生産と技術　第61巻　第1号（2009）

Fault-proneness Filtering:
スパムフィルタに基づく
不具合混入ソフトウェアモジュールの予測手法

技術解説

水　野　　修*

Fault-Proneness Filtering:
An Approach to Predict Fault-Prone Modules Using Spam Filter

Key Words : Fault-proneness Prediction Spam-filter

不具合を含みそうなソフトウェアモジュール(Fault-prone(FP) モジュール) の検出はソフトウェア工学における重要な問題の１つであり，これまでにも多くの研究が行われてきた．それらの研究の多くはソフトウェアの複雑度メトリクスなどに基づいたモデルによる予測であった．しかし，モデルの構築にはメトリクスの収集環境が必要となるため，そのことも適用を難しくしている．

そこで我々は，ソースコードに対して簡単に適用できるFault-proneモジュールの検出手法として，スパムフィルタに基づいたFault-prone モジュール検出法「Fault-prone フィルタリング」を提案している．この手法はソースコードのみを入力とすることができ，また，全く事前の知識がない状態からでも開発プロジェクトに適用できるという特徴を持つ．本稿では適用実験としてオープンソースソフトウェアeclipseに対して予測を行い，予測精度についての評価を行った．

Prediction of fault-prone software modules has been one of the most classical and important area of software engineering so far. Many approaches has been carried out using software complexity metrics and mathematical models. Such approaches, however, have difficulties in collecting the metrics and constructing mathematical models based on the metrics.

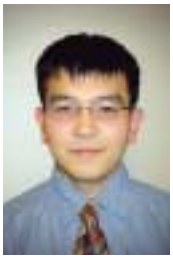
We proposed a novel approach for predicting fault-prone modules using a spam filtering technique, named Fault-prone Filtering. In our approach, fault-prone modules are detected in a way that the source code modules are considered as text files and are applied to the spam filter directly. For the validation of our method, we performed a cross-validation experiment using an open source software project.

1. はじめに

高品質なソースコードの作成はプロダクトの品質向上だけでなくコストの削減にもつながる．コードを作成した時点で不具合を含むかもしれない（Fault-prone, FP）モジュールを特定できれば，早期にバグを除去できるだけでなく，レビュー，デバッグに費やす工数の削減も可能となる．そのため，これまでにもFault-proneモジュールを予測すべく，多くの研究が行われてきた[1, 3, 6, 10－12, 16]．従来の手法では，主にモジュールの複雑さや変更頻度などのソフトウェアメトリクスを用いて予測モデルを構築している．しかし，こうしたソフトウェアメトリクスを測定するためには，メトリクスの測定環境が必要となる．

そこで，我々はソースコードのみを入力として与え，メトリクスなどの測定無しにフォールトプローンなモジュールの予測が可能となる手法を提案する．この手法では，迷惑メール検出に利用されるスパムフィルタで利用される技術をソフトウェアのソースコードに対して適用し，純粋にコードのテキスト情報のみからFPモジュールを予測する．この手法を「Fault-proneフィルタリング」と呼ぶ[13]．

Postini社の調査によると，2006年11月の時点で世界中を流れる電子メールの94％はスパムメールであるとされている[15]．そのため，スパムメールをブロックする技術の開発が進められてきている．初期のスパムフィルタはあらかじめ登録した単語のパターンマッチによるものが主流であったが，この方法ではスパム送信者とのいたちごっこが続くため，根本的な解決にはならなかった．そうした中，Grahamはベイズ識別器によりスパムメールの分別



*Osamu MIZUNO
1973年7月生
大阪大学大学院情報科学研究科博士前期課程修了（1998年）
現在、大阪大学大学院　情報科学研究科情報システム工学専攻　助教　博士（工学）　ソフトウェア工学
TEL：06-6879-4536
FAX：06-6879-4539
E-mail：o-mizuno@ist.osaka-u.ac.jp

が可能であることを示した[9]．このアイディアに触発され，多くのベイズ識別器に基づくスパムフィルタが開発され[2, 14]，ユーザの手元にスパムメールが届く可能性は激減しつつある[8]．本研究ではYerazunisらによって開発されたスパムフィルタCRM 114を用いる[4]．

