Noname manuscript No.
(will be inserted by the editor)

Fault-Prone Module Detection Using Large-Scale Text Features Based on Spam Filtering

Hideaki Hata · Osamu Mizuno · Tohru Kikuno

Received: date / Accepted: date

Abstract This paper proposes an approach using large-scale text features for fault-prone module detection inspired by spam filtering. The number of every text feature in the source code of a module is counted and used as data for training detection models. In this paper, we prepared a naive Bayes classifier and a logistic regression model as detection models. To show the effectiveness of our approaches, we conducted experiments with five open source projects and compared them with a well-known metrics set, thereby achieving higher detection results. The results imply that large-scale text features are useful in constructing practical detection models, and measuring sophisticated metrics is not always necessary for detecting fault-prone modules.

Keywords fault-prone module · large-scale · text feature · spam filtering · text mining · software repository

1 Introduction

Fault-prone module detection is one of the most traditional and important areas in software engineering. Once fault-prone modules are detected at an early stage of development, developers can take more careful notice of the detected modules. Furthermore, keeping track of fault-prone modules is useful in preventing the injection of additional faults. Various studies have been done in the detection of fault-prone modules (Briand et al. 2002; Denaro and Pezze 2002; Guo et al. 2003; Khoshgoftaar and Seliya 2004; Bellini et al. 2005; Seliya et al. 2005; Nagappan et al. 2006; Menzies et al. 2007). Most of these studies 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.

Several studies suggest that there is no best subset of metrics that enables a fault-prone module detector to perform a perfect detection (Nagappan et al. 2006; Menzies et al. 2007). Nagappan et al. (2006) advised not using complexity metrics without validating them for a

Graduate School of Information Science and Technology, Osaka University
Tel.: +81-6-6879-4538
Fax: +81-6-6879-4539
E-mail: {h-hata, o-mizuno, kikuno}@ist.osaka-u.ac.jp

target project. Menzies et al. (2007) concluded that if there is metrics subset appropriate for a particular domain, all available metrics can be used to construct detection models. They also insisted that how metrics are used to build predictors is much more important than which particular metrics are used. However, it is uncertain how many metrics should be collected. Should we carefully design effective metrics for each domain one by one?

Different from generalized sophisticated metrics, more concrete and small granularity of possible cause of faults are also studied. Fowler and Beck (1999) introduced 22 software structures as problematic code, which they called “bad smells”. Mäntylä et al. (2003) presented a subjective taxonomy that categorizes similar bad smells. In addition, they empirically showed correlations between the bad smells. Pan et al. (2009) defined 27 bug fix patterns. Their studies of open source projects showed that the method call and *if*-related bug fix patterns commonly appear. However, software structures in these patterns that introduce bugs do not always cause bugs. Though there are bug fix structure patterns, a bug-introducing change may be project-specific, package-specific, or other environment-specific. Livshits and Zimmermann (2005) tried to find out application-specific error patterns that are concrete method code patterns. Mileva and Zeller (2008) tried to detect project-specific deletion patterns. They looked for code smell patterns on a fine granularity level.

To capture such code smell patterns on a fine granularity level for a fault-prone module detection model, we have introduced a spam filtering based approach to detect fault-prone modules (Mizuno et al. 2007; Mizuno and Kikuno 2007). In spam filtering, a classifier is trained with large-scale text features from both spam and non-spam mails. Then, an incoming mail is classified into either spam or non-spam. The Bayesian spam filtering technique was introduced in 1998 at first as a scholarly publication by Sahami et al. (1998). The model is a well-studied Bayesian model. Since the usefulness of Bayesian theory for spam filtering has been recognized recently, most spam filtering tools implement Bayesian theories. Consequently, the accuracy of spam detection has improved dramatically. This technique has been studied to meet the needs of the spam mail problem, that is, spam filtering systems should be able to automatically adapt to the variable characteristics of spam mails. Moreover, the systems need to be personalized to the user’s needs. This framework is based on the fact that spam e-mails usually include particular patterns of words or sentences. From the viewpoint of source code, similar situations usually occur in faulty software modules. That is, similar faults may occur in similar contexts. Inspired by the spam filtering technique, we tried to apply text-mining techniques to fault-proneness detection. In fault-prone module detection, we treat a software module as an e-mail message, and classify all software modules into either fault-prone (FP) or non-fault-prone (NFP).

This approach means that the numbers of particular text features in a module are regarded as one of its metrics. Numbers of these features are very large-scale. In previous work (Mizuno et al. 2007; Mizuno and Kikuno 2007), we conducted a comparative study of metrics-based methods only on a survey of research papers. In this paper, we prepared an experimental environment for comparative study and conducted a fair comparison with metrics-based methods, which are often used in the literature. In addition to a naive Bayes classifier, we also adopt a logistic regression model for large-scale text features. For replication of the experiment, we adopted the WEKA data mining toolkit (Witten and Frank 2005).

In summary, this paper contributes to the following:

- Investigating an approach using large-scale text features to detect fault-prone modules.
- Conducting fair comparative experiments with a metrics-based approach.

Two experiments with five open source projects were conducted to show the effectiveness of using large-scale text features compared with using well-known metrics. The results showed that both a naive Bayes classifier and a logistic regression model constructed with the large-scale text features acted as an almost equivalent or better fault-prone module detector.

The rest of this paper is organized as follows: Section 2 introduces related work. Section 3 describes the detection methodology of our study. The detailed data for experiments and experimental design are described in Section 4. Experimental results and analysis are shown in Section 5. Section 6 addresses threats to the validity of this study. Finally, Section 7 summarizes this study.

