Spam Filter Based Approach for Finding Fault-Prone Software Modules

Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, JAPAN
o-mizuno@ist.osaka-u.ac.jp

Abstract

Because of the increase of needs for spam e-mail detection, the spam filtering technique has been improved as a convenient and effective technique for text mining. We propose a novel approach to detect fault-prone modules in a way that the source code modules are considered as text files and are applied to the spam filter directly. In order 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 classify more than 75% of software modules correctly.

1. Introduction

Early detection of faulty software modules is of importance for both reduction of development cost and assurance of software quality. Predicting fault-proneness have been conducted so far by many researchers. Munson and Khoshgoftaar used software complexity metrics and the logistic regression analysis to detect fault-prone modules [10]. Basili et al. also used logistic regression for detection of fault-proneness using object-oriented metrics [2]. Fenton et al. proposed a Bayesian Belief Network based approach to calculate the fault-proneness [7]. 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. Metrics based approaches, however, usually depend on the language, development environment, and so on. In addition, collecting such metrics and estimating values of parameters requires additional efforts.

This paper introduces a new idea to detect fault-prone modules. The idea is inspired from the spam e-mail filtering technique. It is said that most e-mail messages on the Internet are spam. Such explosive increase of spam e-mail triggered development of a lot of spam filtering techniques.

We thus tried to apply spam filtering technique to the fault-prone detection, and call our approach "fault-prone filtering." In the fault-prone filtering, we consider a software module[1] as an e-mail message, and assume that all of software modules belong to either fault-prone(FP) or not-fault-prone(NFP). After learning of existing FP and NFP modules, we can classify a new module into either FP or NFP by applying 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 but just apply source code module to the filter. The text classification does not depend on language or development environment but depend on the past history of the development.

2. Overview of Fault-Prone Filtering

2.1. Basic Idea

The basic idea of fault-prone filtering is inspired from spam mail filtering. In the spam mail filtering, the spam filter first learns both spam and ham (non-spam) e-mail messages from 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 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.

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

[1] For example, we consider a software module as function, method, class, and so on.

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

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 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.

2.2. Procedure of Fault-Prone Filtering

In order to apply our approach to data from source code repository, we implemented tools named "FPLearner" and "FPClassifier" for learning and classifying software modules, respectively. The procedure of fault-prone filtering is summarized as follows:
(1) Prepare a set of fault-prone modules, M_{FP}, and a set of non fault-prone modules, M_{NFP}, for the target project.
(2) By learning M_{FP} and M_{NFP}, FPLearner construct corpuses for both FP and NFP modules.
(3) For a new module, m_{new}, we apply FPClassifier to m_{new} and get probability that m_{new} is fault-prone.
(4) Classify m_{new} into either FP or NFP by the probability and a pre-defined threshold.

2.3. Classification Techniques

In this study, we used "CRM114" spam filtering software [4]. The reason why we used CRM114 was its versatility and accuracy. In this experiment, we used the following 3 classification strategies built in CRM114 to evaluate effectiveness of our proposed approach.
Sparse Binary Polynomial Hash Markov model (SBPH): SBPH is the default classification model used in CRM114. It is an extension of Bayesian classification, mapping features in the input text into the Markov Random Field [3]. In this model, tokens are constructed from combinations of 5 words (5-grams) in a text file[2].
Orthogonal Sparse Bigrams Markov model (OSB): OSB is a simplified version of SBPH. It uses a subset of tokens created in the SBPH model so that tokens have exactly 2 words. This decreases both memory consumption of learning and time of classification.
Simple Bayesian model (BAYES): BAYES is a simplified version of SBPH, since it uses only single words as tokens. This model thus considered to be identical to the classical Bayesian classification.

These classifiers have both merits and demerits. In order to investigate the applicability to FP filtering, we compare accuracy of these 3 strategies when they are applied to FP filtering in the experiment.

Table 1. Target projects

Name	argoUML	eclipse BIRT
Type of faults	Bugs	
Status of faults	Resolved, Verified, Closed	
Resolution of faults	Fixed	
Severity	N/A	blocker, critical, major, normal
Priority of faults	all	
Total # of faults	1058	4708

3. Experimental Application

3.1. Target Projects

For the experiment, we selected open source project that can track faults. For this reason, we selected two projects, "argoUML project [1]" and "eclipse BIRT [6]".