本稿の以降の構成について述べる．まず，２節では，我々の提案するFault-prone フィルタリングの概要について述べる．３節では実用的な大きさのプロジェクトへの適用実験について述べる．また，４節では関連研究との比較についてまとめる．最後に，５節で本研究のまとめと今後の課題について述べる．

2. Fault-prone フィルタリングの概要

2.1 着想の背景

Fault-prone フィルタリングはスパムメール（迷惑メール）の判別をおこなうスパムフィルタで用いられるテキスト分類フィルタ技術を利用する．スパムフィルタは，過去に受信した電子メール内の単語群を利用して，スパムメールと通常のメールを判別するための辞書を作成する．そして，新たに受信した電子メールについては，ベイズ識別などの技術により，スパムか否かを判定する．学習は随時行われ，辞書は常にその時点の状況を反映したものになるため，新種のスパムメールなどにも柔軟に対応できるとされている．この考えは，スパムメールには特定の単語群や文章が頻繁に含まれている，という事実に基づいている．

我々は，この考え方がソースコード内の不具合についても適用できるのではないかと考えた．もちろん，元々悪意を持って作成されたスパムメールと，意図的ではないがバグが混入したソースコードを全く同じものと見なすのは無理があるかもしれない．しかし，一連のソフトウェア開発においては，同じ開発者が同じ文脈でバグを混入することや，類似の関数やAPI の呼び出しなどにおいてバグを混入してしまうことは良くあることだと考えられる．すなわち，スパムメールの中の特定の単語のように，バグが存在するところには特定のコード片が存在するのではないかと類推した．

2.2 スパムフィルタ: CRM114

本研究ではテキスト分類フィルタとしてCRM114を用いた[5]．主にスパムフィルタとして開発されているが，汎用的な用途，例えば，計算機のログ監視やネットワークのトラフィック監視などにも活用できるとされている．また，現在開発されているメールフィルタの中でも高い予測精度をあげているものの1つである．

CRM114は基本的にはベイズ識別を利用したテキスト分類フィルタであるが，複数の単語を組み合わせたものをトークンと呼び，学習・分類の単位として利用することが大きな特徴である．従来のテキスト分類フィルタは１単語をトークンとしているのに対し，複数単語の組をトークンとすることで，より複雑な学習が可能となっている．本研究では，CRM114のデフォルトの分類手法である“Othogonal Sparce Bigrams Marcov model (OSB)”を使用する．OSBは任意の連続する５単語の組合せのうち，２単語からなるものだけをトークンとする手法である．

OSBによるテキスト処理について以下に簡単に示す．表１は“a = b + 1 ; return a;”という文について，トークンを生成した様子である．OSB ではある単語を起点として５単語からなる単語列に対し，正確に２単語のみを含むもののみを学習・分類の対象として抽出する．なお，本研究ではプログラム言語中の区切り文字をあらかじめ排除するため，“ ; ”は単語群に含まれていない．

表１　OSBで生成されるトークン例

a	=					
a		b				
a			+			
a				1		
	=	b				
	=		+			
	=			1		
	=				return	
		b	+			
		b		1		
		b			return	
		b				a
			+	1		
			+		return	
			+			a
				1	return	
				1		a
					return	a

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

(a)不具合を含む（FP）モジュール m_{FP}

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

(b)不具合を含まない（NFP）モジュール m_{NFP}

図1　FPとNFPモジュールの例

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

図2　新しく作成したモジュール m_{new}

2.3 CRM114による分類の例

この節ではFault-proneフィルタリングがどのように不具合のあるソフトウェアモジュールを検出するのかを，単純な例を用いて説明する．図1(a)と(b) はそれぞれ，不具合を含む（FP）モジュールと含まない（NFP）モジュールの例である．以降ではそれぞれを m_{FP}，m_{NFP} と表記する．fact( ) は与えられた自然数 x に対してその階乗を返すことを意図しているが，図1(a) の実装では −−x とすべきところを ++x と誤記しているため，正常に動作しない．図1(b) はその不具合を取り除いた状態である．この2つのモジュールのみが辞書に学習された時点で，図2に示すモジュール m_{new} が新たに作成されたとし，このモジュールが不具合を含む確率を計算することを考える．なお，モジュール m_{new} は与えられた正整数 y に対してその総和を求めるつもりであるが，m_{FP} と同様に本来 −− とすべきところを++としている．

