. That is, similar faults may occur in similar contexts. We thus guessed that faulty software modules have similar patterns of words or sentences like spam e-mail messages. In order to grab such features, we adopted a spam filter in fault-prone module prediction.

We then try to apply a spam filter for fault-prone module prediction. We named this approach, "fault-prone filtering". That is, the fault-prone trainer first trains both FP and NFP modules. Then, a new module can be classified into FP or NFP using the fault-prone classifier. To do so, we have to prepare a spam filtering software and sets of FP and NFP modules.

In this study, we used "CRM114" spam filtering software [5]. The reason why we used CRM114 was its versatility and accuracy. Since CRM114 is implemented as a language to classify text files for general purpose, it is easy to apply source code modules.

2.2 Training Only Errors

In order to apply our approach to data from a source code repository, we implemented tools named "FPTrainer" and "FPClassifier" for training and classifying software modules, respectively.

The typical procedure for fault-prone filtering is summarized as follows:

1. Apply FPClassifier to a newly created software module, M_i, and obtain the probability to be fault-prone.
2. By the threshold t_{FP} $(0 < t_{FP} < 1)$, classify module M_i into FP or NFP.
3. When the actual fault-proneness of M_i is revealed by a 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 FPTrainer to M_i to train actual fault-proneness and go to step 1.

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

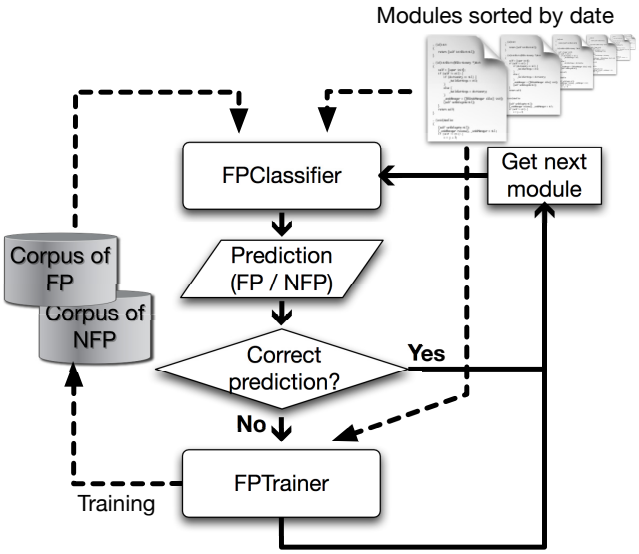


Figure 1: Outline of Fault-Prone Filtering by Training only Errors

Figure 1 shows an outline of this approach. At this point, we consider that fault-prone filtering can be applied to the set of software modules developed in the same (or a similar) project.

As referred to in [2], fault-prone codes could be spread also on different parts in a class file. We thus used class files of a Java program in each revision as a software module.

2.3 Extension to Fault-Prone Filtering

2.3.1 Precise Training

In our previous study [13], the input unit of both FPTrainer and FPClassifier was a method of Java. That was because we treated methods in Java code as software modules. If a method contained bug-introducing changes, we considered it an FP module. A bug-introducing change is a modification that introduces bugs into the source code. However in such an FP module, there may contain "not bug-introducing lines" as well as "bug-introducing lines". An NFP module is considered a module which does not contain "bug-introducing lines," which trained all "not bug-introducing lines" that can be in an actual FP module.

The difference in the corpuses of FP and NFP is the essence of fault-prone filtering. However, the difference between FP and NFP modules is smaller than the difference between spam and ham e-mail messages because of the poor vocabulary in the programming language.

We thus introduce a more precise unit of training, "modified lines of code". Only bug-introducing changes are stored in the FP corpus and only bug-fix changes which resolve bug modifications, are stored in the NFP corpus. Such input of the FPTrainer allows a bigger difference between the corpuses of FP and NFP, because a bug-introducing change must not be in NFP modules, and a bug-fix change must not be in FP modules which are related to the bug.

2.3.2 Dynamic Change of Threshold

The threshold t_{FP} is pre-determined and not be changed in [13].

However, we guessed that the threshold varies depending on the characteristics of modules. We also guessed that more accurate classification may be obtained with the dynamic threshold.

We thus define the way to change the threshold according to the types of modules. The way of changing the threshold is simple as follows:

- If an actual NFP module was misclassified as FP, then the threshold moves higher. That is, more modules are predicted as NFP.
- If an actual FP module was misclassified into NFP, then the threshold moves lower. That is, more modules are predicted as FP.

This is because we can avoid misclassification of NFP if the threshold is sufficiently high. Contrary to that, we can avoid misclassification of FP if the threshold is sufficiently low.

3. EXPERIMENTAL SETTINGS

This section describes the settings of the experiment. To begin, we identify bug-introducing changes and bug-fix changes. These changes are extracted as changed lines we call modified lines, by a version control system, such as CVS. In this paper, we call lines of bug-introducing change "FP lines", and lines of bug-fix change as "FIX lines". A set of FP lines and FIX lines can be extracted after one bug is resolved.

