IEICE TRANS. INF. & SYST., VOL.E91–D, NO.4 APRIL 2008

PAPER Special Section on Knowledge-Based Software Engineering

Prediction of Fault-Prone Software Modules Using a Generic Text Discriminator

Osamu MIZUNO[†a)], ***Member and*** **Tohru KIKUNO**[†], ***Fellow***

SUMMARY This paper describes a novel approach for detecting fault-prone modules using a spam filtering technique. Fault-prone module detection in source code is important for the assurance of software quality. Most previous fault-prone detection approaches have been based on using software metrics. Such approaches, however, have difficulties in collecting the metrics and constructing mathematical models based on the metrics. Because of the increase in the need for spam e-mail detection, the spam filtering technique has progressed as a convenient and effective technique for text mining. In our approach, fault-prone modules are detected in such a way that the source code modules are considered text files and are applied to the spam filter directly. To show the applicability of our approach, we conducted experimental applications using source code repositories of Java based open source developments. The result of experiments shows that our approach can correctly predict 78% of actual fault-prone modules as fault-prone.

key words: *fault-prone module, prediction, spam filter*

1. Introduction

Fault-prone prediction is one of the most mature area of software engineering. The prediction of faulty software modules is important for both the reduction of development cost and the assurance of software quality. Much research has been conducted so far [2], [4], [7], [11], [13]–[18], [20], [24]. Most research used some kind of software metrics, such as program complexity, size of modules, object-oriented metrics, etc., and constructed mathematical models to calculate fault-proneness. This approach is usually based on a hypothesis that the more complex module the more bugs. However, such a hypothesis is not always true.

We thus tried to break through the conventional fault-prone prediction by introducing a text-mining technique. This paper introduces a new idea for fault-prone module detection. The idea is inspired from a spam e-mail filtering technique. According to Postini Inc.'s report, 94% of all e-mail messages on the Internet were spam in November 2006 [23]. Such an explosive increase of spam e-mail messages triggered the development of many spam filtering techniques [3], [22].

In spam e-mail filtering, incoming e-mail messages are classified into spam or ham (non-spam) based on the frequency of tokens appearing in e-mail messages. Recently, since the usefulness of Bayesian theory for spam filtering has been acknowledged, most spam filtering tools have implemented this theory. Consequently, the accuracy of spam detection has been improved drastically.

Spam filters are usually implemented as a generic text discriminator. We thus tried to apply a generic text discriminator to the fault-prone detection. We call our approach "fault-prone filtering." 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) modules or not-fault-prone (NFP) modules. After learning of existing FP and NFP modules, we can classify a new module into either FP or NFP by applying a spam filter. One advantage of such a statistical approach is that we do not have to investigate source code modules in detail. We do not measure any metrics explicitly, but implicitly our approach measures only one metric: frequency of tokens found in the source code.

To validate the usefulness of our approach, we describe experiments using Java-based open source developments, such as argoUML and the Eclipse BIRT plugin. In these experiments, we consider a method in Java class as a module. That is, the judgments of FP and NFP were applied to methods in Java source code. We then performed 10-fold cross validation to evaluate our approach. The result showed that the best classifier can classify 78% of actual fault-prone modules correctly. This result implies that "fault-prone filtering" may be useful for detecting fault-prone modules.

The rest of this paper is organized as follows: Section 2 describes work related to this study. The outline of "fault-prone filtering" is then described in Sect. 3. An experiment to show the effectiveness of our approach is shown in Sect. 4. Section 5 discusses the result obtained in the experiment and Sect. 6 shows threats to the validity of our research. Finally, Sect. 7 summarizes this study and also addresses future work.

2. Related Work

2.1 Fault-Prone Prediction

Fault-prone prediction is a mature area in software engineering with various studies having been done over the past 20 years. From 1999, for example, many studies have been conducted [2], [4], [7], [11], [13]–[18], [20], [24].

Software metrics related to program attributes such as lines of code, complexity, frequency of modification, coherency, coupling, etc., have been used in many previous

Manuscript received July 2, 2007.
Manuscript revised October 15, 2007.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: o-mizuno@ist.osaka-u.ac.jp

DOI: 10.1093/ietisy/e91–d.4.888

studies. In those studies, such metrics are considered explanatory variables and fault-proneness is considered an objective variable. Mathematical models are constructed from those metrics. The selection of metrics varies according to studies. For example, studies such as [11], [20], [24] used NASA's Metrics Data Program. Object oriented metrics are used in [4], for example. Some studies used metrics based on metrics collection tools [2], [7].

Our approach does not use software metrics explicitly, but rather, uses the frequency of tokens (combination of words) in code modules as metrics. To the best of our knowledge, no work has used the frequency of tokens as predictors of fault proneness. The calculation of frequency of tokens is done by a spam filtering tool.

The selection of classification techniques also varies according to studies. Khoshgoftaar et al. performed a series of fault-prone prediction studies using various classification techniques; for example, the classification and regression trees [18], the tree-based classification with S-PLUS [15], the Treedisc algorithm [14], the Sprint-Sliq algorithm [16], and logistic regression [13]. The comparison was summarized in [17]. Logistic regression is a frequently used technique in fault-prone prediction [4], [7], [13]. Menzies et al. compared three classification techniques and reported that the naive Bayesian classifier achieved the best accuracy [20].