2 Related Work

Much research on the detection of fault-prone software modules has been carried out so far. This section introduces three approaches: traditional complexity metrics analysis, historical analysis, and the text mining approach.

2.1 Complexity Metrics

A number of software metrics related to program attributes such as lines of code, complexity, frequency of modification, coherency, and coupling, for example, have been proposed. For detection of fault-prone modules, such metrics are treated as explanatory variables and fault-proneness is considered an objective variable. Then mathematical models are constructed from those metrics.

Basili et al. (1996) validated object-oriented metrics proposed by Chidamber and Kemerer (1994), called the CK metrics suite, for the first time. They used these metrics and constructed logistic regression models for detection of fault-prone modules. Gyimóthy et al. (2005) also used the CK metrics suite and constructed detection models for fault-prone modules on larger size projects. Menzies et al. (2007) used metrics obtained from a NASA repository including Halstead (Halstead 1977), McCabe (McCabe 1976), and lines of code. Their experimental results show that naive Bayes classifiers work well with these metrics.

In contrast to these studies, we do not measure complexity metrics. Fault-prone module detection models are constructed with text features extracted from source code. Since we used large-scale text features, the number of explanatory variables are larger compared with traditional complexity metrics based approaches.

2.2 Historical Metrics

Graves et al. (2000) have studied software change history and found that if modules were changed many times, the modules tended to contain faults. In addition, they found that if modules had not changed for one year, the rate of the modules containing faults would be low. Nagappan and Ball (2005) examined code churn, which is a measure of the amount of code change, and showed that relative code churn is highly predictive of defect density. Śliwerski et al. (2005) computed the risk of code locations based on their observation that "risky to change" is different from "frequently fixed". Ostrand et al. (2005) showed that fault and modification history of the file from previous release could be predictive of faults in the next release of a system.

Schröter et al. (2006) have studied the correlation with past failure component history and failure-prone components. They analyzed SCM repositories and bug-tracking systems to extract the failure component's usage pattern, and applied some prediction models to compare the results. They conclude that the support vector machine yields the best predictive power. Though they examined general failures, Neuhaus et al. (2007) focused specifically on vulnerabilities, and obtained higher precision and recall values. Li and Zhou (2005) paid attention to implicit, undocumented programming rules. They extracted programming patterns and detected violations in source code as potential faults.

Hassan and Holt (2005) computed the ten most fault-prone modules after evaluating four heuristics: most frequently modified, most recently modified, most frequently fixed, and most recently fixed. Kim et al. (2007) have tried to predict the fault density of entities using previous faults localities based on the observation that most faults do not occur uniformly. Ratzinger et al. (2008) investigated the interrelationship between previous refactoring and future software defects. Williams and Hollingsworth (2005) have shown that source code repository data can improve static analysis tools. Livshits and Zimmermann (2005) combined mining revision history and dynamic analysis. As a result, they discovered application-specific patterns.

These studies show that there are more concrete and project-specific possible causes of faults. This insight motivates us to propose our approach. Historical metrics enables us to capture project-specific patterns, which is useful in predicting the future. The difference in these studies and our approach is that we do not intend to create sophisticated historical metrics.

2.3 Text Mining

Aversano et al. (2007) trained prediction classifiers with a weighted-term vector created from text belonging to software changes. They used variables, names, language keywords etc. as terms. They concluded that the K-Nearest Neighbors classifier yielded a significant trade-off between precision and recall. Kim et al. (2008) introduced a change classification technique. They gathered features from source code text and other meta data, and applied them to the Support Vector Machine to predict buggy changes. They obtained 78 percent accuracy and a 60 percent buggy change recall on average. Madhavan and Whitehead Jr. (2007) have implemented the change classification tools as an Eclipse plugin.

These text mining approach has some desirable points, such as:

– Independence from programming languages
– Flexibility in the granularity of a unit
– No need of semantic information

Though Aversano et al. (2007) and Kim et al. (2008) used text features extracted only from software changes, we target entire text features in source code. In addition, although these two studies conducted only 10-fold cross validation, we conduct not only 10-fold cross validation but also evaluate the detection of post-release fault-prone modules.