3.1 Finding Modified Lines

First, we have to collect sets of FP lines and FIX lines to train FP and NFP corpuses. Both FP lines and FIX lines are extracted from source code repositories based on an algorithm shown by Sliwerski et al. [15]. The following restriction and assumption exist in this collection method:

Restriction We seek FP lines and FIX lines by examining cvs logs. Therefore, faults that do not appear in the cvs logs cannot be considered. That is, the set of FP modules used in this study is not complete.

Assumption We assume that faults are reported just after they are injected in the software.

Next, we collected the following information from a bug database of a target project such as Bugzilla.

- FLT: A set of faults found in a bug database.
- f_i: Each fault in FLT.
- $date(f_i)$: Date in which a fault f_i is reported.

Here, we consider a line of source code L_j with d_j, where d_j is the last modified date of L_j.

We then start mining a source code repository according to the following algorithm to extract FP lines and FIX lines.

1. For each fault f_i, find a certain revision of class CR_{FixedRev} in which the fault has just been fixed by checking all revision logs.
2. Take the difference with each CR_{FixedRev} and just previous revision of the same class.
3. For each line L_m in a CR_{FixedRev}, L_m is a FIX line if it is changed or added.

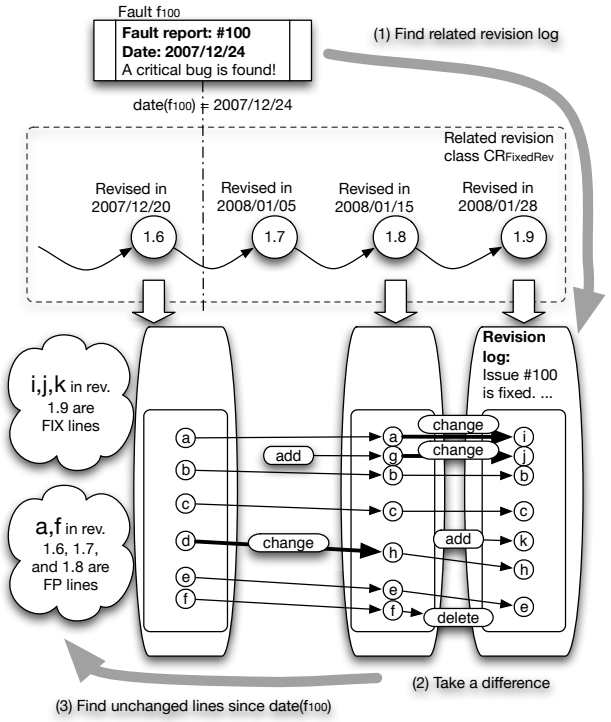


Figure 2: Collection of FP lines and FIX lines

4. For each previous changing or deleting line L_n in the just previous revision class, examine d_n, if $d_n < date(f_i)$, L_n is a FP line.

This algorithm collects a pair of FP lines and FIX lines.

An illustrated example of collecting FP lines and FIX lines is shown in Figure 2. In this example, assume that the class which has the fixed revision classes CR_{FixedRev} has revisions 1.1, 1.2, $\cdots$, 1.9, and revision logs are appended when each revision is committed. First, a fault f_{100} is found on 24th December, 2007. By searching all revision logs, assume that the fixed point is found at revision 1.9 of the class CR_{FixedRev} (Shown as (1) in Figure 2). Then, take the difference between revision 1.8 and 1.9 (Shown as (2) in Figure 2). In the CR_{FixedRev}, lines i and j are changed, and a line k is added, that is they are bug-fix change lines. So these 3 lines are FIX lines.

Lines a, g, and f in revision 1.8 of the class are prospective FP lines since changes or deletions occur in the next revision. In prospective FP lines, by searching revision differences, we find lines that have not been modified since the 24th of December, 2007. This is the purpose of removing prospective FP lines not being related to the f_{100}. After checking the last modified date of each prospective FP line, lines a and f are accepted as FP lines, but line g is not.

We implemented a prototype tool to track bugs in the cvs repository. The inputs of the tool are a cvs repository of the target project and a bug report to track. The outputs of the tool are sets of FP lines and FIX lines.

3.2 Definition of FP and NFP Module

Next, we define FP and NFP modules. We consider a software module M_i with s_i^a, which is the actual fault status (FP or NFP) of M_i. In short, if a module in a certain revision contains FP lines,

l_{FP}: the number of FP lines.
l_{FIX}: the number of FIX lines.

for each M_i
    if $l_{FP} > 0$ in current revision; then
        if $l_{FP} = 0$ in the previous revision; then
            $s_i^a = FP_1$
        else
            $s_i^a = FP_2$
        endif
    else if $l_{FIX} > 0$ then
        $s_i^a = NFP_1$
    else if $l_{FIX} > 0$ in some of previous revisions; then
        $s_i^a = NFP_2$
    else
        $s_i^a = NFP_3$
    endif
endfor

Figure 3: Procedure of dividing FP and NFP status

the module's actual fault status is FP, and we call the module an FP module; otherwise, its actual fault status is NFP and we call the module an NFP module. An important point of this definition is that we do not mention of NFP classes. That is to say, not all NFP modules contain FIX lines.