Our approach adopted a Markov random field for the classification technique. Since it is an extension of the naive Bayesian classifier, the Markov random field is expected to achieve good accuracy.

2.2 Spam E-mail Filtering

Early stages of spam-filtering software were based mainly on pattern matching using dictionaries of spam-prone words in e-mail messages. However, dealing with new spam e-mail messages including novel words is difficult. As a result, spammers and developers of spam filters were in the rat race.

In 2002, Graham stated that most spam e-mail messages can be automatically classified by Bayesian classification [10]. The merit of Bayesian classification is flexibility for new spam messages and the user's corrections. Inspired by his article, various spam filtering software based on Bayesian classification have been developed [3], [22]. Since traditional spam filters such as SpamAssassin [27] also implemented Bayesian techniques, Bayesian classification has become an essential technique.

CRM114 has been developed by Yerazunis [5] as an extension of a Bayesian classification-based filtering. CRM114 has been implemented as a generic text discriminator with remarkable accuracy in detecting spam messages.

3. Overview of Fault-Prone Filtering

3.1 Fundamental Idea

The basic idea of fault-prone filtering is inspired from spam e-mail filtering. In spam e-mail filtering, the spam filter first learns both spam and ham (non-spam) e-mail messages from a learning 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 a similar contexts. We thus guessed that faulty software modules have similar pattern of words or sentences like spam e-mail messages.

From the viewpoint of effort, conventional fault-prone detection techniques require a relatively great 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 requires extra effort and translating the values of metrics into meaningful results also requires additional effort. Thus, easy-to-use techniques that do 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 "fault-prone filtering". That is, the fault-prone learner first learns 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 spam filtering software and sets of FP and NFP modules.

Figure 1 shows an overview of Fault-prone Filtering.

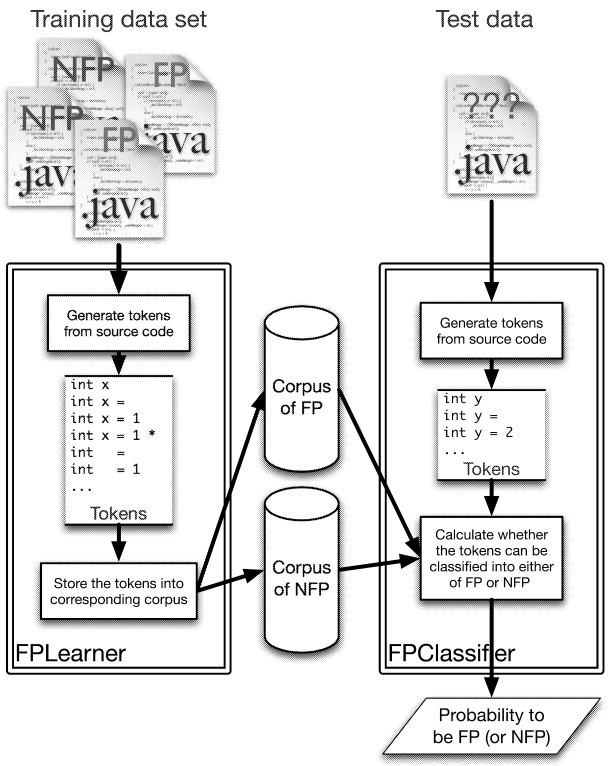

Fig. 1 Outline of FP filtering.

3.2 Classification Techniques

In this study, we used CRM114 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 following 3 classification strategies built in CRM114 to evaluate the effectiveness of our proposed approach.

1. Sparse Binary Polynomial Hash Markov model (SBPH)
   SBPH is the default classification model used in CRM114. It is an extension of the Bayesian classification and maps features in the input text into a Markov Random Field [5]. In this model, tokens are constructed from combinations of n words (n-grams) in a text file. Tokens are then mapped into a Markov random field to calculate the probability.
2. Orthogonal Sparse Bigrams Markov model (OSB)
   OSB is a simplified version of SBPH. It consider 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 SBPH [25].
3. Simple Bayesian model (BAYES)
   BAYES is a simplified version of SBPH, since it uses only single words as tokens. This model is thus considered to be identical to the classical Bayesian classification.

As developers of CRM114 stated, those classifiers have both merits and demerits. To investigate the applicability to FP filtering, we compare the accuracy of these 3 strategies when they are applied to FP filtering in the experiment.

The scripts of CRM114 used for learning and classification are shown in the Appendix.

3.3 Example of Filtering

Here, we explain briefly how these classifiers works. The difference among these 3 classifiers are in both tokenization and classification.