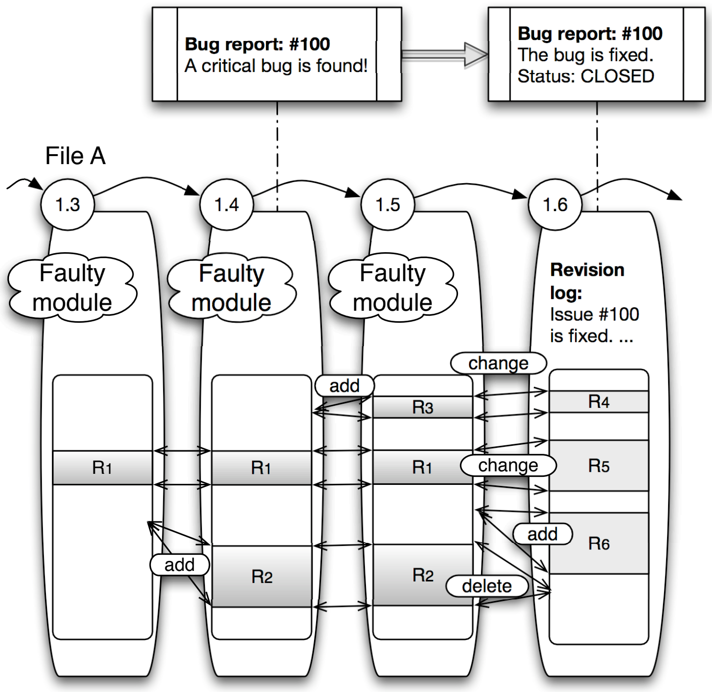


Fig. 1 Identify faulty modules on one file

3 Detection Methodology

3.1 Feature Extraction

Text features are extracted from source code that is removed comment. This means that everything except for comment words separated by space or tab can be treated as a feature. The number of each text feature is counted per module. For replication of experiment, the WEKA data mining toolkit (Witten and Frank 2005) is used in this paper. To extract features properly, every variable, method name, function name, keyword, and operator connecting without a space or tab is separated. Since using all features requires much time and memory, the approximate number of features used can be determined by setting options. We set $5,000$ in this paper[1]. This option is intended to discard other, less useful features. These text features can be regarded as one of the metrics **Num**($term_i$), where $term_i$ represents ith text features. Text feature metrics are very large-scale compared with for example the CK metrics suite proposed by Chidamber and Kemerer (1994).

3.2 Faulty Module Identification

We identify faulty modules with a algorithm proposed by Śliwerski et al. (2005). In the algorithm, we only get fixed and closed status bug reports from bug-tracking systems. From a bug report with bug b_i, where i represents a bug ID, we extract the following information:

- Reported date $Rdate(b_i)$: date when bug b_i was reported.
- Closed date $Cdate(b_i)$: date when the status changed to 'CLOSED' after bug b_i had been fixed.

For each bug b_i, we perform the following procedure:

1. Find fixed revisions of files related to bug b_i by checking all revision logs made before $Cdate(b_i)$.
2. Perform the 'diff' command on the same file between a fixed revision and a preceding revision.
3. Examine when modified regions are inserted into files. If they are inserted before $Rdate(b_i)$, we can assume that they are bug-introducing regions.
4. Identify as a faulty module if the module contains bug-introducing regions.

Figure 1 illustrates an example of identifying faulty modules with one bug on one file. The revision number of file A is increased from 1.3 to 1.6. When the revision number was 1.4, bug b_{100} was reported. After that, the bug was fixed. Then, we locate faulty modules related to bug b_{100}.

By searching all revision logs, we find a number '100' and a keyword 'fixed' at the log of file A in revision 1.6. We can assume that file A was modified in order to fix bug b_{100}. Then, we perform the diff command between revision 1.5 and 1.6. The diff tool returns a list of regions that differ in the two files. As shown in Figure 1, from revision 1.5 to 1.6, region R_3 was changed to region R_4, region R_1 was changed to region R_5, region R_6 was added, and region R_2 was deleted. As a result, regions R_1, R_2, and R_3, which were in revision 1.5 and not in revision 1.6, are recognized to be modified regions. After examining when the modified regions R_1, R_2, and R_3 are inserted into file A, it is revealed that region R_1 and R_2 had been inserted before bug b_{100} was reported. Therefore modified regions R_1 and R_2 can be assumed to be bug-introducing regions. Since regions R_1 or R_2 spread over revision 1.3 to 1.5, we can identify modules of revision 1.3, 1.4, and 1.5 of file A as faulty modules.

3.3 Detection Models

Regarding text features as metrics **Num(**$term_i$**)**, it is easy to construct well-known detection models. In this paper, we constructed the following two models.

3.3.1 Logistic Regression Model

The multivariate logistic regression model is represented as follows:

$$f(m_1, m_2, ..., m_n) = \frac{e^{C_0+C_1m_1+C_2m_2+,,,+C_nm_n}}{1+e^{C_0+C_1m_1+C_2m_2+,,,+C_nm_n}}$$

where m_i is the value of metrics in a module. If $f(m_1, m_2, ..., m_n) > 0.5$, the module is classified as FP, otherwise, as NFP.

[1] java weka.filters.unsupervised.attribute.StringToWordVector -C -W 5000

Table 1 Calculated metrics

Metrics	Description	
LOC	Lines of code	
WMC	CK metrics suite	Weighted methods per class
DIT		Depth of inheritance tree
NOC		Number of children
CBO		Coupling between object classes
RFC		Response for class
LCOM		Lack of cohesion on methods
ADD	Churn metrics	# of added lines
CHG		# of changed lines
FIX	Fixed or not	

3.3.2 Naive Bayes Classifier

The naive Bayes classifier classifies a module as follows:

$$\operatorname*{argmax}_{C \in \{FP, NFP\}} P(C) \prod_{i=1}^{n} P(m_i|C)$$

Menzies et al. (2007) reported that defect predictors using naive Bayes achieved standout good results compared with OneR, J48 in their experiment using the WEKA.

4 Evaluation Settings

To show the effectiveness of using large-scale text features, experiments were conducted. In the experiments, we targeted Java programming language.

4.1 Compared Features

To show the effectiveness of our proposal, we compared our proposal with software metrics in experiments. We collected the CK metrics suite proposed by Chidamber and Kemerer (1994). This metrics suite is collected with an implemented tool also used by Higo et al. (2008). In addition, we collected the code churn, the amount of code change (Layman et al. 2008), change history (Kim et al. 2007), and the LOC of each module. Table 1 shows all collected metrics in this paper.