Table 1 shows the context of the target projects. Both projects are developed in Java language, and revisions are maintained by concurrent version control system (cvs). The source repository of argoUML is prepared one for the use of Mining Challenge in Mining Software Repository workshop in 2006 [5]. As for the Eclipse BIRT, an archive of repository was obtained from official web-site at 27th November, 2006. Fault reports are 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 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. As for the Eclipse BIRT, the severity of faults are also specified. Faults with "blocker", "critical", "major", and "normal" are collected in this experiment.

3.2. Collecting Fault-Prone Modules

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 [8].

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.

[2] The number of words is determined in CRM114 heuristically.

Table 2. Result of FPFinder for target projects

Name	argoUML	eclipse BIRT
# of faults found in cvs log	396 (37% of total)	1973 (42% of total)
# of FP ($\|M_{\mathrm{FP}}\|$)	1093	9547
# of NFP ($\|M_{\mathrm{NFP}}\|$)	20219	86770

We then extracted FP modules from source code by the following procedure. This procedure is based on an algorithm shown by Sliwerski et al. [11]. At first, we collected the following information from bug database of a target project such as Bugzilla.

F: A set of faults found in bug database.

f_i: Each fault in F.

$date(f_i)$: Date in which a fault f_i is reported.

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 classes $C_{\mathrm{FaultFixed}}$ in which the fault has just been fixed by checking all revision logs.
2. Extract modules $M_{\mathrm{FaultFixed}}$ in classes $C_{\mathrm{FaultFixed}}$.
3. For each module m in $M_{\mathrm{FaultFixed}}$, append m to M_{FP} if m is unmodified since $date(f_i)$.
4. Extract modules M_{AllRev} in all revision.
5. For each n in M_{FP}, track back older revisions of n and append found older revisions of n to M_{FPold}
6. $M_{\mathrm{NFP}} = M_{\mathrm{AllRev}} - M_{\mathrm{FPold}} - M_{\mathrm{FP}}$.

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 M_{FP} and M_{NFP}.

The result of FPFinder is shown in Table 2. Number of faults found in cvs log of argoUML is 396. It is 37% of total reported faults in the bug database. As for Eclipse BIRT, 1973 faults are found in cvs log and it was 42% of total. The execution time of FPFinder for eclipse BIRT was 1 hour on MacPro with Xeon 5100 2.66GHz processor.

Currently, FPFinder cannot find all FP modules in the cvs repository since not all faults are commented in the cvs log. This is one of the large limitation of our approach now, and we are trying to overcome it.

3.3. Application of FPClassifier

The number of extracted NFP modules becomes extremely larger than the number of FP modules. Since it is known that imbalanced data set affects the result of prediction [9], we have to make data set to be balanced before applying FPClassifier.

To do so, the number of NFP modules should be reduced. By applying random sampling method, we randomly chose a subset of NFP modules and let them be M'_{NFP}. The number of modules in M'_{NFP} for argoUML is 2022, and the number of modules in M'_{NFP} for eclipse BIRT is 10413.

Then, we performed 10-fold cross validation using modules in M_{FP} and M'_{NFP} by 3 classifiers in subsection 2.3.

Table 3. Legend of experimental result

Target Model		Predicted NFP	 FP
Actual	NFP	N_1	N_2
	FP	N_3	N_4

3.4. Result of Experiment

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 Type-I error. On the contrary N_3 shows the number of modules that are predicted as NFP but are actually FP. N_3 is called Type-II error. 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. For evaluation purpose, we used three metrics: accuracy, recall, and precision. Accuracy shows the ratio of correctly predicted modules to entire modules and it is defined as $(N_1+N_4)/(N_1+N_2+N_3+N_4)$. Recall is the ratio of modules correctly predicted as FP to number of entire modules actually FP and defined as $N_4/(N_3+N_4)$. Precision is the ratio of modules correctly predicted as FP to number of entire modules predicted as FP, and defined as $N_4/(N_2 + N_4)$.