(1) Tokenization

Since SBPH is a base for all techniques, we explain how SBPH tokenizes input text files. First, words in a source code module are separated by a lexical analyzer. Then, separators such as braces, parentheses, colons, and semicolons are deleted. SBPH then picks up a sequence of 5 words. Next, SBPH generates combinations of these words by fixing the first word. For example, a sentence "`if (x == 1) return;`" is tokenized as shown in Fig. 2 (a).

```
 1: if
 2: if x
 3: if    ==
 4: if       1
 5: if          return
 6: if x  ==
 7: if x     1
 8: if x        return
 9: if    == 1
10: if    ==    return
11: if       1  return
12: if x  == 1
13: if x  ==    return
14: if x     1  return
15: if    == 1  return
16: if x  == 1  return
```

(a) Tokens for SBPH

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

(b) Tokens for OSB

Fig. 2 Example of tokens for SBPH.

For all words in a source code module, the above procedure is applied and the tokens are obtained.

In OSB, tokens are extracted from SBPH generated tokens so that these tokens include exactly 2 words. Thus, in the same example as SBPH, tokens are generated as shown in Fig. 2 (b). By definition, the number of tokens drastically decreases compared to SBPH.

In BAYES, a single word is considered a token. In this example, 5 tokens are obtained as follows: `if`, `x`, `==`, `1`, `return`.

(2) Classification

Let T_{FP} and T_{NFP} be sets of tokens included in FP and 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 SBPH and OSB, the probability that a new module m_{new} is faulty, $P(T_{FP}|T_{m_{new}})$, with a given set of token $T_{m_{new}}$ in a new source code module m_{new} is calculated by the following Bayesian formula:

$$\frac{P(T_{m_{\mathrm{new}}}|T_{\mathrm{FP}})P(T_{\mathrm{FP}})}{P(T_{m_{\mathrm{new}}}|T_{\mathrm{FP}})P(T_{\mathrm{FP}}) + P(T_{m_{\mathrm{new}}}|T_{\mathrm{NFP}})P(T_{\mathrm{NFP}})}$$

Intuitively speaking, this probability denotes that the new code is classified into FP. According to $P(T_{FP}|T_{m_{new}})$ and pre-defined threshold t_{FP}, classification is performed.

3.4 Classification Example

Here, we present a very simple example of how modules are classified by the OSB. Figures 3 (a) and (b) show examples of FP and NFP modules, respectively. The module `fact()` is intended to calculate a factorial of a given `x` recursively. However, implementation in Fig. 3 (a) includes a bug `++x` in line 2 which should be `--x`.

Assume that FPTrainer trains only these 2 modules. In this case, bigrams are generated from both modules and trained as either FP or NFP. The difference between FP and

```
1: public int fact(int x) {
2:     return(x==1?1:x*fact(++x));
3: }
```

(a) Example of an FP module

```
1: public int fact(int x) {
2:     return(x==1?1:x*fact(--x));
3: }
```

(b) Example of an NFP module

Fig. 3 Example code for classification.

```
FP1: :          ++
FP2:    x       ++
FP3:      *     ++
FP4:        fact ++
FP5:             ++ x
```

(a) Tokens from FP module

```
NFP1: :          --
NFP2:    x       --
NFP3:      *     --
NFP4:        fact --
NFP5:             -- x
```

(b) Tokens from NFP module

Fig. 4 Difference of generated tokens for FP and NFP modules.

```
1: public int sigma(int y) {
2:     return(y==1?1:y+sigma(++y));
3: }
```

Fig. 5 Example of a new module.

NFP tokens is shown in Figs. 4 (a) and (b). All tokens generated from Fig. 3 are shown in Fig. 6.

In Fig. 4 (a), tokens `FPx` are trained as characteristic of FP modules and stored in the FP corpus. Similarly, tokens `NFPx` in Fig. 4 (b) are trained as NFP and stored in the NFP corpus. All other tokens are stored in both corpuses, too.

Next, assume that the new module shown in Fig. 5 is constructed and has to be classified. After tokenization, we can obtain the tokens shown in Fig. 6 (c).

Figure 6 shows the contents of the FP and NFP corpuses and tokens generated from the new module m_{new}. We can see the following from Fig. 6:

- The number of all tokens in Figs. 6 (a), (b), and (c) is 45.
- The number of common tokens between Figs. 6 (a) and (c) is 14. They are shown as grayed rectangle in Fig. 6 (a).
- The number of common tokens between Figs. 6 (b) and (c) is 13. They are shown in the grayed rectangle in Fig. 6 (b).

For example, we can see that the 37th token of ": ++" in the new module in Fig. 6 (c) is also found in line 37 of the FP corpus in Fig. 6 (a), too.

By Bayesian formula, we can get a probability to be fault-prone for the new module. In this example, $P(T_{\mathrm{FP}}) = P(T_{\mathrm{NFP}}) = 1/2$ since there are only 2 modules trained. As mentioned before, the number of tokens in both FP and NFP corpuses is 45. The number of identical tokens between FP and the new module is 14. The number of identical tokens between NFP and the new module is 13. Thus, $P(T_{m_{\mathrm{new}}}|T_{\mathrm{FP}}) = 14/45$ and $P(T_{m_{\mathrm{new}}}|T_{\mathrm{NFP}}) = 13/45$. The probability that the new code is classified as FP is thus calculated as follows:

$$P(T_{\mathrm{FP}}|T_{m_{\mathrm{new}}}) = \frac{\frac{14}{45} \times \frac{1}{2}}{\frac{14}{45} \times \frac{1}{2} + \frac{13}{45} \times \frac{1}{2}} = 0.519.$$

As a result, a new module in Fig. 5 is classified as FP with the probability of 0.519. In fact, the new module has a similar bug as the FP module in Fig. 3. That is, ++ should be --.