In the actual fault-prone filtering procedure, we want to train only FP lines into the FP corpus, and train only FIX lines into the NFP corpus.

For TOE, we divide the actual fault-prone status into two types, and the actual not-fault-prone status into three types. First, we divide NFP status into two types depending on whether a module contains FIX lines or not. If a certain revision of one module is an NFP module which does not contain FIX lines, $s_i^a = NFP_3$; this means that this module has not been introduced to bugs and has not been fixed until this revision. Next, we divide status of FP and NFP, which contain FP or FIX lines in each of the two types depending on the timing of inserting these lines. One type is just inserted revision, $s_i^a = FP_1$ or $s_i^a = NFP_1$, and otherwise, $s_i^a = FP_2$ or $s_i^a = NFP_2$.

Here we call a module with $s_i^a = FP_x$ and NFP_y an FP_x module and an NFP_y module, respectively. In the history of a certain module, the module becomes an FP_1 module, then becomes an FP_2 module successively. The FP_1 module becomes a NFP_1 module if bugs in the FP_1 module are fixed. The NFP_1 module becomes an NFP_2 module, successively. As described in subsection 3.1, FP lines in a certain module, which is related to one bug, can be extracted in a sequence of revisions of the modules. However, FIX lines can be extracted in only one revision. This revision is a NFP_1 module, and the following revisions of the module are NFP_2 modules if there are no FP lines. Division of FP status is related to the movement of the threshold.

We decide those types in Figure 3.

3.3 Procedure of TOE

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 in all revisions.

Here, we consider a software module M_i as a tuple of $date_i$ m_i, in addition to s_i^a, where $date_i$ is the committed date of M_i, and m_i is the source code of M_i. As stated in subsection 3.2, if a module M_i contains FP lines, actual fault status s_i^a is FP_1, FP_2 and otherwise, NFP_1, NFP_2, and NFP_3.

The list is sorted by $date_i$ of each module so that the first element of the list is the oldest module. We then start the simulated experiment in the procedure shown in Figure 4. During the simulation, modules are classified in order of date. If the predicted result differs from the actual fault status, that is s_i^p is FP though s_i^a is NFP_1, NFP_2, or NFP_3, and s_i^p is NFP though s_i^a is FP_1, or FP_2, the training procedure is invoked.

As shown in Figure 4, if actual fault status $s_i^a = NFP_3$, none of the lines is trained because there are no FIX lines as the definition of a NFP_3 module. In a pilot survey, after training all lines of NFP_3 modules, we got low recall. Since the volume of the corpus of NFP became much bigger than the volume of FP, modules tend to be classified as NFP. To avoid such misclassification, we try to

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 $date_i$'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)
        if $s_i^a = NFP_3$ & $prob < 1$ then
            $t_{FP} = prob$
        endif
        if $s_i^a = FP_2$ & $prob > 0$ then
            $t_{FP} = prob$
        endif
    endif
endfor

fpclassify(m):
    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 $P(T_{FP} \mid T_m)$.

fptrain(M, s^a):
    if $s^a = FP_1$ or FP_2 then
        Generate a set of tokens T_m from FP lines in M.
        Store tokens T_m to the corpus T_{s^a}.
    else if $s^a = NFP_1$ then
        Generate a set of tokens T_m from FIX lines in M.
        Store tokens T_m to the corpus T_{s^a}.
    else if $s^a = NFP_2$ then
        Generate a set of tokens T_m from FIX lines
            in older revisions of M.
        Store tokens T_m to the corpus T_{s^a}.
    endif
    Return

Figure 4: Procedure of TOE experiment

Table 1: Legend of experimental result

		Prediction	
		NFP	FP
Actual	NFP	N_1	N_2
	FP	N_3	N_4

make precise training. However, by raising the threshold, we expect to reduce misclassification on new NFP_3 modules. Intuitively speaking, we avoid misclassification of NFP_1 and NFP_2 modules, with the NFP corpus, which contains FIX lines, and NFP_3 modules with the raised threshold.

With this raising procedure, the threshold becomes higher and higher. As a result, the number of misclassifications of the FP modules increases. To avoid such a case, we lower the threshold to proper level. If we lower the threshold when both FP_1 and FP_2 modules are misclassified, we guess that the threshold falls down to a level too low. Looking at the revision history of a module, a module becomes an FP_1 module first, then it becomes an FP_2 module. Therefore, the probability of fault-proneness in the FP_2 modules tends to be higher than in that of the FP_1 modules. Therefore, we lower the threshold when FP_2 modules are misclassified.