まず，m_{FP} と m_{NFP} をそれぞれFP, NFPとして学習する．この時，CRM114によってそれぞれのモジュールについて図3(a), (b) に示すようなトークンの集合 T^{FP}, T^{NFP} が生成される．m_{FP} から生成されたトークンの集合 T^{FP} はFault-proneモジュールの特徴としてFP　辞書に格納される．同様に，m_{NFP} のトークンの集合 T^{NFP} は，NFP 辞書に格納される．

新たなモジュールが与えられると，その時点で辞書に学習されている全てのトークンとのマッチング

public	int
public	fact
public	x
int	fact
int	int
int	x
int	return
fact	int
fact	x
fact	return
int	==
x	return
x	x
x	==
x	1
return	x
return	==
return	1
return	?
x	?
x	:
==	1
==	?
==	:
==	x
1	?
1	:
1	x
1	*
?	:
?	x
?	*
?	fact
:	x
:	*
:	fact
:	++
x	*
x	fact
x	++
*	fact
*	++
*	x
fact	++
++	x

(a)m_{FP} から生成されるトークン T^{FP}

public	int
public	fact
public	x
int	fact
int	int
int	x
int	return
fact	int
fact	x
fact	return
int	==
x	return
x	x
x	==
x	1
return	x
return	==
return	1
return	?
x	?
x	:
==	1
==	?
==	:
==	x
1	?
1	:
1	x
1	*
?	:
?	x
?	*
?	fact
:	x
:	*
:	fact
:	--
x	*
x	fact
x	--
*	fact
*	--
*	x
fact	--
--	x

(b)m_{NFP} から生成されるトークン T^{NFP}

public	int
public	sigma
public	y
int	sigma
int	int
int	y
int	return
sigma	int
sigma	y
sigma	return
int	==
y	return
y	y
y	==
y	1
return	y
return	==
return	1
return	?
y	?
y	:
==	1
==	?
==	:
==	y
1	?
1	:
1	y
1	+
?	:
?	y
?	+
?	sigma
:	y
:	+
:	sigma
:	++
y	+
y	sigma
y	++
+	sigma
+	++
+	y
sigma	++
++	y

(c)m_{new} から生成されるトークン T^{new}

図3　各モジュールに対して生成されるトークン

がとられ，確率の計算が行われる．図3は，m_{FP}, m_{NFP}, m_{new} について生成されるトークンの集合，T^{FP}, T^{NFP}, T^{new} を列挙したものである．図3(a) と (b) が現時点でそれぞれFPとNFPの辞書に格納されている全てのトークンであり，図3(c) は新たに生成されたトークンである．下線を引いた部分はモジュール m_{new} と同一のトークンであることを表す．この図から，m_{FP} と m_{new} の間で同一なトークンの数は14であり，m_{NFP} と m_{new} の間で同一なトークンの数は13であることが分かる．この情報から新規モジュール m_{new} が不具合を含む確率 $P(T^{FP}|T^{new})$ を算出する．ベイズの定理によって，確率 $P(T^{FP}|T^{new})$ は次のように求められる．

$$\frac{P(T^{new}|T^{FP})P(T^{FP})}{P(T^{new}|T^{FP})P(T^{FP}) + P(T^{new}|T^{NFP})P(T^{NFP})}$$

まず，学習されているのは T^{FP} と T^{NFP} だけなので，それぞれのトークンの存在する事前確率は $P(T^{FP}) = P(T^{NFP}) = 1/2$ となる．次に，m_{FP} のトークン T^{FP} 内に m_{new} のトークン T^{new} が存在する確率 $P(T^{new}|T^{FP}) = 14/45$ である．また，m_{NFP} のトークン T^{NFP} 内に T^{new} が存在する確率 $P(T^{new}|T^{NFP}) = 13/45$ である．よって，確率は次式で求められる:

$$P(T^{FP}|T^{new}) = \frac{\frac{14}{45} \times \frac{1}{2}}{\frac{14}{45} \times \frac{1}{2} + \frac{13}{45} \times \frac{1}{2}} = 0.519$$