Our approach is based on the tendency that developers

	(a) Tokens from FP module	(b) Tokens from NFP module	(c) Tokens from new module
1	public int	public int	public int
2	public fact	public fact	public sigma
3	public x	public x	public y
4	int fact	int fact	int sigma
5	2 int int	2 int int	int int
6	int x	int x	int y
7	3 int return	3 int return	int return
8	fact int	fact int	sigma int
9	fact x	fact x	sigma y
10	fact return	fact return	sigma return
11	4 int ==	4 int ==	int ==
12	x return	x return	y return
13	x x	x x	y y
14	x ==	x ==	y ==
15	x 1	x 1	y 1
16	return x	return x	return y
17	5 return ==	5 return ==	return ==
18	6 return 1	6 return 1	return 1
19	7 return ?	7 return ?	return ?
20	x ?	x ?	y ?
21	x :	x :	y :
22	8 == 1	8 == 1	== 1
23	9 == ?	9 == ?	== ?
24	10 == :	10 == :	== :
25	== x	== x	== y
26	11 1 ?	11 1 ?	1 ?
27	12 1 :	12 1 :	1 :
28	1 x	1 x	1 y
29	1 *	1 *	1 +
30	13 ? :	13 ? :	? :
31	? x	? x	? y
32	? *	? *	? +
33	? fact	? fact	? sigma
34	: x	: x	: y
35	: *	: *	: +
36	: fact	: fact	: sigma
37	14 : ++	: --	: ++
38	x *	x *	y +
39	x fact	x fact	y sigma
40	x ++	x --	y ++
41	* fact	* fact	+ sigma
42	* ++	* --	+ ++
43	* x	* x	+ y
44	fact ++	fact --	sigma ++
45	++ x	-- x	++ y

Fig. 6 Tokens generated for NFP, FP, and new modules.

often make similar mistakes and thus inject similar bugs. In other words, there is a pattern of bugs for individual developers. In this example, mistaking -- for ++ tends to take place in different modules. Such bugs are difficult to obtain by metrics-based fault-prone predictions. By using the spam-filtering technique, we try to capture such similar patterns of bugs.

Of course, this is just a trivial example. In a real situation, many of other tokens also affect classification. The calculation of probability thus becomes more complex.

4. Experimental Application

To see the effectiveness of the proposed approach, we have conducted an experiment to evaluate the accuracy of classification strategies.

4.1 Target Projects

For the experiment, we selected an open source project that can track faults. For this reason, we selected two projects, "argoUML project [1]" and "Eclipse BIRT plugin [9]".

Table 1 shows the context of the target projects. Both projects are developed in Java language, and revisions are maintained by a concurrent version control system (cvs). The source repository of argoUML was prepared for use in the Mining Challenge in the Mining Software Repository Workshop in 2006 [8]. As for the Eclipse BIRT plugin, an archive of the repository was obtained from the official Web site on the 27th of November, 2006. Fault reports were obtained from the bug database of both projects. The type of faults is "bugs", therefore these faults do not include any enhancements or functional patches. The status of the faults is either "resolved", "verified", or "closed", and the resolution of faults is "fixed". This status means that the collected faults have already been resolved and fixed and thus this fixed revision should be included in the entire repository. As for the Eclipse BIRT, the severity of faults are also specified. Faults with "blocker", "critical", "major", and "normal" are collected in this experiment.

4.2 Collecting Fault-Prone Modules

We have to collect both fault-prone (FP) modules and non fault-prone (NFP) modules from the source code repository for this research. The collection of such modules seems easy for a software project which has a bug database such as an Open Source Software development. However, even in such an environment, the revision control system and bug database system are usually separated and thus tracking on the fault-prone modules requires effort. In the development of software in companies, the situation becomes more difficult [17].

Table 1 Target projects.

Name	argoUML	eclipse BIRT plugin
Language	Java	Java
Revision control	cvs	cvs
Type of faults	Bugs	Bugs
Status of faults	Resolved, Verified, Closed	Resolved, Verified, Closed
Resolution of faults	Fixed	Fixed
Severity	N/A	blocker, critical, major, normal
Priority of faults	all	all
Total number of faults	1058	4708

We have to extract FP and NFP modules by ourselves. We assumed that the target project is a Java-based development in this study. We also assumed that a module of source code is a method in Java class. We then extracted FP modules from source code based on an algorithm by Sliwerski et al. [26].

The following restriction and assumption exist in this collection method:

Restriction We seek FP modules by examining the cvs log. Therefore, faults that do not appear in the cvs log 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.

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

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

Here, we consider the software module M_i as a tuple of d_i, m_i, and s_i^a, where d_i is the last modified date of M_i, m_i is a source code of M_i, and s_i^a is the actual fault status (FP or NFP) of M_i.

We then start mining a source code repository according to the following algorithm to extract fault-prone modules.