4.2 Data for experiments

For the experiment, we selected open source software projects in which we can track faults. For this reason, we targeted five projects in Eclipse: Business Intelligence and Reporting Tools (BIRT), Eclipse (ECLP), Eclipse Modeling Project (MODE), the Test and Performance Tools Platform (TPTP), and the Eclipse Web Tools Platform (WTP) [2]. Table 2 shows

[2] http://www.eclipse.org/

Table 2 Target project information

Project	Release 1			Release 2		
	Release	(Date)	Total LOC	Release	(Date)	Total LOC
BIRT	2.1	(2006-06-30)	768K	2.2.0	(2007-06-29)	1,135K
ECLP	3.2	(2006-06-29)	2,617K	3.3	(2007-06-28)	2,588K
MODE	Callisto	(2006-06-30)	1,730K	Europa	(2007-06-29)	2,191K
TPTP	4.2.0	(2006-06-30)	718K	4.4.0	(2007-06-29)	722K
WTP	1.5	(2006-06-30)	1,432K	2.0	(2007-06-29)	2,338K

Table 3 Result of module collection

Project	Release 1			Release 2		
	# of faulty modules		Total modules	# of faulty modules		Total modules
BIRT	227	(8.6%)	2,645	291	(8.2%)	3,563
ECLP	376	(4.5%)	8,429	236	(3.2%)	7,351
MODE	36	(0.6%)	5,649	44	(0.6%)	7,049
TPTP	792	(28.2%)	2,811	366	(15.8%)	2,310
WTP	183	(2.5%)	7,336	133	(1.7%)	7,996

the information of each target project. These projects are written in Java language, and revisions are maintained by CVS. The source repository of CVS used in this study was uploaded on the Eclipse project Web site, and was obtained on the 6th January, 2009. We treated a Java class file in each revision as a software module.

We also obtained bug reports from the bug databases of each project. We extracted faults from the bug database (Bugzilla) under the following conditions. The type of these faults is “bugs”; therefore, these faults do not include any enhancements or functional patches. The status of faults is either “resolved”, “verified”, or “closed”, and the resolution of faults is “fixed”. This means that the collected faults have already been fixed and have been resolved, and thus fixed revisions should be included in the entire repository. The severity of the faults is either “BLOCKER”, “CRITICAL”, or “MAJOR” in order to remove trivial bugs. Herraiz et al. (2008) categorized these severity categories as important and the others without ENHANCEMENT as non-important. Using our faulty modules collection tool, we collected both faulty and not faulty modules from these five projects. The result of the module collection is shown in Table 3.

4.3 Design of Experiments

Using collected data shown in Table 3, we conducted the following two experiments.

1. Ten-fold cross validation
   For 10-fold cross validation, we used release 1 data only. The 10-fold cross validation can show relatively fair results for a given data set. However, it cannot take into account important features such as the order of construction of the modules.
2. Fault-prone module detection on post-release
   Here, we used both release 1 data and release 2 data. Fault-prone modules are detected on release 2 data using detection models trained with release 1 data. On the release 2 data, we evaluate the detection performance.

To show the effectiveness of using large-scale text features, the same two experiments were also conducted with well-known software metrics as shown in Table 1. Generally

Table 4 Legend of classification matrix

		Classified	
		NFP	FP
Actual	not faulty	True negative (TN)	False positive (FP)
	faulty	False negative (FN)	True positive (TP)

speaking, the performance of fault-prone module detection varies according to the combination of these metrics used in a detection model. In order to find the best metrics subset for the release 1 data, we prepared all ($= 2^{10} = 1,024$) combinations of metrics shown in Table 1. Then, we performed 10-fold cross validation for each combination, and obtained the best combination with the highest evaluation measurement. This procedure is iterated for all projects. Once we get the best combination of compared features, we construct a detection model using the best combination of metrics and the release 1 data. Next, we apply the constructed model to the release 2 data.

4.4 Performance Evaluation

For the evaluation of the experiment, we define several measures. Table 4 shows a legend of the classification result matrix. True negative (TN) shows the number of modules that are classified as not fault-prone, and are actually not faulty. False positive (FP) shows the number of modules that are classified as fault-prone, but are actually not faulty. On the contrary, false negative (FN) shows the number of modules that are classified as non-fault-prone, but are actually faulty. Finally, true positive (TP) shows the number of modules that are classified as fault-prone which are actually faulty.

To evaluate the results, we prepared four measures: accuracy, recall, precision, and F_1. The accuracy rate shows the ratio of correctly predicted modules to entire modules and is defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TN + FP + FN + TP}$$

Recall is the ratio of modules correctly classified as fault-prone to the number of entire faulty modules Recall is defined as follows:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Precision is the ratio of modules correctly classified as fault-prone to the number of entire modules classified fault-prone. Precision is defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP}$$

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

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

Table 5 The best subset of metrics for naive Bayes models

Project	Subset of metrics
BIRT	LOC,CHG,DIT,CBO,NOC
ECLP	FIX,CHG,WMC,LCOM
MODE	CHG,CBO,RFC
TPTP	LOC,FIX,ADD,WMC,DIT,CBO,LCOM,RFC
WTP	FIX,WMC,DIT,LCOM,NOC

Table 6 Regression coefficients of selected metrics for logistic regression models

Project	LOC	FIX	ADD	CHG	WMC
BIRT	0.0004	0.920			
ECLP	-0.001	1.292	-0.005	-0.0003	0.012
MODE				0.002	
TPTP	-0.001	0.776		0.005	0.001
WTP		1.737			0.007

Project	DIT	CBO	LCOM	RFC	NOC
BIRT	-0.079	0.008	-0.0002	0.0006	-0.114
ECLP	-0.053	0.001			
MODE					
TPTP	0.030	0.002	-0.001	0.002	-0.025
WTP	-0.053			0.002	

5 Analysis

5.1 Ten-fold Cross Validation

Table 5 shows the best subset of metrics in each project on naive Bayes models. As described in Section 4.3, each subset of metrics achieved the highest F_1 value with a naive Bayes classifier in each project. Similarly, the best subset of metrics for logistic regression models in each project and the regression coefficient of each selected metrics are seen in Table 6. In Table 6, a blank represents a corresponding metrics not used in a corresponding project. For example, in project WTP, the best subset of metrics for logistic regression models are “FIX”, “WMC”, “DIT”, and “RFC”. Each value in Table 6 is an estimated regression coefficient value. The larger the absolute value of the regression coefficient, the stronger the impact of the metrics on fault-prone modules detection. The used metrics sets are different from each other. From the viewpoint of the regression coefficient value, FIX and DIT are relatively high in used projects.

