Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter

Osamu Mizuno
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita
Osaka 565-0871, Japan
o-mizuno@ist.osaka-u.ac.jp

Tohru Kikuno
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita
Osaka 565-0871, Japan
kikuno@ist.osaka-u.ac.jp

ABSTRACT

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

In order to mitigate such difficulties, we propose a novel approach for detecting fault-prone modules using a spam filtering technique, named Fault-Prone Filtering. Because of the increase of needs for spam e-mail detection, the spam filtering technique has been progressed as a convenient and effective technique for text mining. In our approach, fault-prone modules are detected in a way that the source code modules are considered as text files and are applied to the spam filter directly.

This paper describes the training on errors procedure to apply fault-prone filtering in practice. Since no pre-training is required, this procedure can be applied to actual development field immediately. In order to show the usefulness of our approach, we conducted an experiment using a large source code repository of Java based open source project. The result of experiment shows that our approach can classify about 85% of software modules correctly. The result also indicates that fault-prone modules can be detected relatively low cost at an early stage.

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

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

1. INTRODUCTION

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

Various studies have been done in the detection of the fault-prone modules [1,3,6,9,11–17,22]. Most of them 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 [18]. The spam filter is one of the most widely used text mining applications. According to Postini Inc.'s report, 94% of entire e-mail messages on the Internet are spam on November 2006 [21]. Such explosive increase of spam e-mail messages triggered development of a lot of spam filtering techniques [2, 20]. In the spam e-mail filtering, incoming e-mail messages are classified into spam or ham (non-spam) based on the frequency of tokens appeared in e-mail messages. Recently, since the usefulness of Bayesian theory for the spam filtering has been recognized, most of spam filtering tools implemented them. Consequently, the accuracy of spam detection has been improved drastically.

Inspired by the spam filtering technique, we tried to apply text mining technique to the fault-prone detection. In the fault-prone filtering, we consider a software module as an e-mail message, and assume that all of software modules belong to either fault-prone(FP) or not-fault-prone(NFP). In the previous research [18], although we have conducted experiments using 10-fold cross validation, the experiments were not sufficient to show practical usefulness of our approach. Furthermore, the target software in the experiment was relatively small.

In this paper, we thus conduct a new experiment based on training on errors(TOE) procedure that can simulate practical situation. Training on errors procedure is a reasonable way of classification and training of spam e-mail filtering. In this procedure, e-mail messages are classified in arrival order. Only misclassified e-mail messages are used for training for further classification. This procedure reduces the time for training and avoids too much training.

The experiment is prepared to simulate actual TOE process using the source code repository of eclipse project [7]. For the experiment, methods in Java code are considered as software modules. FP and NFP modules are then collected from the repository. We then conducted TOE simulation using about 1.2 million modules and classified them into FP or NFP. The result of experiment shows

that our approach can detect about 90% of modules correctly. Furthermore, we can see that about 85% actual FP modules can be predicted correctly and critical misclassifications rarely happen.

The rest of this paper is organized as follows: Section 2 describes related works to this study. The outline of "fault-prone filtering" is then described in Section 3. An experiment to show the effectiveness of our approach is shown in Section 4. Section 5 discusses on the result obtained in the experiment. In Section 6, we compared our approach with other fault-prone prediction studies based on survey. Section 7 addresses the threats to validity of this study. Finally, Section 8 summarizes this study and also addresses the future work.

2. RELATED WORKS

2.1 Fault-Prone Detection

Much research on detection of fault prone software modules has been carried out so far. Just looking for works since 1999, a lot of fault prone prediction studies are found [1, 3, 6, 9, 11–17, 22]. Previous studies can be categorized by data sets, metrics, and classification techniques.

Data sets used in these studies are three folds: public domain, open source, and original. First, as for the public domain data sets, one of the most famous public domain data set is the NASA's Metrics Data Program(MDP) [19]. For example, studies such as [9, 17, 22] used the NASA's MDP. By using such public domain data sets, a new approach can be easily comparable with other approaches. Our approach, however, cannot be applied to such data sets since it requires raw software code. Next, as for the open source software data, studies such as [3, 6] collected and used for the evaluation of their fault-prone prediction approaches. Finally, original data sets are usually used in empirical studies in industries [1, 15]. Especially, Khoshgoftaar used a data from very large embedded software system and evaluated classification techniques [15].

Software metrics related to program attribute such as lines of code, complexity, frequency of modification, coherency, coupling, and so on are used in most of previous studies. In those studies, such metrics are considered as explanatory variables and fault-proneness are considered as an objective variable. Then mathematical models are constructed from those metrics. The selection of metrics varies according to studies. For example, studies such as [9, 17, 22] used NASA MDP collected metrics. The object oriented metrics are used in [3]. Some studies used metrics based on metrics collection tools [1, 6].

Our approach does not use software metrics explicitly. Accurately speaking, it uses the frequency of tokens (combination of words) in code modules as metrics. To our best knowledge, there is no work that used the frequency of tokens as predictors of fault proneness. The calculation of frequency of tokens is done by spam filtering tool. Brief explanation of calculation is shown in subsection 3.4.