1. For each fault f_i, find class files $CL_{\text{FaultFixed}}$ in which the fault has just been fixed by checking all revision logs.
2. Extract modules $MOD_{\text{FaultFixed}}$ in classes $CL_{\text{FaultFixed}}$.
3. For each module M_i in $MOD_{\text{FaultFixed}}$, let $s_i^a = \text{FP}$ if M_i is unmodified since $date(f_i)$.
4. Let $MOD_{FP} = \{M_i | s_i^a = \text{FP}\}$
5. Extract modules MOD_{AllRev} in all revision.
6. For each module M_j in MOD_{FP}, track back older revisions of M_j and append older revisions of M_j to MOD_{FPold} only if M_j has remained unchanged until the bug is fixed.
7. Let $MOD_{\text{NFP}} = MOD_{\text{AllRev}} - MOD_{\text{FPold}} - MOD_{\text{FP}}$. For each module M_k in MOD_{NFP}, let $s_k^a = \text{NFP}$.

This algorithm collects fault-prone modules very strictly. In other words, we collect modules in which faults are definitely included. Therefore, some modules are not collected as FP since there is a possibility that the module is not FP.

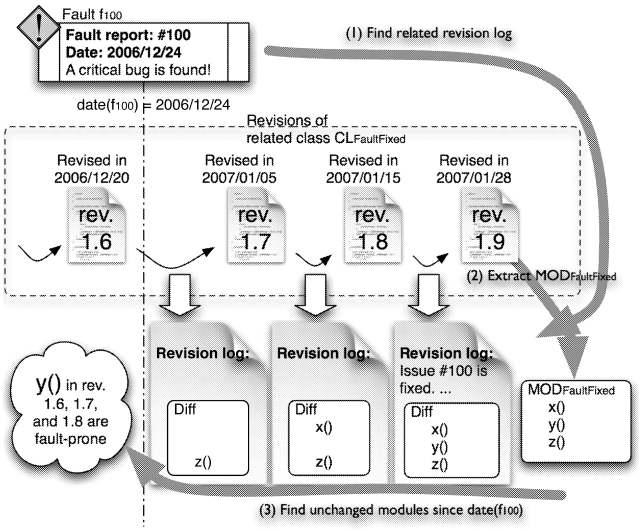

Fig. 7 Collection of FP modules.

An illustrated example of collecting a fault-prone module is shown in Fig. 7. In this example, assume that a class $CL_{\text{FaultFixed}}$ has revisions 1.1, 1.2, $\cdots$, 1.9, and revision logs are appended when each revision is committed. At first, a fault f_{100} is found on the 24th of December, 2006. By searching all revision logs, assume that the fixed point is found as revision 1.9 of $CL_{\text{FaultFixed}}$ (Shown as (1) in Fig. 7). Then, $MOD_{\text{FaultFixed}}$ can be extracted by taking the difference between revision 1.8 and 1.9 (Shown as (2) in Fig. 7). For each module in $MOD_{\text{FaultFixed}}$, we find modules which have not been modified since the 24th of December, 2006 by searching revision differences. Here, assume that revision 1.6 of $CL_{\text{FaultFixed}}$ was committed on 20th of December, 2006. Therefore, we have to check all differences between revision 1.6 and 1.9. Assume that the difference between revision 1.7 and 1.8 includes modification to `x()` and `z()`, and the difference between revision 1.6 and 1.7 includes modification to `z()`. Then, we can find that the modification to `y()` between 1.8 and 1.9 is the first modification since the fault f_{100} was reported, and that the fault f_{100} is fixed whereupon `y()` is modified. This implies that `y()` is a cause of fault f_{100}. The modules `y()` in revision 1.6, 1.7, and 1.8 are then added to MOD_{FP} (Shown as (3) in Fig. 7). On the other hand, modules such as `x()` and `z()` in revision 1.8 are not included in MOD_{FP}, because they are modified between 1.6 and 1.8 for some reason. Of course, they may include the cause of f_{100}, but the probability is smaller than that of `y()` in 1.8. Therefore we do not include `x()` and `z()` in revision 1.8 in MOD_{FP}.

We implemented a prototype tool named "FPFinder" to track bugs in the cvs repository. The inputs of FPFinder is a cvs repository of the target project and a bug report to track the bugs. The output of FPFinder are sets of FP modules (MOD_{FP}) and NFP modules (MOD_{NFP}).

The result of FPFinder is shown in Table 2. The number of faults found in the cvs log of argoUML is 396. This number was 37% of the total reported faults in the bug database. As for Eclipse BIRT, 1973 faults were found in the cvs log and this number was 42% of total.

Table 2 Result of FPFinder for target projects.

Name	argoUML	eclipse BIRT plugin
# of faults found in cvs log	396 (37% of total)	1973 (42% of total)
# of FP modules	1093	9547
# of NFP modules	20219	86770

Table 3 Legend of experimental result.

		Predicted	
		NFP	FP
Actual	NFP	N_1	N_2
	FP	N_3	N_4

4.3 Application of FPClassifier

We performed a 10-fold cross validation with 3 classifiers using modules in MOD_{FP} and MOD_{NFP}.