Table 7 presents the top three text features ordered by positive and negative regression coefficient values of logistic regression models in each project. A positive regression coefficient indicates an increase in the probability of FP, while a negative regression coefficient indicates a decrease in the FP probability. For example, in project BIRT, if there is “pointer” and/or “getObject” in the source code of a module, the FP probability of the module is high. If there is “excel” and/or “Member”, the FP probability is low.

Next, the distribution of the regression coefficient value is investigated. Figure 2 shows the histogram of the regression coefficient value of a logistic regression model in project ECLP. A large regression coefficient means a strong impact of the feature on the FP probability, while a near zero regression coefficient means little impact on the FP probability. As shown in Figure 2, most of the regression coefficient values are near zero. Such distribution

Table 7 Top three text features ordered by positive and negative regression coefficient values of logistic regression models

Project	Positive regression coefficient		Negative regression coefficient	
	Feature	Value	Feature	Value
BIRT	pointer	79.2	excel	-665.1
	getObject	73.9	em	-190.0
	package	71.8	Member	-148.7
ECLP	NavigatorPlugin	21.6	PerformanceTestSetup	-32.6
	launchConfigurations	14.3	AbstractUIPlugin	-18.0
	isBaseType	13.0	removeSelectionChangedListener	-16.8
MODE	org/uml2/2	11.7	0/UML	-12.0
	g1	4.5	getFactory	-6.7
	Factory	3.9	V	-5.3
TPTP	LF	75.6	atts	-153.4
	setTestInvocationId	52.3	scenario	-43.0
	createPlatformResourceURL	49.6	OK_STATUS	-39.5
WTP	Missing	10.0	ArrayCreation	-31.8
	extra	9.5	FieldAccess	-31.8
	COMPILATION_UNIT	8.7	SimpleName	-31.8

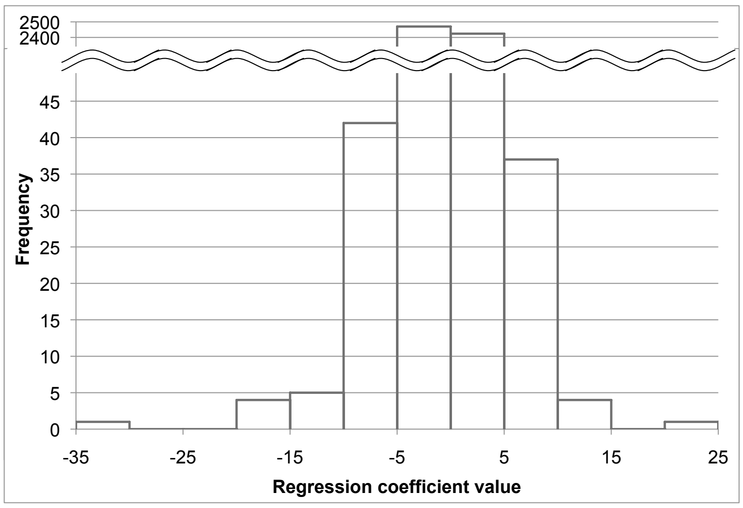
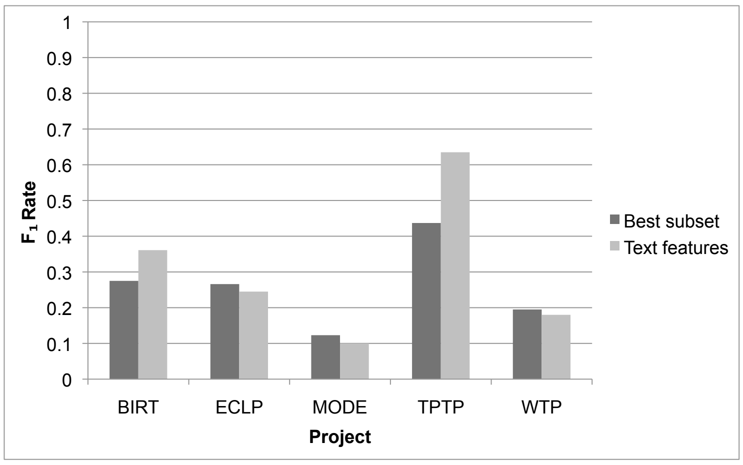