Selection of classification techniques also varies according to studies. Khoshgoftaar et al. has been performed a series of fault-prone prediction studies using various classification techniques. For example, classification and regression trees [16], tree-based classification with S-PLUS [13], the Treedisc algorithm [12], Sprint-Sliq algorithm [14], logistic regression [11]. The comparison was summarized in [15]. Logistic regression is one of frequently used technique in fault-prone prediction [3, 6, 11]. Menzies et al. compared three classification techniques and reported that Naive Bayesian classifier achieved the best accuracy [17].

Our approach adopted Markov random field for the classification technique. Since it is an extension of naive Bayesian classifier, it is expected to achieve good accuracy. The comparative study based on survey is shown in Section 6.

2.2 Spam E-mail Filtering

Spam e-mail filtering is one of the most practical application of text classification technique nowadays.

At an early stage of spam filtering software, it was mainly based on pattern matching using dictionaries of spam-prone words in e-mail messages. However, it is difficult to deal with new spam e-mail messages including new words. As a result, spammers and spam filters have been in the rat race.

Graham stated in his article that most spam e-mail messages can be automatically classified by Bayesian classification in his article [8]. The merit of Bayesian classification is flexibility for new spam messages and user's correction. Inspired by his article, various spam filtering software based on Bayesian classification have been developed [2, 20]. Since traditional spam filters such as SpamAssassin [25] also implemented Bayesian technique in them, Bayesian classification becomes essential technique for spam filtering nowadays.

CRM114 is developed by Yerazunis [4] as an extension of Bayesian classification based filtering and it has remarkable accuracy on detecting spam messages. Since it is implemented as a generic text discriminator, it can be applied to not only spam filtering but also data stream analysis.

3. FAULT-PRONE FILTERING

3.1 Fundamental Idea

The basic idea of fault-prone filtering is inspired by spam mail filtering. In the spam mail filtering, the spam filter first trains both spam and ham (non-spam) e-mail messages from 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 include particular patterns of words or sentences. From a viewpoint of source code, similar situation usually occurs in faulty software modules. That is, similar faults may occur in similar contexts. We thus guessed that faulty software modules have similar pattern 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.

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

We then try to apply a spam filter to identification of fault-prone modules. We named this approach as "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 spam filtering software and sets of FP and NFP modules.

3.2 Procedure of Filtering (Training on Errors)

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

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

1. Apply FPClassifier to a newly created software module (say,

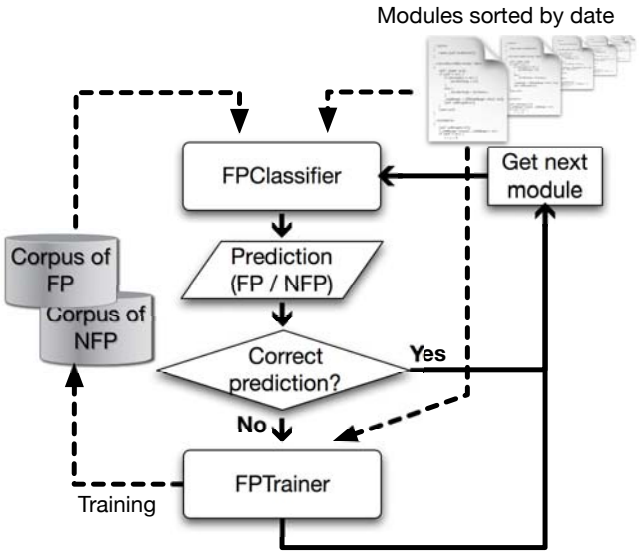


Figure 1: Outline of Fault-Prone Filtering by Training on Errors

method in Java, function in C, and so on), M_i, and obtain the probability to be fault-prone.

2. By the pre-determined threshold t_{FP} $(0 < t_{FP} < 1)$, 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 FPTrainer to M_i to learn actual fault-proneness and go to step 1.

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 arrived. If some of them are misclassified, actual results (spam or non-spam) should be trained.

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

3.3 Classification Techniques

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. Furthermore, the classification techniques implemented in CRM114 are mainly based on Markov random field model [4] instead of naive Bayesian classifier.

In this experiment, we used the following 2 classification strategies built in CRM114, which have relatively good accuracy.

1. Sparse Binary Polynomial Hash Markov model (SBPH) [4]

   SBPH is an extension of Bayesian classification, mapping features in the input text into a Markov Random Field. In this model, tokens are constructed from combinations of n words (n-grams) in a text file. In CRM114, the number of words n is set as 5 by default. It is determined by experience of CRM114 developers. Tokens are then stored in corpuses via hash tables.

   Intuitively speaking, SBPH constructs tokens at most n words and uses it as predictors.

   Using n-grams for tokenization seems to be effective for our objective, fault-prone prediction of software source code. For the spam filtering, the simple Bayesian classification can achieve high accuracy of filtering. In the simple Bayesian classification, the frequency of a single word is stored in corpus. However, since the difference between an FP module and an NFP module is more subtle than e-mail messages, a certain combination or sequence of words should be considered.

2. Orthogonal Sparse Bigrams Markov model (OSB) [23]

   OSB is a simplified version of SBPH and the default classification model used in CRM114. It consider tokens as combinations of exactly 2 words created in the SBPH model. This decreases both memory consumption of training and time of classification. Furthermore, it is reported that OSB usually achieves higher accuracy than SBPH [23] even though it is a simplified version of SBPH.