この結果，新規モジュール m_{new} が不具合を含む確率は 0.519 となる．本研究では確率の閾値を 0.50 と定め，0.50 以上であれば FP モジュール，0.50 未満であれば NFP モジュールと判定する．よって，この例では FP モジュールと判定されることになる．

3. 大規模プロジェクトへの適用実験

3.1 対象プロジェクト

ここでは，オープンソースソフトウェアである Eclipse Project(EP) [7] の開発データについて交差検証を用いた実験を行った結果を示す．

我々の手法においては，本来事前にソースコードを準備する必要は無い．しかし，本実験に当たっては事前にソースコードリポジトリからモジュールを取得しておき，そのモジュールにバグが含まれていたか否かの真値をあらかじめ把握しておく必要がある．そこで，本研究では文献 [17] に示されているアルゴリズムを用いてオープンソースのソフトウェアリポジトリから FP モジュールと NFP モジュールを抽出した．

実験対象である Eclipse とその関連プロジェクトにおける FP モジュール抽出の結果を表 2 に示す．

表 2　対象プロジェクトの概要

開発言語	Java
リポジトリのサイズ	15.6GB
収集した不具合の状態	Resolved, Verified, Closed
不具合の解決状況	Fixed
不具合の重大度	blocker, critical, major, normal
上記条件での不具合の数	44,600
CVS のログから発見した不具合数	24,344 (54.6%)
FP モジュールの数	73,902
NFP モジュールの数	1,289,463

表 3　実験結果の凡例

		予測	
		NFP	FP
実測	NFP	N_1	N_2
	FP	N_3	N_4

3.2 予測精度の評価尺度

表 3 は今回の実験で得られる結果の凡例である．N_1，N_2，N_3，N_4 は横に示す予測と縦に示す実測にそれぞれ該当する例数を表す．この得られた結果の評価指標として精度(Accuracy)，再現率(Recall)，適合率(Precision) を用いる．

精度(Accuracy) は全モジュールのうち，実測が NFP のモジュールをNFP，実測が FP のモジュールを FP と正しく予測した割合を示す．よって精度は凡例の表 3 を用いると以下のように定義される．

$$Accuracy = \frac{N_1 + N_4}{N_1 + N_2 + N_3 + N_4}$$

精度は予測の全体的な傾向を把握するには便利であるが，実測値の偏りなどに大きく影響を受ける指標であるため，この値のみで予測の良さを判断するのは危険である☆．そのため，本研究では以下の 2 つの指標に重点を置く．

再現率（Recall）は実測が FP である全てのモジュールのうち，正しく FP と予測できたものの割合を示す．よって再現率は以下のように定義される．

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

直感的には，再現率は「予測によって実際の不具合をどれだけ網羅できるか」を示している．そのため，Fault-prone モジュール予測にあっては，不具合を未然に防ぐという観点から最も重視すべき指標と言える．

適合率（Precision）は予測が FP であるモジュールのうち，実測が FP であったものの割合を示す．すなわち，適合率は以下のように定義される．

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

☆ 例えば，全ての予測を NFP とした場合でも，精度は $\frac{N_1}{N_1+N_3}$ となるので，実測での FP と NFP の比を表す値を示す．

表4　予測実験の結果

OSB		Predicted	
		NFP	FP
Actual	NFP	12,249	7,093
	FP	2,972	16,243
		精度	0.739
		適合率	0.696
		再現率	0.845
		F_1	0.763

適合率は，直感的には1つの不具合を見つけるのにどのぐらい無駄なモジュールを調べる必要があるか，すなわちテストのためのコストを表している．

3.3 交差検証

本手法の妥当性検証のために，今回の実験では十重交差検定を行った．具体的には，データをランダムに10群に均等に分割し，まず1番目の群を外して残りの9群を学習データとしてCRM114に学習させ，外しておいた1群をテストデータとして用いてFPかNFPかの予測を行う．次は2番目の群を外して，と同様にこの作業を10回繰り返す．