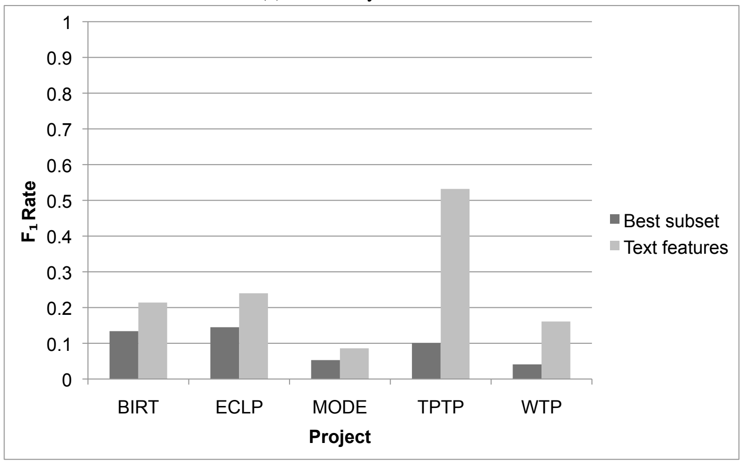
Fig. 2 Histogram of the regression coefficient value of a logistic regression model in project ECLP

of the regression coefficient values is seen in the other projects. These distribution can be interpreted as being able to train logistic regression models without distinguishing a few project-specific useful text features and other not so useful text features. However, there is only one text feature whose corresponding regression coefficient value is zero. Therefore, almost all large-scale text features are needed to construct logistic regression models.

Figure 3 shows the F_1 rate in each project comparing the best subset from ten metrics and text features. Figure 3 (a) is the result of naive Bayes classifiers and (b) is the result of logistic regression models. Table 8 presents the detailed results of 10-fold cross validation. As seen in Figure 3 (a), which shows the results of the naive Bayes classifier, though the F_1 rate of the results using text features are narrowly less than the results using the best subset of metrics in project ECLP, MODE, and WTP, the results using text features are much greater



(a) Naive Bayes classifiers



(b) Logistic regression models

Fig. 3 Comparing F_1 rate of the 10-fold cross validation results

than best subset in project BIRT and TPTP. The results of the logistic regression models, which are shown in Figure 3, illustrate that large-scale text features have a greater capability of fault-prone module detection than a best metrics subset. As shown in Table 8, the best metrics subset achieved a higher precision rate and the text features achieved a higher recall rate.

For example, with the naive Bayes classifiers using text features, the F_1 rates range from 0.100 to 0.635. To explain the difference of the detection performance, Pearson's correlations are calculated between the evaluation metrics and the percentage of faulty modules. Table 9 lists the correlation values. The values show a strong negative correlation for accuracy. This means that if the percentages of faulty modules are low, accuracy rates are high. This is because it is easy to achieve high accuracy with classifying most modules as NFP when the percentages of faulty modules are low since most modules are not faulty. On the contrary, there are strong correlations between the F_1 rate and the percentage of faulty modules except for logistic regression models with a best metrics subset. Logistic regres-

Table 8 Detailed results of the 10-fold cross validation

Project (% of faulty modules)	Detection model	Features	Accuracy	Recall	Precision	F_1
BIRT (8.6%)	Naive Bayes	Best subset	0.902	0.216	0.377	0.275
		Text features	0.806	0.634	0.252	0.361
	Logistic Regression	Best subset	0.917	0.075	0.654	0.134
		Text features	0.732	0.423	0.143	0.214
ECLP (4.5%)	Naive Bayes	Best subset	0.947	0.215	0.346	0.266
		Text features	0.879	0.449	0.169	0.245
	Logistic Regression	Best subset	0.956	0.082	0.585	0.145
		Text features	0.897	0.371	0.177	0.240
MODE (0.6%)	Naive Bayes	Best subset	0.980	0.222	0.085	0.123
		Text features	0.940	0.463	0.056	0.100
	Logistic Regression	Best subset	0.994	0.028	0.500	0.053
		Text features	0.966	0.220	0.054	0.086
TPTP (28.2%)	Naive Bayes	Best subset	0.353	0.891	0.290	0.437
		Text features	0.745	0.779	0.535	0.635
	Logistic Regression	Best subset	0.722	0.056	0.571	0.101
		Text features	0.703	0.594	0.482	0.532
WTP (2.5%)	Naive Bayes	Best subset	0.956	0.213	0.180	0.195
		Text features	0.854	0.623	0.105	0.180
	Logistic Regression	Best subset	0.974	0.022	0.308	0.041
		Text features	0.898	0.383	0.102	0.161

Table 9 Pearson's correlation in evaluation metrics and the percentage of faulty modules

Features	Naive Bayes		Logistic regression	
	Accuracy	F_1	Accuracy	F_1
Best metrics subset	-0.987	0.944	-0.999	0.276
Text features	-0.881	0.978	-0.830	0.975

sion models with the best metrics subset obtained always less than the 0.15 F_1 rate for the five projects. The other combination of detection models and used features revealed that the higher the percentage of faulty modules, the higher the F_1 rates can be achieved. This is because if there are few faulty modules, it is very difficult to detect the faulty modules with only a few false positives and false negatives.

5.2 Detection on Post-Release

Table 10 presents the detailed results of the detection on post-release. Table 11 shows the F_1 rate in each project comparing the best subset from ten metrics and text features. Each value represents the F_1 rate with text features, minus the F_1 rate with the best metrics subset. Therefore, a positive value means that text features overcame the best metrics subset, and a negative value, vice versa. As seen in Table 11 results of the naive Bayes classifier, although the F_1 rate of the results using text features are narrowly less than the results using the best subset of metrics in project ECLP and WTP, the results using text features are much greater than the best subset in project BIRT, MODE, and TPTP. In TPTP especially, the text features achieved almost the 0.15 higher F_1 rate. The results of logistic regression models illustrate how large-scale text features overcame the best metrics subset in every project. As shown in Table 10, the best metrics subset tends to obtain low recall and relatively high precision, and