   Intuitively speaking, OSB constructs tokens with 2 words and uses it as predictors.

Let me explain the difference between these text classifiers and typical Bayesian text classifier. The typical Bayesian text classifier utilizes the frequency of single words appeared in input text. In the case of e-mail filtering, the frequency of single words is enough to achieve high accuracy. However, in the case of source code, the problem is more complex. In our previous study, we showed that the use of simple Bayesian text classifier did not achieve enough accuracy [18].

3.4 Probability Calculation

Here, we explain how these classifiers works briefly. The difference among these 2 classifiers are in both tokenization and classification.

3.4.1 Tokenization

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

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

In OSB, tokens are extracted from SBPH generated ones so that they include exactly 2 words in it. Thus, in the same example as SBPH, tokens are generated as shown in Figure 3. By definition, the number of tokens drastically decreases compared to SBPH.

3.4.2 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 given set of token $T_{m_{new}}$ in a new source code module m_{new} is calculated by

```
 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
```

Figure 2: Example of tokens for SBPH

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

Figure 3: Example of tokens for OSB

the following Bayesian formula:

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

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.

3.5 Classification Example

Here, we present a very simple example of how modules are classified by the OSB. Figures 4 (a) and (b) show examples of FP and NFP modules, respectively. The module `fact` intended to calculate a factorial of given `x` recursively. However, implementation in Figure 4 (a) includes a bug that `++x` in line 3 should be `--x`.

Assume that FPTrainer trains these 2 modules only. In this case, Bigrams are generated from both modules and trained as either FP or NFP. The difference between FP and NFP tokens are shown in Figure 5 (a) and (b), respectively.

In Figure 5 (a), tokens `FPx` are trained as characteristic of FP modules and stored in FP corpus. Similarly, tokens `NFPx` in Figure 5 (b) are trained as NFP and stored in NFP corpus. All other tokens are stored in both corpuses, too. However, since they are identical between FP and NFP modules, they have no effect on future classification.

```
1: public int fact(int x) {
2:     if (x == 1) return 1;
3:     return(x*fact(++x));
4: }
```

(a) Example of an FP module

```
1: public int fact(int x) {
2:     if (x == 1) return 1;
3:     return(x*fact(--x));
4: }
```

(b) Example of an NFP module

Figure 4: Example code for classification

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

(a) Tokens for line 3 of Figure 4 (a)

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

(b) Tokens for line 3 of Figure 4 (b)

Figure 5: Difference of generated tokens for FP and NFP modules

```
1: public int sigma(int x) {
2:     if (x == 1) return 1;
3:     return(x+sigma(++x));
4: }
```

Figure 6: Example of a new module

Then, assume that a new module shown in Figure 6 is constructed and have to be classified. After tokenization, we can obtain the tokens shown in Figure 7 near line 3 of Figure 6.

We can see that tokens `NEW1`, `NEW2`, `NEW8` are found in FP corpus. According to equation (1), we can get 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. The number of tokens in both FP and NFP corpuses is 58. The number of identical tokens between FP and new module is 53. The number of identical tokens between NFP and new module is 50. Thus, $P(T_{m_{\mathrm{new}}}|T_{\mathrm{FP}}) = 40/58$ and $P(T_{m_{\mathrm{new}}}|T_{\mathrm{NFP}}) = 37/58$. The probability that the new code is classified as FP is thus calculated as follows:

$$\begin{aligned} P(T_{\mathrm{FP}}|T_{m_{\mathrm{new}}}) &= \frac{\frac{40}{58} \times \frac{1}{2}}{\frac{40}{58} \times \frac{1}{2} + \frac{37}{58} \times \frac{1}{2}} \\ &= 0.519 \end{aligned}$$

As a result, a new module in Figure 6 is classified as FP with probability of 0.519. In fact, the new module has the similar bug that the FP module in Figure 4 (a) has.

Our approach is based on the tendency that developers often make similar mistakes and thus inject similar bugs. In other words,

```
NEW1: return            ++
NEW2:         x         ++
NEW3:           +       ++
NEW4:             sigma ++
NEW5:         x             x
NEW6:           +           x
NEW7:             sigma     x
NEW8:                   ++ x
```

Figure 7: Generated tokens for line 3 of Figure 6

Table 1: Target project (eclipse)

Name	eclipse
Language	Java
Revision control	cvs
Size of entire repository	14 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	40,627

it is a pattern of bugs for individual developer. In this example, making mistake `--x` for `++x` tends to take place in different modules. By using spam filtering technique, we try to capture such similar patterns of bugs.

Of course, this is just a trivial example. In the real situation, there is a lot of other tokens that affects classification. The calculation of probability thus becomes complex.

4. TRAINING ON ERRORS EXPERIMENT

4.1 Target Project

For the experiment, we selected an open source project that can track faults. For this reason, we selected eclipse project [7]. Table 1 shows the context of the target project. The eclipse is constructed in Java language, and revisions are maintained by concurrent version control system (cvs). The source repository of cvs used in this study is uploaded one on the eclipse project Web site, and is obtained in 27th January, 2007. We obtained fault reports from the bug database of eclipse project [7]. The total number of faults found in bug database was 40,627 in the following condition: The type of these faults is "bugs", therefore these faults do not include any enhancements or functional patches. The status of faults are either "resolved", "verified", or "closed", and the resolution of faults is "fixed". This means that the collected faults have already resolved and fixed 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 research.