By the way, some FIX lines can be turned out to be FP lines, that is, fix-on-fix lines. These fix-on-fix lines are first stored in the NFP corpus. After being turned out to be FP lines, these fix-on-fix lines are then stored in the FP corpus. Consequently, the probability of misclassification of a new module that contains FP lines similar to the fix-on-fix lines, which is a FP modules, is getting small. The probability of correct classification as FP is also small because both FP and NFP corpuses contain the same fix-on-fix lines. However, if the module is misclassified as NFP, the module's FP lines, which are similar to the fix-on-fix lines, are stored in the FP corpus with the TOE procedure. As a result, the probability of correct classification of new modules that contain FP lines similar to the fix-on-fix lines is expected to be larger.

3.4 Evaluation Measurements

For the evaluation of the experiments, we define several evaluation measurements. Table 1 shows a legend of tables for experimental result. In Table 1, N_1 shows the number of modules that are predicted as NFP, and are actually NFP. N_2 shows the number of modules that are predicted as FP but are actually NFP. Usually, N_2 is called a false positive. On the contrary, N_3 shows the number of modules that are predicted as NFP, but are actually FP. N_3 is called a false negative. Finally, N_4 shows the number of modules that are predicted as FP and are actually FP.

For evaluation purposes, we used two measurements: recall and precision. Recall is the ratio of modules correctly predicted as FP to the number of entire modules actually FP. Recall is defined as follows:

$$\text{Recall} = \frac{N_4}{N_3 + N_4}$$

Intuitively speaking, recall implies the reliability of the approach because a large recall denotes that actual FP modules can be covered by the predicted FP modules.

Precision is the ratio of modules correctly predicted as FP to the number of entire modules predicted as FP. Precision is defined as follows:

$$\text{Precision} = \frac{N_4}{N_2 + N_4}$$

Intuitively speaking, precision implies the cost of the approach because a small precision requires much effort to find actual FP modules from the predicted FP modules.

Table 2: Target projects

Name	BIRT	EMF
Language	Java	
Revision control	cvs	
Size of entire repository	610 MB	1.07 GB
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	7,788	4,821

F_1 is used to combine recall and precision. F_1 is defined as follows:

$$F_1 = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$$

In this definition, recall and precision are evenly weighted.

The rates of Type I error and Type II error are defined as follows:

$$\text{Type I error rate} = \frac{N_2}{N_1 + N_2 + N_3 + N_4}$$

$$\text{Type II error rate} = \frac{N_3}{N_1 + N_2 + N_3 + N_4}$$

4. EXPERIMENT

4.1 Target Project

For the experiment, we selected open source software projects that can track faults. For this reason, we targeted 2 projects: Eclipse BIRT plugin(BIRT) and Eclipse modeling framework(EMF) [7]. Table 2 shows the context of each target project. These projects are constructed in Java language, and revisions are maintained by concurrent version control system (cvs). The source repository of cvs used in this study was uploaded once on the eclipse project Web site, and was obtained on the 1st of December, 2007.

We also obtained a fault report from the bug database of the eclipse project. We extracted faults from the bug database (Bugzilla) under the following conditions: The type of these faults is "bugs", therefore these faults do not include any enhancements or functional patches. The status of faults is either "resolved", "verified", or "closed", and the resolution of faults is "fixed". This means that the collected faults have already been fixed and been resolved, and thus fixed revision should be included in the entire repository. The severity of the faults was either blocker, critical, major, or normal. We did not use trivial bugs in this study.

Using our FP module collection tool, we collected both FP and NFP modules from these 2 projects. The result of the collection is shown in Table 3.

4.2 Result of Experiment

In the experiment, we conducted the following 4 experiments:

e_1: TOE procedure with static threshold ($t_{FP} = 0.50$) applied to the BIRT project.

e_2: TOE procedure with static threshold ($t_{FP} = 0.90$) applied to the BIRT project.

Table 3: Result of FP module collection for target projects

Name		BIRT	EMF
# of faults found in cvs log		2758 (35% of total)	629 (13% of total)
# of FP lines		102,290	43,155
# of NFP lines		172,359	73,486
# of modules		49,726	99,612
FP	# of FP_1 modules	2,190	1,853
	# of FP_2 modules	15,441	10,087
	subtotal	17,631	32,095
NFP	# of NFP_1 modules	3,158	2,495
	# of NFP_2 modules	5,120	13,443
	# of NFP_3 modules	23,817	83,880
	subtotal	11,940	99,818

Table 4: Result in each experiment of BIRT

Experiment	e_1	e_2	e_3
Recall	0.844	0.820	0.798
Precision	0.515	0.599	0.639
F-measure	0.640	0.692	0.710
Type I error rate	0.282	0.194	0.160
Type II error rate	0.055	0.064	0.072

e_3: TOE procedure with dynamic threshold applied to the BIRT project.

e_4: TOE procedure with dynamic threshold applied to the EMF project.