Here, let us explain the details of the 10-fold cross validation in this experiment. First, we randomly shuffled the order of the modules in MOD_{FP} and MOD_{NFP}. Next, we split MOD_{FP} into 10 subsets $MOD^1_{\text{FP}}, MOD^2_{\text{FP}}, \cdots, MOD^{10}_{\text{FP}}$ so that the number of modules in each subset is almost the same. Similarly, we split MOD_{NFP} into MOD^1_{NFP}, $MOD^2_{\text{NFP}}, \cdots, MOD^{10}_{\text{NFP}}$.

We then picked MOD^1_{NFP} and MOD^1_{FP} for testing data. Obviously, $MOD^2_{\text{NFP}}, \cdots, MOD^{10}_{\text{NFP}}$ and $MOD^2_{\text{FP}}, \cdots$, MOD^{10}_{FP} are used for learning. Next, $MOD^2_{\text{NFP}}, \cdots, MOD^{10}_{\text{NFP}}$ and $MOD^2_{\text{FP}}, \cdots, MOD^{10}_{\text{FP}}$ are learnt to the corpuses. Using the learnt corpus, MOD^1_{NFP} and MOD^1_{FP} are classified into FP or NFP by calculating the probability of being fault-prone. Here, threshold t_{FP} for classifying FP or NFP is determined to be 0.5. By only using the subset MOD^i_{NFP} and MOD^i_{FP} for testing, we can obtain a relatively fair result of the classification.

4.4 Result of Experiment

For the evaluation of the experiments, we define several evaluation measurements. Table 3 shows a legend of tables for experimental result. In Table 3, 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. Therefore, $N_1 + N_4$ is the number of correctly predicted modules. The accuracy rate shows the ratio of correctly predicted modules to entire modules and is defined as follows:

$$\text{accuracy} = \frac{N_1 + N_4}{N_1 + N_2 + N_3 + N_4}$$

Table 4 Result of 10-fold cross validation for argoUML.

(a) Result by SBPH

SBPH		Predicted	
		NFP	FP
Actual	NFP	25,822	457
	FP	1,096	482

(b) Result by OSB

OSB		Predicted	
		NFP	FP
Actual	NFP	22,550	3,792
	FP	493	1,085

(c) Result by BAYES

BAYES		Predicted	
		NFP	FP
Actual	NFP	26,275	4
	FP	1,537	41

Table 5 Result of 10-fold cross validation for eclipse BIRT plugin.

(a) Result by SBPH

SBPH		Predicted	
		NFP	FP
Actual	NFP	84,234	2,146
	FP	6,647	2,893

(b) Result by OSB

OSB		Predicted	
		NFP	FP
Actual	NFP	70,475	15,905
	FP	2,039	7,501

(c) Result by BAYES

BAYES		Predicted	
		NFP	FP
Actual	NFP	86,246	134
	FP	8,948	592

Table 6 Evaluation metrics for argoUML.

Classifier	Precision	Recall	F_1	Accuracy
SBPH	0.513	0.305	0.383	0.944
OSB	0.225	0.687	0.339	0.848
BAYES	0.911	0.026	0.051	0.945

Table 7 Evaluation metrics for eclipse BIRT plugin.

Classifier	Precision	Recall	F_1	Accuracy
SBPH	0.574	0.303	0.397	0.908
OSB	0.320	0.786	0.455	0.812
BAYES	0.815	0.062	0.115	0.905

For evaluation purposes, we used two measurements: recall, precision, and F_1. Recall is the ratio of modules correctly predicted as FP to the number of entire modules that are actually FP. The ratio 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. This ratio 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 the actual FP modules from the predicted FP modules.

Finally, F_1 is a combined evaluation criterion of recall and precision [12]. F_1 is defined as follows:

$$F_1 = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

Tables 4 and 5 show the results of the 10-fold cross validation using 3 classifiers for argoUML and the Eclipse BIRT plugin, respectively. The result of the evaluation by recall, precision, and F_1 is shown in Tables 6 and 7.

5. Discussion

5.1 Comparison between Classifiers

To investigate the difference of classifiers, we look into the result of the Eclipse BIRT plugin in Tables 5 and 7.

By looking at Table 5 carefully, we can find that OSB can predict more FP modules than the other 2 classifiers. That is, $N_4 = 7{,}501$ in Table 5 (b), $N_4 = 2{,}893$ in Table 5 (a), and $N_4 = 592$ in Table 5 (c). The amount of N_4 is one of the most important measurements for prediction of fault-proneness.

From Table 7, we can see several characteristics of each classifier. The F_1 and recall of the BAYES classifier were the worst (0.062 and 0.115, respectively) in 3 classifiers. In other words, BAYES predicts almost all modules to be NFP. The reason, we assume, is because source code modules of software are more similar to each other than e-mail messages are. BAYES considers a word in a text file as a token for learning. However, too much similar words appear in software code modules. The characteristics of FP modules were difficult to extract by BAYES. In other classifiers, more than 2 words are used for a token. This extension drastically increases the number of tokens to learn. The difference of BAYES and the other classifiers may be derived from too many similar words.