4.2 Collection of Fault-Prone Modules for Experiment

We have to collect both fault-prone(FP) modules and non fault-prone(NFP) modules from 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 needs effort. In the development of software in companies, the situation becomes harder [15].

We thus have to extract FP and NFP modules by ourselves. We assumed 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 shown by Sliwerski et al. [24].

The following restriction and assumption exist in this collection method:

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

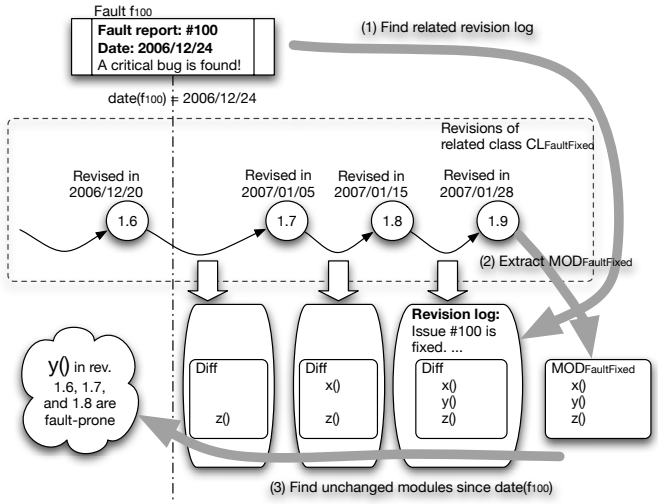


Figure 8: Collection of FP modules

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

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

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

Here, we consider a 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 source code of M_i, and s_i^a is 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 $CL_{\mathrm{FaultFixed}}$ in which the fault has just been fixed by checking all revision logs.
2. Extract modules $MOD_{\mathrm{FaultFixed}}$ in classes $CL_{\mathrm{FaultFixed}}$.
3. For each module M_i in $MOD_{\mathrm{FaultFixed}}$, let $s_i^a = \mathrm{FP}$ if M_i is unmodified since $date(f_i)$.
4. Let $MOD_{FP} = \{M_i | s_i^a = \mathrm{FP}\}$
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 the M_j has remained unchanged until the bug fix.
7. Let $MOD_{\mathrm{NFP}} = MOD_{\mathrm{AllRev}} - MOD_{\mathrm{FPold}} - MOD_{\mathrm{FP}}$. For each module M_k in MOD_{NFP}, let $s_k^a = \mathrm{NFP}$.

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

An illustrated example of collecting a fault-prone module is shown in Figure 8. In this example, assume that a class $CL_{\mathrm{FaultFixed}}$ has

Table 2: Result of FPFinder for eclipse

# of faults found in cvs log	21,761 (52% of total)
# of FP modules ($\|MOD_{\mathrm{FP}}\|$)	65,782
# of NFP modules ($\|MOD_{\mathrm{NFP}}\|$)	1,113,063

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 in 24th December, 2006. By searching all revision logs, assume that the fixed point is found as revision 1.9 of $CL_{\mathrm{FaultFixed}}$ (Shown as (1) in Figure 8). Then, $MOD_{\mathrm{FaultFixed}}$ can be extracted by taking a difference between revision 1.8 and 1.9 (Shown as (2) in Figure 8). For each module in $MOD_{\mathrm{FaultFixed}}$, we find modules which are not modified since 24th December, 2006 by searching revision differences. Here, assume that revision 1.6 of $CL_{\mathrm{FaultFixed}}$ is committed in 20th December, 2006. Therefore, we have to check all differences between revision 1.6 and 1.9. Assume that difference between revision 1.7 and 1.8 includes modification to `x()` and `z()`, and difference between revision 1.6 and 1.7 includes modification to `z()`. Then, we can find that modification to `y()` between 1.8 and 1.9 is the first modification since the fault f_{100} was reported, and the fault f_{100} is fixed hereupon `y()` is modified. This implies that `y()` has 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 Figure 8). On the other hand, the 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 by some reasons. Of course, they may include the cause of f_{100}, but the confidence is smaller than that of `y()` in 1.8. We thus 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 target project and a bug report to track. 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. Number of faults found in cvs log was 21,761. It is 52% of total reported faults in the bug database. The number of FP modules corresponded to these faults was 65,782. The number of NFP modules becomes huge, and is 1,113,063.

4.3 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 subsection 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 Figure 9. During the simulation, modules are classified in order of date. If the predicted result s_i^p differs from actual status s_i^a, the training procedure is invoked.

4.4 Result of Experiment

In the experiment we conducted 4 experiments with 2 classifiers and 2 threshold values. That is, (e1) SBPH classifier with $t_{FP} = 0.50$, (e2) SBPH classifier with $t_{FP} = 0.25$, (e3) OSB classifier with $t_{FP} = 0.50$, and (e4) OSB classifier with $t_{FP} = 0.25$. On the threshold, $t_{FP} = 0.50$ is a normal way of classification. On the other hand, $t_{FP} = 0.25$ means that the prediction of modules tends to be FP than $t_{FP} = 0.50$. This configuration is reasonable on fault-prone detection since FP modules must not be missed.