By comparing experiments e_1, e_2, and e_3, we try to show the effectiveness of the dynamic threshold proposed in this study. Furthermore, we try to show the generality of our approach by comparing experiments e_3 and e_4.

Figure 5 shows transitions of evaluation measurements for each experiment. In these graphs, the x-axis shows all software modules sorted by dates and the smaller number shows older modules. The y-axis shows rates of recall, precision, etc.

Figure 5 (a) shows that high recall is achieved but precision is low in experiment e_1. This low precision implies that modules tend to be predicted as FP. Figure 5 (b) shows that precision is improved with a higher static threshold in experiment e_2. This improved precision is because the FPClassifier predicted FP modules more strictly.

In the case of experiment e_3 in Figure 5 (c), precision is more improved and the type I error rate is held down. This further improvement implies that the number of misclassifications of actual NFP modules has decreased. Table 4 shows the results of evaluation measurements at the end of the TOE procedure for 3 experiments, e_1, e_2, and e_3.

Table 5 shows the detailed classification results for two experiments e_3 and e_4 at the final point of the TOE application. As mentioned before, we categorized actual NFP and FP modules into three and two categories, respectively. Table 5 shows these categories, too. We can see that the number of NFP_3 and FP_2 modules are larger than other categories.

In more detail, we can see that the NFP_1 and FP_1 modules tend to be misclassified and the NFP_2, NFP_3, and FP_2 modules tend to be classified correctly. For example in Table 5 (a), the number of actual NFP_1 modules was 3,158. However, 1,336 modules are wrongly predicted as FP and the error rate is 42.3%. On the other hand, the number of actual NFP_2 modules was 5,120. Among them, 1,044 modules are wrongly predicted as FP and thus the error rate is 20.4%. Since the NFP_1 module is a module in which a bug has just been corrected, the accuracy of prediction becomes lower. On the other hand, the NFP_2 module is a module in which some bugs have been corrected in the past and the information of the bugs has already been trained to the corpuses. For this reason, the NFP_2 modules tend to be predicted correctly.

Table 5: Final classification results in the TOE experiment

(a) Result of experiment e_3

			Prediction		
			NFP	FP	Total
Actual	NFP	NFP_1	1,822	1,336	3,158
		NFP_2	4,076	1,044	5,120
		NFP_3	18,236	5,581	23,817
		subtotal	24,134	7,961	32,095
	FP	FP_1	1,201	989	2,190
		FP_2	2,357	13,084	15,441
		subtotal	3,558	14,073	17,631

(b) Result of experiment e_4

			Prediction		
			NFP	FP	Total
Actual	NFP	NFP_1	1,899	596	2,495
		NFP_2	12,595	848	13,443
		NFP_3	76,764	7,116	83,880
		subtotal	91,258	8,560	99,818
	FP	FP_1	1,100	753	1,853
		FP_2	2,486	7,601	10,087
		subtotal	3,586	8,354	11,940

Table 6: Evaluation measurements in TOE experiment

Experiment	e_3	e_4
Recall	0.798	0.700
Precision	0.639	0.494
F_1	0.710	0.580
Type I error rate	0.160	0.072
Type II error rate	0.077	0.032

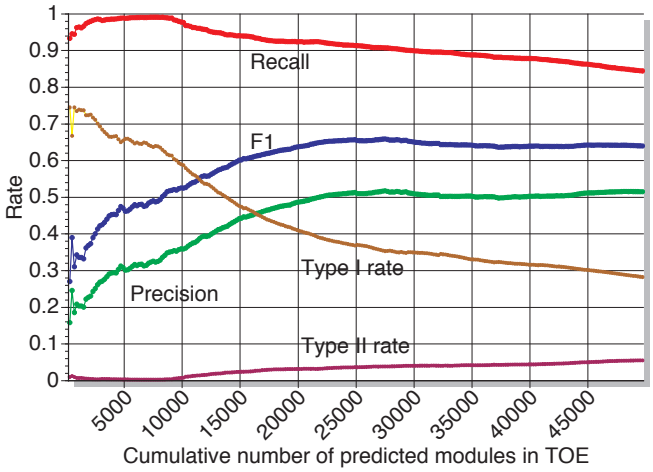
The evaluation measurements for each experiment are calculated in Table 6. As shown in Table 6, we obtained better precision and recall in the experiment e_3 than that in e_4.