Table 10 Detailed results of the detection on post-release

Project (% of faulty modules)	Detection model	Features	Accuracy	Recall	Precision	F_1
BIRT (8.6%)	Naive Bayes	Best subset	0.893	0.199	0.279	0.232
		Text features	0.759	0.630	0.196	0.299
	Logistic Regression	Best subset	0.919	0.069	0.513	0.121
		Text features	0.802	0.526	0.213	0.303
ECLP (4.5%)	Naive Bayes	Best subset	0.946	0.191	0.181	0.186
		Text features	0.868	0.461	0.112	0.180
	Logistic Regression	Best subset	0.965	0.089	0.350	0.142
		Text features	0.946	0.557	0.303	0.392
MODE (0.6%)	Naive Bayes	Best subset	0.974	0	0	NaN
		Text features	0.926	0.023	0.002	0.004
	Logistic Regression	Best subset	0.994	0	0	NaN
		Text features	0.965	0.023	0.006	0.009
TPTP (28.2%)	Naive Bayes	Best subset	0.213	0.896	0.156	0.265
		Text features	0.631	0.807	0.276	0.411
	Logistic Regression	Best subset	0.831	0.126	0.397	0.191
		Text features	0.789	0.658	0.402	0.499
WTP (2.5%)	Naive Bayes	Best subset	0.938	0.188	0.061	0.092
		Text features	0.774	0.579	0.043	0.080
	Logistic Regression	Best subset	0.980	0.045	0.171	0.071
		Text features	0.805	0.609	0.052	0.096

Table 11 F_1(text features) - F_1(best metrics subset)

Detection model	BIRT	ECLP	MODE	TPTP	WTP
Naive Bayes	0.067	-0.006	0.004	0.146	-0.012
Logistic regression	0.182	0.393	0.057	0.324	0.148

Table 12 Pearson's correlation in naive Bayes probability and LOC

BIRT	ECLP	MODE	TPTP	WTP
0.136	0.032	0.026	0.007	0.041

text features tend to obtain high recall and low precision, similar to the results of the 10-fold cross validation.

Although the proposed approach using large-scale text features seems to work well, it is questionable whether the FP probability of a module may be strongly influenced by the number of text features in the module. That is, modules whose source code contain lots of text features might be simply detected as FP. Since the number of text features is related to LOC, we computed the Pearson's correlation between the probability yields from the naive Bayes classifiers and the LOC. Table 12 lists the correlation values. As shown in Table 12, every correlation value in the five projects is low. This means that there are weak correlations between the probability yielded from the naive Bayes classifiers and the LOC. Therefore, it can be said that FP probability is not simply derived from naive Bayes classifiers based on the number of text features in a module. In addition, it might be said that code smell patterns related to faults are captured.

6 Threats to Validity

There are four threats to the validity of this study.

Target projects are the Eclipse projects only. This is the external validity threat for generality of data used in the experiments. In general, the Eclipse projects do much better than other open source projects when using machine learning classifiers to predict fault-prone modules. Using other open source projects, different results may be obtained. In addition, industrial projects may lead to different results.

There may be incorrect identifications of faulty and not faulty modules. The algorithm adopted in this study to identify faulty modules has a limitation. For example, faults that are not recorded in logs cannot be collected. Incorrect identifications of training data badly influence the quality of the detection models. In addition, if identifications of test data are incorrect, performance evaluation metrics cannot be calculated properly. To make a complete collection of faulty modules from a source code repository, further research is required.

Specific settings for implementing the approach may influence the detection performance improperly. Because of the limitations of time and memory, we limit the number of text features used in each project to approximate $5,000$. Important text features may be discarded by this setting. In addition, we removed all comments before counting the number of text features. These settings may result in improper detection.

There might be flaws in the design of experiments In order to show the effectiveness of our approach using large-scale text features, we compared our approach with the best subset of well-known metrics including the CK metrics suite. However, these prepared metrics may be not enough. In addition, although we prepared two experiments including (1) 10-fold cross validation and (2) fault-prone module detection on post-release in order to compare fairly, there might be flaws in showing effectiveness. For example, in the (2nd) experiment, every period between release 1 and release 2 is one year. If we vary the period, the results might change.

7 Conclusion

We proposed an approach using large-scale text features for fault-prone module detection. To show the effectiveness of our approach, we conducted two experiments and compared our approach with metrics-based methods by applying it to five open source Java projects in Eclipse, and obtained higher F_1 values. The performance results of spam filtering based approaches implies that:

– Large-scale text features are useful to build a practical model.
– Measuring sophisticated metrics is not always necessary for detecting fault-prone modules.

Constructed models with large-scale text features just detect fault-prone modules. While traditional sophisticated metrics can suggest how a developer should modify modules or what problems are in them, text features do not derive such suggestions. However, since there is no need to collect meaningful module features, applying our approach to projects is easy.

Moreover, large-scale text features approaches have several desirable points as follows:

– They are independent from programming languages.

- We can treat the flexible granularity of a unit as classified modules or as a training set. For example, a method can be treated as a module.
- We do not need semantic information.

Acknowledgements The authors would like to express their thanks to the three anonymous reviewers and the editor for providing insightful and useful suggestions and comments. We also thank the developers of the CRM114 classifier. Without the CRM114, this work could not be conducted. We thank Tatsuya Miyake, Yoshiki Higo, and Katsuro Inoue who implemented a software metrics measurement tool. Finally, the authors also wish to thank the developers of Eclipse who have made the repository of Eclipse available for research.