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 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 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. Accuracy rate shows the ratio of correctly predicted modules to entire modules and is defined as follows:

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

The rates of false positive and false negative are defined as follows:

$$\text{false positive rate} = \frac{N_2}{N_1 + N_3}$$

$$\text{false negative rate} = \frac{N_3}{N_2 + N_4}$$

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

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

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

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):
    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):
    Generate a set of tokens T_m from m.
    Store tokens T_m to the corpus T_{s^a}.

Figure 9: Procedure of TOE experiment

Table 3: Legend of experimental result

		Prediction	
		NFP	FP
Actual	NFP	N_1	N_2
	FP	N_3	N_4

Table 4: Final classification results in TOE experiment

(a) SBPH classifier with t_{FP}=0.50

		Prediction			
		NFP		FP	
Actual	NFP	1,075,162	(91.2%)	37,901	(3.2%)
	FP	26,947	(2.2%)	38,835	(3.3%)

(b) SBPH classifier with t_{FP}=0.25

		Prediction			
		NFP		FP	
Actual	NFP	1,071,765	(90.9%)	41,298	(3.5%)
	FP	26,292	(2.2%)	39,490	(3.3%)

(c) OSB classifier with t_{FP}=0.50

		Prediction			
		NFP		FP	
Actual	NFP	1,022,895	(86.8%)	90,168	(7.6%)
	FP	17,890	(1.5%)	47,892	(4.1%)

(d) OSB classifier with t_{FP}=0.25

		Prediction			
		NFP		FP	
Actual	NFP	930,218	(78.9%)	182,845	(15.5%)
	FP	10,592	(0.9%)	55,190	(4.7%)

Table 5: Evaluation measurements in TOE experiment

Classifiers	SBPH		OSB	
threshold (t_{FP})	0.50	0.25	0.50	0.25
Accuracy	0.945	0.943	0.908	0.835
Recall	0.590	0.600	0.728	0.839
Precision	0.506	0.489	0.347	0.232
False positive rate	0.034	0.038	0.087	0.194
False negative rate	0.351	0.325	0.129	0.044

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

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

Intuitively speaking, the precision implies the cost of the approach because small precision makes much efforts to find actually FP modules from predicted FP modules.

Table 4 shows the classification results for each experiment at the final point of train on errors experiment. The evaluation measurements for each experiment is shown in Table 5. From the viewpoint of accuracy, SBPH classifiers seems superior to OSB classifier. However, from the viewpoint of recall and false negative, the OSB is better than SBPH. From the viewpoint of threshold of fault-prone judgement, setting the threshold t_{FP} to lower value (that is, more likely to predict FP) makes recall and false negative better in both classifiers.

Figure 10 shows transitions of evaluation measurements for each experiment. In these graphs, the x-axis shows all software modules sorted by dates and smaller number shows older modules. The y-axis shows rates of accuracy, recall, precision, and so on.

The time needed for experiments using SBPH and OSB were 125 hours 11 minutes and 12 hours 15 minutes, respectively, on MacPro workstation with Xeon 2.66GHz processor. Indeed, OSB is ten times faster than SBPH in this experiment. Since there are over 1 million modules to be classified, classification time per module is less than 1 second even in SBPH.

5. ANALYSIS OF EXPERIMENTS

By investigating experimental results in more detail, we can obtain interesting findings.

5.1 Difference between Classifiers

Here we discuss the difference of classifiers. Comparing Table 4 (a) through (d), the difference between the SBPH and OSB is found on the capability of predicting FP modules. We can see in Table 4 (a) and (c) that the numbers of modules predicted as FP, $N_2 + N_4$, are 76,736 and 138,060 for SBPH with $t_{FP} = 0.5$ and OSB with $t_{FP} = 0.5$, respectively. Therefore, we can say that OSB tends to predict modules as FP even if it is not correct.

For the SBPH with $t_{FP} = 0.5$, the rates of false positive and false negative are 0.034 and 0.356, respectively (See Table 5.). On the other hand, for the OSB with $t_{FP} = 0.5$, they are 0.087 and 0.129, respectively (See Table 5, too.). In fault-prone detection, it is usually expected that actual FP modules should not be predicted as NFP. Low false negative rate helps to avoid such critical misclassification. Furthermore, the result of experiment showed that execution time of OSB is ten times faster than SBPH.

For these reasons, we conclude that OSB classifier is more better to apply our fault-prone filtering.

5.2 Difference of Threshold

The threshold between FP and NFP is one of the most effective parameters for the classification. From Table 4 (c) and (d), we can see that the number of predicted FP modules drastically increases in the case of OSB classifier. (However, in the case of SBPH shown in Table 4 (a) and (b), the change is much smaller than the case of OSB.) From Table 5, we can see that changing the threshold from 0.5 to 0.25 improves the recall, but the precision becomes worse. Since the recall and the precision is in trade-off, users should determine appropriate threshold for their purpose. In the software development, it is usually required to detect as many FP modules as possible. In such case, lower recall is preferred. However, lower recall increases the precision.