5. DISCUSSION

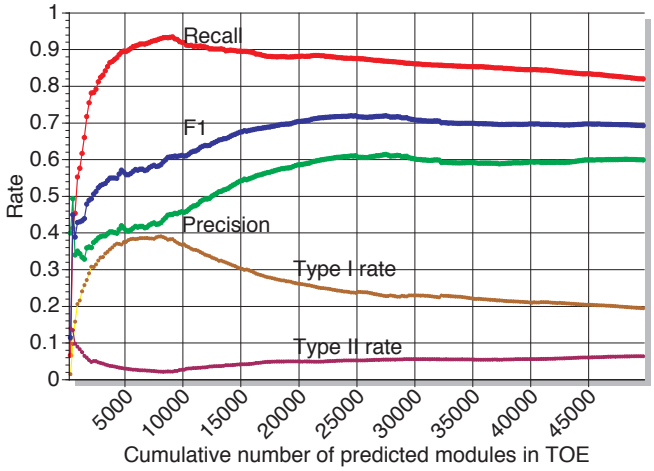
5.1 Transition of Evaluation Measurements

In Figure 5, it is observed that measurements are not good at an early stage of the development.In actuality, type I error rate is relatively high at an early stage of development and thus precision is low. After the training of sufficient modules, type I error rate becomes lower and thus precision becomes higher. This fact indicates that the TOE procedure works well after a certain training period of the development.

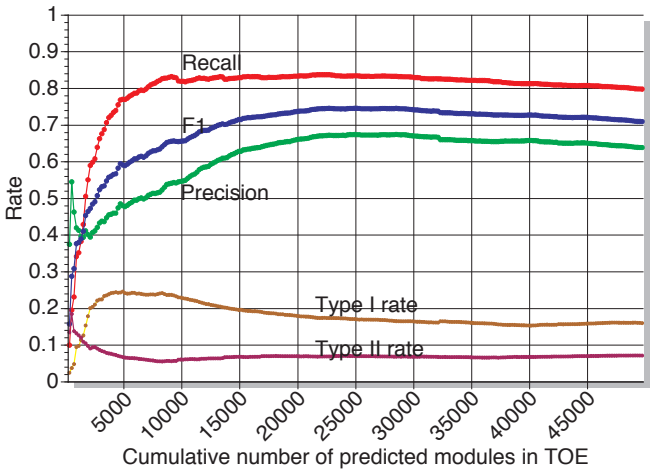
As for the transition of the evaluation measurements, these measurements are expected to increase as time elapses because more training usually achieves more accuracy. However, recall and precision do not follow this expectation because the number of type I errors (N_2 in Table 1) and the number of type II errors (N_3) increase more rapidly than the number of correctly predicted as FP (N_4).



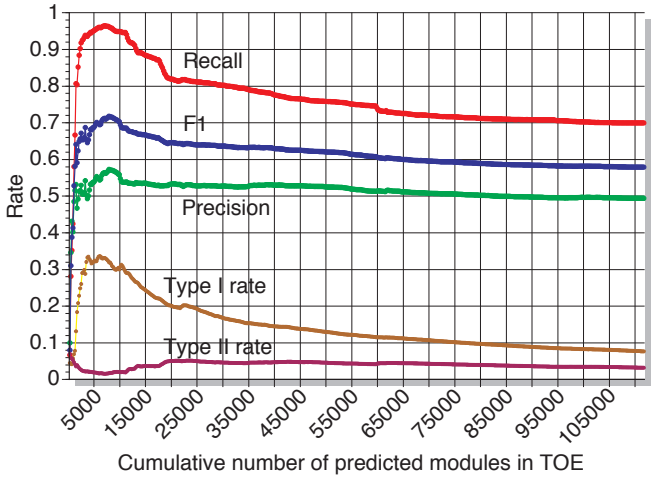
(a) Experiment e_1: Static threshold ($t_{FP} = 0.50$) in BIRT



(b) Experiment e_2: Static threshold ($t_{FP} = 0.90$) in BIRT



(c) Experiment e_3: Dynamic threshold in BIRT



(d) Experiment e_4: Dynamic threshold in EMF

Figure 5: Transition of evaluation measurements in TOE experiment

However, the type II error rate remains low during the entire development. This low type II error rate means that the number of misclassifications of actual FP modules is always small.

5.2 Difference between Projects

Both recall and precision in the experiment for BIRT are higher than that for EMF. In order to find a reason, we examine the transition of the number of FP modules in both projects. Figure 6 shows the number of actual FP modules per 1000 modules as well as the transitions of recall, precision, and F_1 measurement.

As shown in Figure 6 (a), the number of FP modules per 1000 modules was about 200 to 400 during the whole development of BIRT. However, in Figure 6 (b), most of FP modules appear at an early stage of development in EMF.

We guess that the difference of the number of FP modules per 1000 modules between BIRT and EMF derives the difference of growth of recall, precision, and the F_1 measurements between BIRT and EMF.

5.3 Comparative Study

In this section, we compare evaluation measurements with our previous study and other related studies in order to show the effectiveness of our approach. For comparison, we surveyed previous studies and compared the evaluation measurements shown in these other studies with our own.

Twenty-four classification results in five fault-prone prediction studies in 2007 were surveyed. The following describes a summary of these studies:

Menzies07 Menzies et al. compared 3 classification techniques for fault-prone prediction in [11]. They then concluded that the naive Bayesian classifier is the most accurate. They also used NASA's MDP (PC1, PC2, PC3, PC4, MW1, KC3, KC4, CM1) as well as the 38 metrics related to Halstead, McCabe, etc.