References

Aversano L, Cerulo L, Grosso CD (2007) Learning from bug-introducing changes to prevent fault prone code. In: Proc. of 9th International workshop on Principles of software evolution, ACM, New York, NY, USA, pp 19–26

Basili VR, Briand LC, Melo WL (1996) A validation of object oriented metrics as quality indicators. IEEE Trans on Software Engineering 22(10):751–761

Bellini P, Bruno I, Nesi P, Rogai D (2005) Comparing fault-proneness estimation models. In: Proc. of 10th IEEE International Conference on Engineering of Complex Computer Systems, pp 205–214

Briand LC, Melo WL, Wust J (2002) Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans on Software Engineering 28(7):706–720

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans on Software Engineering 20(6):476–493

Denaro G, Pezze M (2002) An empirical evaluation of fault-proneness models. In: Proc. of 24th International Conference on Software Engineering, pp 241–251

Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Graves TL, Karr AF, Marron J, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans on Software Engineering 26(7):653–661

Guo L, Cukic B, Singh H (2003) Predicting fault prone modules by the dempster-shafer belief networks. In: Proc. of 18st International Conference on Automated Software Engineering, pp 249–252

Gyimóthy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans on Software Engineering 31(10):897–910

Halstead MH (1977) Elements of Software Science. Elsevier

Hassan AE, Holt RC (2005) The top ten list: Dynamic fault prediction. In: Proc. of 21st IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp 263–272

Herraiz I, German DM, Gonzalez-Barahona JM, Robles G (2008) Towards a simplification of the bug report form in eclipse. In: Proc. of 5th International workshop on Mining software repositories, ACM, pp 145–148

Higo Y, Murao K, Kusumoto S, Inoue K (2008) Predicting fault-prone modules based on metrics transitions. In: Proc. of 2008 workshop on Defects in large software systems, ACM, New York, NY, USA, pp 6–10

Khoshgoftaar TM, Seliya N (2004) Comparative assessment of software quality classification techniques: An empirical study. Empirical Software Engineering 9:229–257

Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A (2007) Predicting faults from cached history. In: Proc. of 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, pp 489–498

Kim S, Whitehead Jr EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans on Software Engineering 34(2):181–196

Layman L, Kudrjavets G, Nagappan N (2008) Iterative identification of fault-prone binaries using in-process metrics. In: Proc. of 2nd International Symposium on Empirical Software Engineering and Measurement, ACM, New York, NY, USA, pp 206–212

Li Z, Zhou Y (2005) PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. of 5th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ACM New York, NY, USA, pp 306–315

Livshits B, Zimmermann T (2005) Dynamine: finding common error patterns by mining software revision histories. ACM SIGSOFT Software Engineering Notes 30(5):296–305

Madhavan J, Whitehead Jr EJ (2007) Predicting buggy changes inside an integrated development environment. In: Proc. of the 2007 OOPSLA workshop on eclipse technology eXchange, ACM New York, NY, USA, pp 36–40

Mäntylä M, Vanhanen J, Lassenius C (2003) A taxonomy and an initial empirical study of bad smells in code. In: Proc. of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp 381–384

McCabe TJ (1976) A complexity measure. In: Proc. of 2nd International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, p 407

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

Mileva YM, Zeller A (2008) Project-specific deletion patterns. In: Proc. of international workshop on Recommendation systems for software engineering, ACM, New York, NY, USA, pp 41–42

Mizuno O, Kikuno T (2007) Training on errors experiment to detect fault-prone software modules by spam filter. In: Proc. of 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp 405–414

Mizuno O, Ikami S, Nakaichi S, Kikuno T (2007) Spam filter based approach for finding fault-prone software modules. In: Proc. of 4th International workshop on Mining software repositories

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proc. of 27th International Conference on Software Engineering, pp 284–292

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proc. of 28th International Conference on Software Engineering, ACM, New York, NY, USA, pp 452–461

Neuhaus S, Zimmermann T, Holler C, Zeller A (2007) Predicting vulnerable software components. In: Proc. of 14th ACM conference on Computer and communications security, ACM, New York, NY, USA, pp 529–540

Ostrand T, Weyuker E, Bell R (2005) Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans on Software Engineering pp 340–355

Pan K, Kim S, Whitehead EJ Jr (2009) Toward an understanding of bug fix patterns. Empirical Software Engineering 14(3):286–315

Ratzinger J, Sigmund T, Gall H (2008) On the relation of refactorings and software defect prediction. In: Proc. of 5th International workshop on Mining software repositories, ACM New York, NY, USA, pp 35–38

Sahami M, Dumais S, Heckerman D, Horvitz E (1998) A bayesian approach to filtering junk e-mail. In: Proc. of AAAI Workshop on Learning for Text Categorization, AAAI Technical Report WS-98-05

Schröter A, Zimmermann T, Zeller A (2006) Predicting component failures at design time. In: Proc. of ACM/IEEE international symposium on Empirical software engineering, ACM, New York, NY, USA, pp 18–27

Seliya N, Khoshgoftaar TM, Zhong S (2005) Analyzing software quality with limited fault-proneness defect data. In: Proc. of 9th IEEE International Symposium on High-Assurance Systems Engineering, pp 89–98

Śliwerski J, Zimmermann T, Zeller A (2005) HATARI: raising risk awareness. In: Proc. of 5th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ACM Press New York, NY, USA, pp 107–110

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? (on Fridays.). In: Proc. of 2nd International workshop on Mining software repositories, pp 24–28

Williams C, Hollingsworth J (2005) Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans on Software Engineering pp 466–480

Witten IH, Frank E (2005) Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, URL /bib/private/witten/Data Mining Practical Machine Learning Tools and Techniques 2d ed - Morgan Kaufmann.pdf