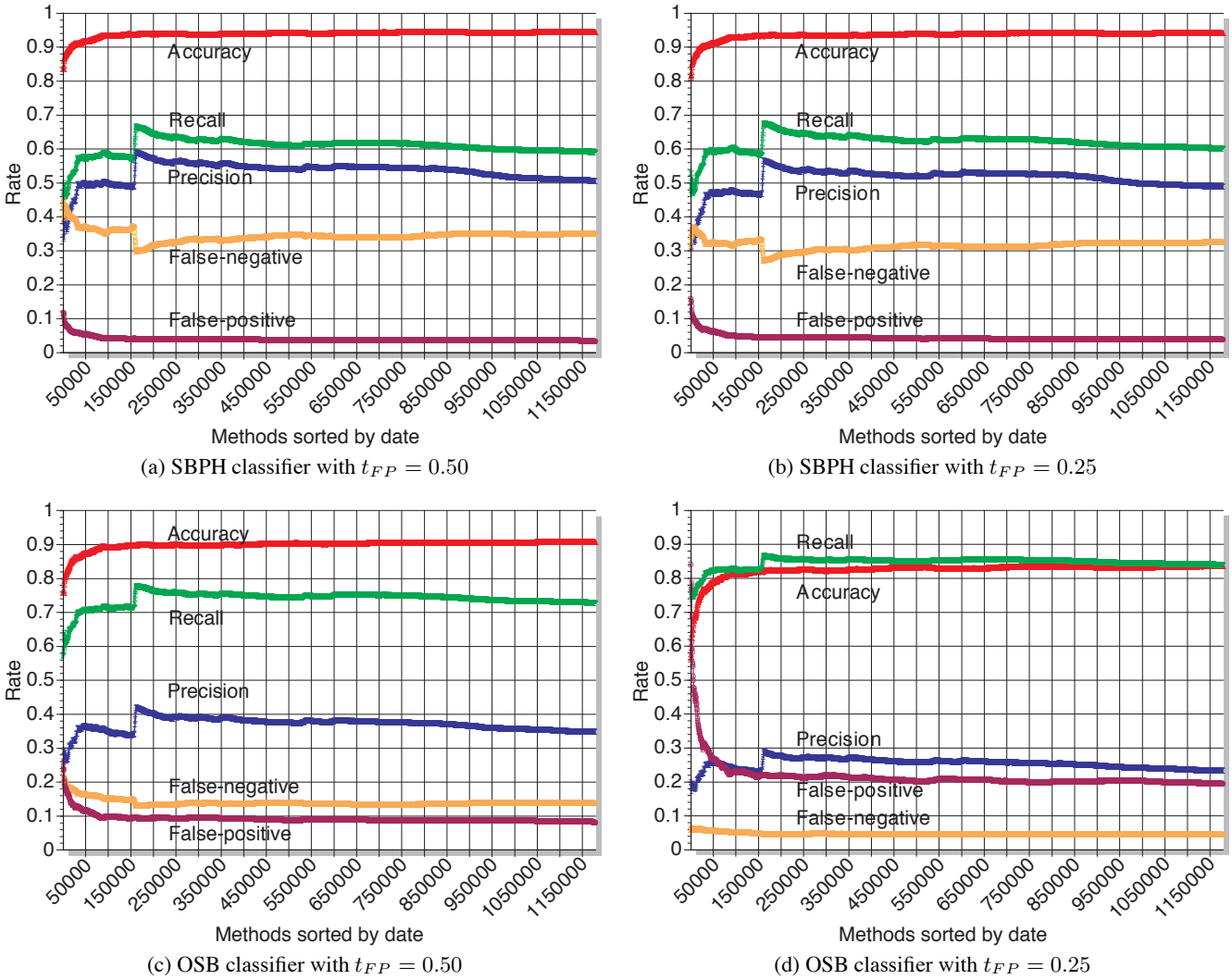
In a case of OSB classifier in the experiment in Table 5, recall improved from 0.728 to 0.839 by changing the threshold from 0.5 to 0.25. This means 83.9% of actual FP modules is covered in the predicted FP modules. However, the precision become 0.347 to 0.232. It means that 34.7% of predicted FP modules is actually FP when $t_{FP} = 0.5$ and the rate decreases to 23.2% when $t_{FP} = 0.25$. This implies the increase of detection cost because about 3 out of 4 FP predicted modules are actually NFP and investigating these 3 modules is in vain for detecting faults. At the same time, changing threshold decreases the number of false negative. The rate of false negative becomes 0.044 when $t_{FP} = 0.25$. In a case of SBPH, similar tendency is observed in Table 5, too. However, the degree of improvement is rather small.

We thus conclude that finding admissible threshold is required for the practical use of our approach.

5.3 Transition of Evaluation Measurements

In Figure 10, it is observed that measurements are not good at an early stage of the development. However, in all cases, the measurements became saturated after classification and training of 50,000 modules. It is about 4% of total number of modules. This fact indicates that TOE procedure works well after a certain period of the development.

Investigating the case of eclipse in more detail, it takes 12 months for the development of initial 50,000 modules. Since the eclipse has been maintained for 6 years long, it needs 1/6 of total period of the development until the fault-prone filtering takes effect.