Kamei07 Kamei et al. adopted over/under sampling methods techniques for fault-prone prediction [9]. They experimentally evaluated four sampling methods (ROS, SMOTE, RUS, ONESS) by using 2 module sets of industry legacy software.

Arisholm07 Arisholm et al. compared various classification techniques in [1]. They used a C4.5 classification tree, Practical Machine Learning Tools (PART), a Support Vector Machine classifier (SVM), Logistic Regression, and Neural Networks.

Table 7: Comparison with previous fault-prone prediction studies

Study	Approach	Recall	Precision	F_1
Menzies07 [11]	Naive Bayes	0.79	0.703*°	0.74*°
Kamei07 [9]	Sampling method	0.595	0.282	0.382*
Arisholm07 [1]	C4.5 classification tree	0.711*	0.047*	0.088*°
	PART	0.785*	0.046*	0.087*°
	SVM	0.745*	0.047*	0.088*°
	Logistic Regression	0.758*	0.054*	0.101*°
	DecorateC4.5	0.765*	0.055*	0.103*°
	BoostC4.5	0.752*	0.047*	0.088*°
	CFSC4.5	0.779*	0.048*	0.090*°
	C4.5 + PART	0.779*	0.051*	0.096*°
	Neural Networks	0.732*	0.058*	0.107*°
Aversano07 [2]	KNN	0.588*	0.588	0.588*°
	Simple Logistic Regression	0.211*	0.800*	0.334*°
	Multi-boosting	0.300	0.520	0.380*°
	C4.5 classification tree	0.290*	0.488	0.364*°
	SVM	0.316*	0.399	0.353*°
Mizuno07 [13]	FP Filtering ($t_{FP} = 0.50$)†	0.590	0.506	0.545*
Hata08	Extended FP Filtering‡	**0.798**	**0.639**	**0.710***

* The best value shown in paper.
° Calculated from data shown in paper.
† Classified modules are Java methods. Project Eclipse (# of faults is 40,627.)
‡ Classified modules are Java classes. Project Eclipse BIRT (# of faults is 7,788.)

In addition, Arisholm et al. compared improvement of the C4.5 classification tree, named DecorqateC4.5, BoostC4.5, CFSC4.5, C4.5+PART.

Aversano07 Aversano et al. compared five prediction models with two projects in [2]. They used K-Nearest Neighbors (KNN), Simple logistic regression, Multi-boosting, the C4.5 classification tree, and Support Vector Machine classifier (SVM).

Mizuno07 This is our previous approach [13]. We proposed a spam-filter based approach named fault-prone filtering with a static threshold between FP and NFP judgment and method-based training for corpuses.

Table 7 shows evaluation measurements (recall, precision, and F_1) shown in these studies.

Each row in Table 7 shows the best F_1 measure and the corresponding measurements for a classification technique. The mark "*" with a value in Table 7 denotes that the value is the best case in the paper. The mark "°" indicates that we calculated the value from the data shown in the paper.

Menzies07 achieves high F_1 with high recall and precision in their best case, which is in good condition. However,in their paper, there is a low F_1 with another data set in not good condition, for example, 0.04 of F_1 and 0.02 of precision, which were calculated from data shown in their paper.

Aversano07 used bug-introducing changes for training, which are similar to our approach. However, they used other prediction models. Our approach obtain more higher recall and F_1.

Mizuno07, which is our previous study, shows relatively low recall. In the study, we got high recall, 0.839, with low precision, 0.232, which is an unbalanced result. In addition, F_1 is low, 0.363. Compared with previous studies, we can get twice the precision with about the same recall. As a result, we can improve 15% on the best F_1.

Table 7 shows the difficulty in a good balance of recall and precision which leads to a high F_1. For example, Arisholm07 obtained high recall and low precision, and Aversano07 obtained high precision and low recall. However we can obtain high recall precision, and F_1.

Since this is a survey-based comparison, we cannot validate advantage of our approach in a rigorous manner. More rigorous comparison should be done in the future work.

6. THREATS TO VALIDITY

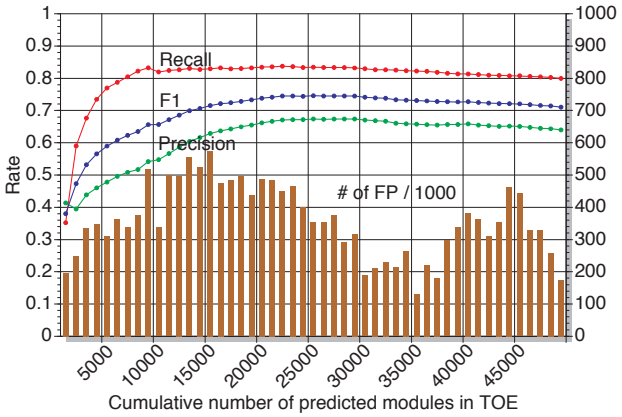
The threats to validity are categorized into 4 threats as recommended in [16]: external, internal, conclusion, and construction validity. In this study, we include external and construction validity.