前節で抽出したすべてのモジュールを利用すると莫大な時間が必要になるため，今回はFPモジュールとNFPモジュールからそれぞれ約20,000モジュールをランダムサンプリングして実験データとした．

3.4 適用結果

表4は，交差検証が終了した時点での予測と実測をクロス集計したものである．表4からは，Fault-proneフィルタリングによって精度と再現率に関してはそれぞれ高い値が得られたことを確認できる．再現率に関しては0.845を示しており，高い不具合予測能力が確認できる．また，適合率に関しても0.696と比較的高い値を示していることから，不具合の予測に必要なコストもそれほど高くないものと考えられる．このことから，Eclipseプロジェクトにおける実験ではFault-proneフィルタリングがうまく適用できていることが確認できた．

4. 従来研究との比較

Fault-proneモジュールを予測する研究は古くから行われている．1999年以降だけでも多くの研究がなされている[1, 3, 6, 10−12, 16]．

多くの研究では複雑度やソフトウェアの構造に関するソフトウェアメトリクスを収集し，数理的モデルを作成することで予測を行ってきている．例えば，Brianらはオブジェクト指向メトリクスからロジスティック回帰モデルを作成することでFault-prone　モジュールの予測を行っている[3]．また，KhoshgoftaarらはMcCabeの複雑度メトリクスやHalsteadのメトリクスなどを用いて，さまざまな分類手法によるFault-proneモジュール予測の比較を行っている[11]．

ここでは，これまでに提案されてきた手法をその予測精度を中心にまとめる．Denaroらはロジスティック回帰モデルに基づくFault-prone予測を提案し，その精度は0.906，再現率は0.682であったとしている[6]．上で述べたBrianらの手法では精度が0.840，再現率は0.483であった[3]．また，GuoらはDempster-Shafer Belief Networkをもちいた予測手法を提案し，精度は0.690，再現率は0.915であったと報告している[10]．Bellini らの研究では判別分析を用いて予測を行った結果，精度は0.736，再現率は0.543となっている[1]．Khoshgoftaar らは再現率を評価尺度とせずに第二種の過誤の率☆を評価尺度としている[11]．それによると，分類手法によって異なるものの，およそ10 %から20 %の第二種の過誤が発生するとしている．

我々の提案するFault-prone フィルタリングでは表4から精度は0.739，再現率は0.845と比較的高い値を示している．また，第二種の過誤の率も計算してみると，15 %前後と低い値を示した．

もちろん実験する環境が違うために一概に優劣を比較はできないが，Fault-proneフィルタリングでは従来手法で示されている程度の精度，再現率を達成できていることが分かる．

5. まとめと今後の課題

本稿では，スパムフィルタを利用したフォールト・プローン・モジュールの予測手法，Fault-proneフィルタリングを紹介し，その適用実験について述べた．実験の結果，本手法が高い予測性能を持つことが確認できた．

本研究で提案するFault-prone　フィルタリングはソースコードのみを入力として実施することができ

☆ 第二種の過誤の率は表3の表記では $\frac{N_3}{N_1+N_2+N_3+N_4}$ となる．

るため，実プロジェクトへの適用も容易であると考える．今後は企業で実施された開発への適用が挑戦すべき課題の１つである．

参　考　文　献

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 (ICDM2004)*, 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) J. Goodman, G. V. Cormack, and D. Heckerman. Spam and the ongoing battle for the inbox. *Communications of the ACM*, 50(2):25－33, February 2007.

9) P. Graham. *Hackers and Painters: Big Ideas from the Computer Age*, chapter 8, pages 121－129. O Reilly Media, 2004.

10) 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.

11) T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical study. *Empirical Software Engineering*, 9:229－257, 2004.

12) 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.

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

14) *POPFile*. http://popfile.sourceforge.net/.

15) 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.

16) 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.

17) 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.