(a) SBPH classifier with $t_{FP} = 0.50$

(b) SBPH classifier with $t_{FP} = 0.25$

(c) OSB classifier with $t_{FP} = 0.50$

(d) OSB classifier with $t_{FP} = 0.25$

Figure 10: Transition of evaluation measurements in TOE experiment

We can also see that recall and precision jumped up just after 150,000 modules. Concretely speaking, recall and precision started to raise up near 158,000 modules and stopped raising near 164,000 modules. During classification of these 6,000 modules, there are few misclassifications for both FP and NFP. Although we cannot see what was happened during this period exactly, we guess that some kind of roll back occurred or a number of code clones [10] are generated in wide area.

As for the transition of evaluation measurements, it is expected that these measurements increase as time elapses because the more training usually achieves more accuracy. However, the recall and the precision do not follow the expectation. It is because the number of false positives (N_2 in Table 3) and the number of false negatives (N_3) increase more rapidly than the number of correctly predicted as FP (N_4). In order to improve this situation, it is required to achieve more accurate fault-prone detection. To do so, for example, the source code oriented classification techniques may be effective.

6. COMPARATIVE STUDY

Since fault-prone prediction is a traditional research theme in software engineering, many works has been done so far. We here compare evaluation measurements with them in order to show the effectiveness of our approach.

For comparison, we did not perform experiments using other methods since contexts of previous studies differ widely. Alternatively, we surveyed previous studies and compare their evaluation measurements shown in their paper with our ones. Thirteen classification results in seven fault-prone prediction studies since 2002 were surveyed. The following describes summaries of these studies:

Denaro02 Denaro and Pezze proposed a logistic regression based fault-prone prediction [6]. The target is open source software (Apache 1.3 and 2.0). They used 38 metrics that can be collected by tools, RSM and TestBed.

Briand02 Briand et al. proposed an approach based on principle component analysis(PCA) and logistic regression [3]. They used open source software for experiment, Xpose and JWriter. A model is constructed using 22 object oriented metrics.

Guo03 Guo et al. used the Dempster-Shafer Belief Networks for fault-prone prediction [9]. They used NASA's MDP (KC2)

Table 6: Comparison with previous fault-prone prediction works

Study	Approach	Accuracy	Recall	false negative rate	% of N_4
Denaro02 [6]	Logistic regression	0.906*	0.682*	–	
Briand02 [3]	PCA + Logistic regression	0.840†	0.483*†	0.727†	10.4%
Guo03 [9]	Dempster-Shafer Belief Networks	0.690‡	0.915*	–	–
Khoshgoftaar04 [15]	Sprint-Sliq classification tree	0.747	–	0.208*	–
	Classification and regression tree	0.699	–	0.149*	–
	Regression tree in S-PLUS	0.734	–	0.213*	–
	Treedisc classification tree	0.721	–	0.255*	–
	C4.5 classification tree	0.746	–	0.213*	–
	Case-based reasoning	0.728	–	0.277*	–
	Logistic regression	0.723	–	0.128*	–
Bellini05 [1]	Discriminant analysis	0.736	0.543*†	0.568†	20.0%
Seliya05 [22]	Semi-supervised clustering	0.836*	–	0.274*	–
Menzies07 [17]	Naive Bayes	–	0.710** , 0.980*	–	–
Mizuno07	FP Filtering (CRM114, OSB, $t_{FP} = 0.25$)	**0.835**	**0.839***	**0.044***	**4.7%**

* The best value shown in paper.
** Average value shown in paper.
† Calculated from data shown in paper.
‡ Approximate value read from graph in paper.

for their case study and thus they used 21 metrics collected in NASA projects.

Khoshgoftaar04 Khoshgoftaar et al. compared various classification techniques for fault-prone prediction [15]. Classification techniques used in [15] were Sprint-Sliq classification tree, classification and regression tree (CART), Regression tree in S-PLUS, the Treedisc classification tree, C4.5 classification tree, case-based reasoning (CBR), and logistic regression. They applied their approach to very large embedded system written in high level language (more than 10 million LOC). They used 28 metrics including call graph metrics, control flow graph metrics, statement metrics, and software execution metrics. Ten-fold cross validation is used for evaluation.

Bellini05 Bellini et al. proposed a discriminant analysis based approach [1]. They showed two data sets, INDUSTRIAL and MEDICAL for experiment. The metrics are collected by CPP analyzer and PAMPA tools, and the number of metrics were 113.

Seliya05 Semi-supervised clustering based approach was proposed by Seliya et al. [22]. They also used NASA's MDP (JM1 and KC2), and 13 metrics such as line count, Halstead, McCabe, branch count, are used.

Menzies07 Menzies et al. compared three classification techniques for fault-prone prediction in [17]. Then they concluded that naive Bayesian classifier is the most accurate. They also used NASA's MDP (PC1, PC2, PC3, PC4, MW1, KC3, KC4, CM1) and used 38 metrics related to Halstead, McCabe, and so on.

Table 6 shows evaluation measurements (accuracy, recall, and false negative rate) shown in these studies. As mentioned before, since we consider that the recall is the most important in fault-prone prediction, we mainly picked up the best value of recall in each study. If the paper did not show the recall, we collected the best false negative rate instead.