One of the external validity threats for our study is the generality of the result. In general, the Eclipse project does much better than other projects when using machine learning classifiers to predict fault-prone modules. We must study with other projects and analyze performance across them.

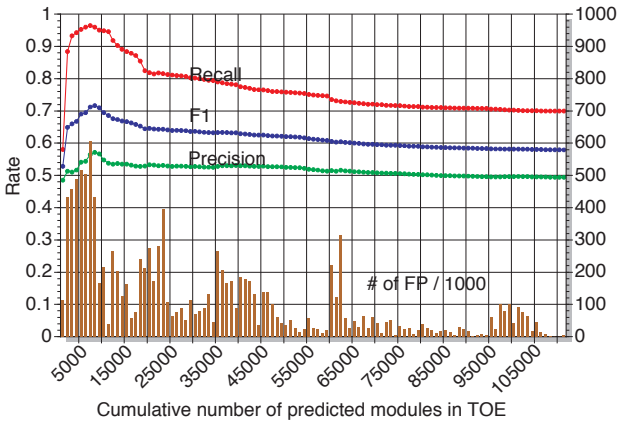
One of the construction validity threats is the collection of fault-prone modules from open source software projects. The algorithm adopted in this study has a limitation in that faults that are not recorded in a cvs log cannot be collected. To make an accurate collection of FP modules from source code repository, further research is required.

Another construction validity threats is that the threshold does not converge with our procedure. For this reason, misclassification happens when the threshold is very low or very high. To reduce such misclassification, we have to create more appropriate procedures to change the threshold.

One more construction validity threats is that our extension of smaller granularity of training units, modified lines of code, may be ignoring the context. That is, when certain lines are modified



(a) Experiment e_3: BIRT



(b) Experiment e_4: EMF

Figure 6: Transition of evaluation measurements with the number of FP modules per 1000 modules

as bug-introducing lines, which may be completely different from another context where the same lines are changed but not introduce a bug. However, bigger granularity of training units may contain much noise. This is a trade-off problem.

7. CONCLUSION

This paper showed the extension of training only errors procedure to classify fault-prone software modules using the spam filtering technique. The result of our experiment showed that our approach leads to twice the precision with about the same recall and improves 15% on the best F_1 measurement. In addition, our extension brings another merit, that is, independence from a programing language.

By analyzing our results, we can say that precision converges quickly without enough FP modules and precision gets higher with enough FP modules. In addition, enough FP modules at an early stage might bring high precision.

Our future research includes creating more appropriate procedures to change the threshold. We expect more accurate classification with more appropriate procedures.

8. ACKNOWLEDGMENTS

The authors would like to express their thanks to the developers of the CRM114 classifier. Without the CRM114, this work could not be conducted. Finally, authors also thank to the developers of Eclipse who have made the repository of Eclipse available for research.

9. REFERENCES

[1] E. Arisholm, L. C. Briand, and M. J. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Proc. of 18th International Symposium on Software Reliability Engineering (ISSRE2007)*, pages 215–224, 2007.

[2] L. Aversano, L. Cerulo, and C. D. Grosso. Learning from bug-introducing changes to prevent fault prone code. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 19–26, New York, NY, USA, 2007. ACM.

[3] P. Bellini, I. Bruno, P. Nesi, and D. Rogai. Comparing fault-proneness estimation models. In *Proc. of 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 205–214, 2005.

[4] L. C. Briand, W. L. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. on Software Engineering*, 28(7):706–720, 2002.

[5] *CRM114 – the Controllable Regex Mutilator*. http://crm114.sourceforge.net/.

[6] G. Denaro and M. Pezze. An empirical evaluation of fault-proneness models. In *Proc. of 24th International Conference on Software Engineering (ICSE '02)*, pages 241–251, 2002.

[7] *Eclipse Project*. http://www.eclipse.org/.

[8] L. Guo, B. Cukic, and H. Singh. Predicting fault prone modules by the Dempster-Shafer belief networks. In *Proc. of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 249–252, 2003.

[9] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM2007)*, pages 196–204, September 2007.

[10] T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical study. *Empirical Software Engineering*, 9:229–257, 2004.

[11] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. on Software Engineering*, 33(1):2–13, January 2007.

[12] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno. Spam filter based approach for finding fault-prone software modules. In *Proc. of 2007 International Workshop on Mining Software Repositories (MSR2007)*, 2007.

[13] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007)*, pages 405–414, 2007. Dubrovnik, Croatia.

[14] N. Seliya, T. M. Khoshgoftaar, and S. Zhong. Analyzing software quality with limited fault-proneness defect data. In *Proc. of Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*, pages 89–98, 2005.

[15] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? (on Fridays.). In *Proc. of Mining Software Repository 2005*, pages 24–28, 2005.

[16] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: An introduction*. Kluwer Academic Publishers, 2000.