The measurement "accuracy" shows an extremely high value even in BAYES, but this fact is almost meaningless since the accuracy is strongly affected from unbalanced data. For example, if all modules are simply predicted as NFP, the accuracy becomes more than 0.9. Thus, we do not use accuracy for evaluation. We showed the values of accuracy only for reference.

The OSB classifier achieves the most relevant result for FP prediction. Since recall is 0.786, we can say that 78% of actual fault-prone modules can be detected by the OSB classifier. The fact that precision is as low as 0.320 implies that we have to investigate 3 modules to find 1 fault-prone module on average.

In fault-prone modules prediction, recall implies the coverage of actual faults and precision implies the cost

needed for testing. However, the balance of the cost of testing and the quality of software is always a problem of testing economics [19]. It is said that a type-II error requires 10 to 200 times more cost to remove than a type-I error [17]. We thus believe that the coverage of actual faults is more important than the cost of testing in an actual software development and such imbalanced recall and precision are acceptable.

Note that we can adjust the balance of recall and precision by changing the threshold of probability t_{FP} shown in Sect. 3.3. According to the domain of development, the balance of recall and precision should be adjusted.

The results for argoUML in Tables 4 and 6 also show that the OSB achieves the best recall and the worst precision. We can say that the tendency between the two projects is almost the same.

Therefore, we can conclude that the OSB classifier is the most appropriate classifier for fault-prone prediction from the viewpoint of quality assurance.

5.2 Comparison between Previous Studies

Here, we briefly survey the accuracy of previous metrics-based studies shown in Sect. 2.1. Denaro et al. proposed a logistic regression based approach, and achieved 0.906 precision and 0.682 recall [7]. Briand et al. adopted logistic regression and object-oriented metrics based method, and achieved 0.840 precision and 0.483 recall [4]. Guo et al. proposed an approach using Dempster-Shafer Belief Network and achieved 0.690 precision and 0.915 recall [11]. Bellini et al. used the discriminate analysis and achieved 0.736 precision and 0.543 recall [2].

On the other hand, our approach with the OSB classifier achieved 0.320 precision and 0.786 recall. Since the environment of application differs from previous studies, we cannot compare these results directly. We can see that the recall of our approach is relatively better, but the precision is rather lower than previous results.

6. Threats to Validity

The threats to validity are categorized into four categories as in [28]: external, internal, conclusion, and construction validity.

External validity mainly includes the generalizability of the proposed approach. For these experiments, we can confirm that the fault-prone filtering works correctly for two projects. In Tables 6 and 7, we can see that the tendency of evaluation measurements is similar in both projects. This fact implies that our approach has a certain kind of generality. Therefore, the threat to external validity is mitigated.

As for internal validity, 10-fold cross validation can be a threat. Although cross validation is a good method for validation of a prediction technique, cross validation cannot deal with several important aspects such as the order of the creation of modules. One solution is the 'Training only errors' approach [21]. The approach can deal with the order of creation and the modification of software modules. We have conducted an experiment using this approach.

One of the construction validity threats is the collection of fault-prone modules from open source software projects. As mentioned before, the number of faults found in the cvs repository was about 40% of the total faults reported in a bug database. The algorithm adopted in this study has the limitation that faults not recorded in the cvs log cannot be collected. To make an accurate collection of FP modules from the source code repository, further research is required.

The way of statistical analysis usually causes threats to conclusion validity. We cannot find any threats to conclusion validity in our study at this point.

7. Conclusion

This paper proposed an approach to classify fault-prone software modules using a spam filtering technique. In our approach, source code modules were considered text files and were applied to the spam filter directly. For the spam filtering software, we selected a generic text discriminator.

We conducted an experiment using source code repositories of Java-based open source developments. The result of our experiment showed that the OSB classifier is the most appropriate for fault-prone prediction. By using the OSB classifier, our approach can classify 78% of actual fault-prone modules as fault-prone.

For future work, we have to apply our approach to not only open source development, but also to actual development in industries. Additionally, further investigation of misclassified modules will contribute to improvement of accuracy. Finally, an application environment has to be developed.

Acknowledgement

The authors would like to thank Mr. Shiro Ikami and Mr. Shuya Nakaichi who made remarkable contributions to an earlier version of this research. The authors also would like to thank Dr. Juichi Takahashi in SONY corporation who gave us much useful advice.

References

[1] ArgoUML Project. http://argouml.tigris.org/
[2] P. Bellini, I. Bruno, P. Nesi, and D. Rogai, "Comparing fault-proneness estimation models," Proc. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05), pp.205–214, 2005.
[3] bogofilter. http://bogofilter.sourceforge.net/
[4] L.C. Briand, W.L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," IEEE Trans. Softw. Eng., vol.28, no.7, pp.706–720, 2002.
[5] S. Chhabra, W.S. Yerazunis, and C. Siefkes, "Spam filtering using a Markov random field model with variable weighting schemas," Proc. Fourth IEEE International Conference on Data Mining (ICDM 2004), pp.347–350, 2004.
[6] CRM114 — the Controllable Regex Mutilator. http://crm114.sourceforge.net/