Table 4 shows results of 10-fold cross validation using 3 classifiers for argoUML and eclipse BIRT. The time needed for the experiment of OSB classifier in eclipse BIRT was 11 hours 4 minutes, and this indicates that 19960 modules are learned and classified in this period. That is, it needs 1.98 seconds for learn and classify 1 module in average. The result of evaluation by accuracy, recall, and precision is shown in Tables 5 and 6.

4. Discussion

The result for argoUML shows that the SBPH classifier achieved the best accuracy among 3 classifiers. OSB is the second best, and BAYES is the third. On the other hand, the result for eclipse BIRT shows that OSB is the best, SBPH is the second, and BAYES is the third.

As for SBPH, from a viewpoint of balance of three evaluation metrics, accuracy, precision, and recall, it can be considered the most balanced classifiers among 3. In particular, for the experiment of argoUML, SBPH achieves the best accuracy, 0.797. This implies that almost 80% of all modules

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

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

argoUML		Predicted	
SBPH		NFP	FP
Actual	NFP	1721	301
	FP	331	762

argoUML		Predicted	
OSB		NFP	FP
Actual	NFP	1366	656
	FP	158	935

argoUML		Predicted	
BAYES		NFP	FP
Actual	NFP	1291	731
	FP	151	942

eclipse BIRT		Predicted	
SBPH		NFP	FP
Actual	NFP	8055	2358
	FP	2649	6898

eclipse BIRT		Predicted	
OSB		NFP	FP
Actual	NFP	6833	3580
	FP	1283	8264

eclipse BIRT		Predicted	
BAYES		NFP	FP
Actual	NFP	1921	8492
	FP	217	9330

Table 5. Evaluation metrics for argoUML

Classifier	Precision	Recall	Accuracy
SBPH	0.717	0.697	0.797
OSB	0.587	0.855	0.739
BAYES	0.563	0.861	0.717

Table 6. Evaluation metrics for eclipse BIRT

Classifier	Precision	Recall	Accuracy
SBPH	0.745	0.723	0.749
OSB	0.698	0.866	0.756
BAYES	0.524	0.977	0.564

classified correctly. SBPH also achieves 75% of accuracy in eclipse BIRT experiment.

As for OSB, the result shows a bit less accuracy than SBPH as its definition implies. Especially, the value of recall for OSB is relatively high in both experiments (that is, 0.855 and 0.866 for argoUML and eclipse BIRT, respectively). It indicates that this method merely misclassified actually FP modules as NFP. This is a big advantage of OSB classifier. Unlike BAYES classifier to be mentioned below, OSB has relatively good accuracy. It is thus considered that OSB is good for a situation that FP modules must not be missed.

The accuracy of BAYES classifier was the worst accuracy among 3. Although the recall is extremely high in both experiments, it also indicates that about most of actual NFP modules are misclassified as FP. In other words, BAYES predicts most modules as FP, and thus it is not acceptable. The comparison of 3 classifiers is summarized as follows:

SBPH The best accuracy, better precision and recall.
OSB Better accuracy, fair precision, and better recall.
BAYES Poor accuracy, poor precision, and the best recall.

From experiments, we can confirm that the fault-prone filtering works correctly for two projects. It is also shown that SBPH is relatively good accuracy to predict FP modules.

5. Conclusion

This paper proposed an approach to classify fault-prone software modules using spam filtering technique. In our approach, source code modules were considered as text files and they are applied to the spam filter directly. We conducted an experiment using source code repositories of Java based open source developments. The result of experiment showed that our approach can classify over 75% of software modules correctly.

As future works, we are now trying to perform experiments assuming an actual situation in a development project have to be done. It will show practical advantage of our approach. Additionally, further investigation of misclassified modules will contribute to improvement of accuracy.

References

[1] *ArgoUML Project.* http://argouml.tigris.org/.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object oriented metrics as quality indicators. *IEEE Trans. on Software Engineering*, 22(10):751–761, 1996.

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

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

[5] S. Diehl, H. Gall, and A. E. Hassan, editors. *Proc. 2006 International Workshop on Mining Software Repositories, MSR 2006*. ACM, 2006.

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

[7] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. on Software Engineering*, 25(5):675–689, 1999.

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

[9] M. Kubat and S. Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In *Proc. 14th Intl Conf. on Machine Learning*, pages 179–186, 1997.

[10] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. on Software Engineering*, 18(5):423–433, 1992.

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