Each row in Table 6 shows the best recall or false negative rate and the corresponding accuracy rate for a classification technique. The mark "*" with a value in Table 6 denotes that the value is the best case in the paper. The mark "**" denotes that the value is average value explicitly shown in the paper. The mark "†" indicates that we calculate the value from the data shown in the paper. The mark "‡" denotes that the value is approximate one since it was read from the graph.

Comparison of recall values showed that Guo03 and Menzies07 achieves extremely high recall in their best cases. However, the value of recall in our approach, 0.845, is still higher than studies such as Denaro02, Briand02, and Bellini05.

As for the false negative rate, Khoshgoftaar uses it as an evaluation measure in their fault-prone prediction works. We can see that our approach achieves lowest false negative rate (0.044) in Table 6.

As for the similarity of the context of target software, Denaro02 and Briand02 are more similar to our approach. Comparing our approach with these two works, accuracy of our approach (0.835) is slightly smaller than these two works. However, recall of our approach (0.839) is much higher than them.

Since this is a survey based comparison, we cannot validate advantage of our approach in a rigorous manner. However, we can show a certain degree of possibility that FP filtering can be applied to actual software development.

7. THREATS TO VALIDITY

The threats to validity are categorized into four as recommended in [26]: external, internal, conclusion, and construction validity. In this study, external and construction validity can be found.

One of the external validity threats for our study is the generalizability of the result. In the previous study [18], we applied our approach to several open source software projects, Eclipse BIRT plugin and argoUML. Even though these projects are much smaller than the Eclipse project, we got almost the same results to the experiment in this paper. More application to other projects, especially industrial ones, may mitigate this threat.

One of the construction validity threats is the collection of fault-prone modules from open source software projects. As I mentioned before, the number of faults found in Eclipse cvs repository was 52% of total faults reported in Bugzilla database. The algorithm adopted in this study has a limitation that faults that is not recorded

in cvs log cannot be collected. In order to make accurate collection of FP modules from source code repository, further research is required.

8. CONCLUSION

This paper showed the training on errors procedure to classify fault-prone software modules using spam filtering technique. In our fault-prone filtering, source code modules were considered as text files and they are applied to the spam filter directly. In order to show the practical usefulness of fault-prone filtering, we conducted an training on errors experiment using source code repositories of Java based open source developments. The result of experiment showed that our approach can classify about 85% of software modules correctly. Furthermore, the result of experience showed that we can improve the prediction accuracy by modifying the threshold of the probability.

One of the most important future works is improvement of prediction accuracy. The current spam filter based approach considers only text information of the source code modules. Although it contributes the speed of the classification and training, some kind of improvements that considers characteristics of source code can be implemented without increasing the execution time so much.

9. ACKNOWLEDGMENTS

Authors would like to thank Mr. Shiro Ikami and Mr. Shuya Nakaichi who made remarkable contribution to the earlier version of this research. Authors also would like to thank Dr. Juichi Takahashi in SONY corporation who gave us many useful advice.

Authors would like to thank anonymous reviewers who gave us many appropriate corrections and suggestions to the paper. Authors also would like to express great thanks to Professor Claes Wohlin as a shepherd of our revision.

Moreover, authors would like to express their thanks to the developers of CRM114 classifier. Without the CRM114, this work cannot be established. Finally, authors also thank to the developers of eclipse who make the repository of eclipse available for research.

10. REFERENCES

[1] 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.

[2] *bogofilter*. http://bogofilter.sourceforge.net/.

[3] 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.

[4] S. Chhabra, W. S. Yerazunis, and C. Siefkes. Spam filtering using a markov random field model with variable weighting schemas. In *Proc. of Fourth IEEE International Conference on Data Mining (ICDM 2004)*, pages 347–350, 2004.

[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] P. Graham. *Hackers and Painters: Big Ideas from the Computer Age*, chapter 8, pages 121–129. O'Reilly Media, 2004.

[9] 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.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, 2002.

[11] T. M. Khoshgoftaar and E. B. Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineeering*, 6(4):303–317, 1999.

[12] T. M. Khoshgoftaar and E. B. Allen. Controlling overfitting in classification tree models of software quality. *Empirical Software Engineering*, 6(1):59–79, 2001.

[13] T. M. Khoshgoftaar, E. B. Allen, and J. Deng. Using regressin trees to classify fault-prone software modules. *IEEE Transactions on Reliability*, 51(4):455–462, 2002.

[14] T. M. Khoshgoftaar and N. Seliya. Software quality classification modeling using SPRINT decision tree algorithm. In *Proc. of 14th International Conference on Tools with Artificial Intelligence*, pages 365–374, 2002.

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

[16] T. M. Khoshgoftaar, R. Shan, and E. B. Allen. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. In *Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE'00)*, pages 301–310, 2000.

[17] 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.

[18] 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)*, page 4, 2007.

[19] *NASA's Metrics Data Program*. http://mdp.ivv.nasa.gov/.

[20] *POPFile*. http://popfile.sourceforge.net/.

[21] 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.

[22] 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.

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

[24] 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.

[25] *SpamAssassin*. http://spamassassin.apache.org/index.html.

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