[7] G. Denaro and M. Pezze, "An empirical evaluation of fault-proneness models," Proc. 24th International Conference on Software Engineering (ICSE '02), pp.241–251, 2002.
[8] S. Diehl, H. Gall, and A.E. Hassan, ed., Proc. 2006 International Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, ACM, May, 2006.
[9] Eclipse Project. http://www.eclipse.org/
[10] P. Graham, Hackers and Painters: Big Ideas from the Computer Age, chapter 8, pp.121–129, O'Reilly Media, 2004.
[11] L. Guo, B. Cukic, and H. Singh, "Predicting fault prone modules by the dempster-shafer belief networks," Proc. 18th IEEE International Conference on Automated Software Engineering (ASE '03), pp.249–252, 2003.
[12] J.L. Herlocker, J.A. Konstan, L.G. Terveen, and J.T. Riedl, "Evaluating collaborative filtering recommender systems," ACM Trans. Inf. Syst., vol.22, no.1, pp.5–53, 2004.
[13] T.M. Khoshgoftaar and E.B. Allen, "Logistic regression modeling of software quality," International Journal of Reliability, Quality and Safety Engineeering, vol.6, no.4, pp.303–317, 1999.
[14] T.M. Khoshgoftaar and E.B. Allen, "Controlling overfitting in classification tree models of software quality," Empir. Softw. Eng., vol.6, no.1, pp.59–79, 2001.
[15] T.M. Khoshgoftaar, E.B. Allen, and J. Deng, "Using regression trees to classify fault-prone software modules," IEEE Trans. Reliab., vol.51, no.4, pp.455–462, 2002.
[16] T.M. Khoshgoftaar and N. Seliya, "Software quality classification modeling using SPRINT decision tree algorithm," Proc. 14th International Conference on Tools with Artificial Intelligence, pp.365–374, 2002.
[17] T.M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical study," Empir. Softw. Eng., vol.9, pp.229–257, 2004.
[18] T.M. Khoshgoftaar, R. Shan, and E.B. Allen, "Using product, process, and execution metrics to predict fault-prone software modules with classification trees," Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE '00), pp.301–310, 2000.
[19] D.M. Marks, Testing very big systems, McGraw-Hill, 1992.
[20] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. Softw. Eng., vol.33, no.1, pp.2–13, Jan. 2007.
[21] O. Mizuno and T. Kikuno, "Training on errors experiment to detect fault-prone software modules by spam filter," 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007), pp.405–414, Dubrovnik, Croatia, 2007.
[22] POPFile. http://popfile.sourceforge.net/
[23] Postini Inc., Postini Announces Top Five 2007 Messaging Security Predictions As Email Spam Becomes Front Burner Issue Again In The New Year. http://www.postini.com/news_events/pr/pr120606.php
[24] N. Seliya, T.M. Khoshgoftaar, and S. Zhong, "Analyzing software quality with limited fault-proneness defect data," Proc. Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE '05), pp.89–98, 2005.
[25] C. Siefkes, F. Assis, S. Chhabra, and W.S. Yerazunis, "Combining winnow and orthogonal sparse bigrams for incremental spam filtering," Proc. Conference on Machine Learning (ECML)/European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), 2004.
[26] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? (on fridays.)," Proc. Mining Software Repository 2005, pp.24–28, 2005.
[27] SpamAssassin. http://spamassassin.apache.org/index.html
[28] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering: An introduction, Kluwer Academic Publishers, 2000.

Appendix: CRM114 Scripts for Learning and Classification

CRM114 is a powerful tool for generic text discrimination and it accepts original scripting language. Figures A· 1 and A· 2 show scripts of CRM114 used in our study. In these scripts, $CLASSIFIER is replaced with the following options according to the selected classifier:

SBPH `<unique microgroom>`
OSB `<osb unique microgroomx>`
BAYES `<unigram>`.

```
match (:data:) /.*/
learn $CLASSIFIER (:*:_arg2:) [:data:] /[[:graph:]]+/
```

Fig. A· 1 Learning script for CRM114.

```
match (:data:) /.*/
isolate (:stats:)
{
  classify $CLASSIFIER [:data:]
    (:*:_arg2: | :*:_arg3:) (:stats:) /[[:graph:]]+/
  output /file :*:_arg2: matches better :*:_nl::*:stats::*:_nl:/
  exit
}
output /file :*:_arg3: matches better :*:_nl::*:stats::*:_nl:/
```

Fig. A· 2 Classification script for CRM114.

Osamu Mizuno received M.E. and Ph.D. degrees from Osaka University in 1998 and 2001, respectively. He is currently an Assistant Professor in the Graduate School of Information Science and Technology at Osaka University. His current research interests include fault-prone module prediction, software process improvement, and risk evaluation and prediction of software development. He is a member of the IEEE.

Tohru Kikuno received M.Sc. and Ph.D. degrees from Osaka University in 1972 and 1975, respectively. He joined Hiroshima University from 1975 to 1987. Since 1990, he has been a Professor in the Department of Information and Computer Sciences at Osaka University. Since 2002, he has been a Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include the analysis and design of fault-tolerant systems, the quantitative evaluation of software development processes, and the design of procedures for testing communication protocols. He is a senior member of IEEE, a member of ACM, and a fellow of IPSJ (Information Processing Society of Japan).