フォールト・プローン・フィルタリング：不具合を含むモジュールのスパムフィルタを利用した予測手法

水野 修[†] 菊野 亨[†]

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．本論文では適用実験としてオープンソースソフトウェア eclipse とその関連プロジェクトに対して予測を行い，予測精度についての評価を行った．

Fault-Prone Filtering: A Simple Approach to Predict Fault-Prone Modules Using Spam Filter

OSAMU MIZUNO [†] and TOHRU KIKUNO[†]

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. Source code modules are applied to fault-prone filter with Training Only Errors(TOE) procedure. By experiments using open source project, we confirmed that our approach has high accuracy.

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, 17]．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, 14].

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 [16]．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が可能であることを示し

† 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

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 [2,15]，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 CRM114 を用いる [4]．

Fault-prone フィルタリングを実際のプロジェクトに適用するにあたっては，スパムフィルタリングで用いられる手法である「誤判定時のみ学習 (Training Only Errors: TOE)」という方式が現実的である．この方法は，まずモジュールの分類を行い，分類結果が実際の結果と異なっていたときのみ，そのモジュールを学習するという手法である．これは一般的に電子メールのフィルタリングを行う動作同様である．この方法を用いることで，実際の環境に近い状況，すなわち，事前の知識が全くない状態からの予測モデル構築を行うことが可能である．

本論文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 TOE の手続きとその効果を確かめる小規模な実験について述べる．4 節では実用的な大きさの 2 つのプロジェクトへの適用実験について述べる．また，5 節では実験結果についての議論，および，関連研究との比較についてまとめる．最後に，6 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．

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	

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

```
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}

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．

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

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

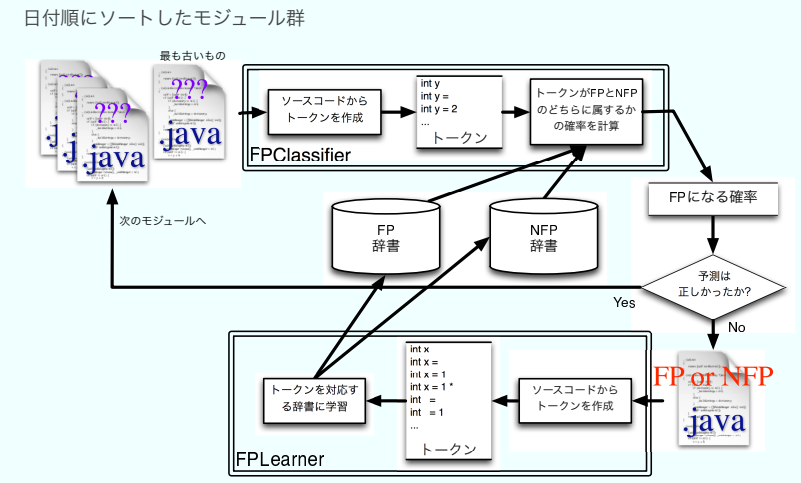


図 4 Fault-prone フィルタリングでの TOE の流れ

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. Fault-Prone フィルタリングの適用

3.1 誤判定時のみ学習 (TOE)

Fault-prone フィルタリングを適用するにあたって，我々は「誤判定時のみ学習 (Training Only Errors: TOE)」という方式を採用する．これは，スパムフィルタにおいても利用される方式であり，メールを到着順に分類し，その分類が正しかったかどうかを判断できる時点で学習を行う手法である．Fault-prone フィルタリングの適用実験では，次のような手順になる．

(1) 当該プロジェクトのモジュール群を古いものから順番にソートする．
(2) 古いモジュールから順に 1 つずつ取り出し，そのモジュールが FP か NFP であるかを Fault-prone フィルタにより判定する．
(3) モジュールに不具合が含まれているかことが判明した時点で，予測が正しければ何もせずに (2) へ戻る．
(4) 予測が正しくなければ，正しい結果を学習させる．例えば，NFP と予測したのに実際の結果が FP であった場合，FP 辞書に該当するモジュールの内容を学習させる．その後，(2) へ戻る．

図 4 にこの手順を図示する．この実験はモジュールが作られた順を追って Fault-prone フィルタを適用するため，実際の開発環境に近い状態を再現することができる．

3.2 小規模なプロジェクトへの適用

ここでは図 5 に示す小規模なソースコードの履歴について TOE 実験を行った例を示す．図 5 は階乗と総和の計算の結果を表示するプログラムを改訂した履歴である．ここでは 5 つのリビジョン (Rev.) があったとしている．なお，本研究では Java を開発言語としたプロジェクトを対象としたため,「モジュール」の単位を Java クラス内の「メソッド」と定義した．

表 2 に，この実験で各モジュールがどのように予測されたのかと，その予測の正誤を示す．表 2 の「実際」はそのモジュールが実際に不具合を含んでいた (FP) か，いなかったか (NFP) を示す．なお，本研究ではモジュールが変更された時にそのモジュールの予測を行うため，その直前の版から変更がないモジュールは予測の対象とならない．そのため，変更の無いモジュールはこの表には含まれていない．Rev. 1.1 から 1.5 までの間に変更されたモジュール数は 11 である．

より詳しく表 2 を見ると，次のようなことが分かる．

- `fact()` については，Rev. 1.1 で本来 FP であるところを NFP と誤判定している．続く Rev. 1.2 と Rev. 1.3 では正しく FP を FP と予測しているが，Rev. 1.4 で NFP を FP と誤判定する．最後に

Hello.java.1.1

```
public class Hello {
    public static int fact(int x) {
        return(x<=1?1:fact(x++));                    FP
    }
    public static void main(String[] args) {
        int n = 10;
        System.out.printf("%d\n",n,Hello.fact(n));  NFP
    }
}
```

Hello.java.1.2

```
public class Hello {
    public static int fact(int x) {
            return(x<=1?1:fact(x--));                FP
    }
    public static void main(String[] args) {
        int n = 10;
        System.out.printf("%d\n",Hello.fact(n));     NFP
    }
}
```

Hello.java.1.3

```
public class Hello {
    public static int fact(int x) {
            return(x<=1?1:x*fact(x--));              FP
    }
    public static int sigma(int y) {
        return(y<=1?1:fact(y++));                    FP
    }
    public static void main(String[] args) {
        int n = 100;
        System.out.printf("%d\n",Hello.fact(n));     NFP
        System.out.printf("%d\n",Hello.sigma(n));
    }
}
```

Hello.java.1.4

```
public class Hello {
    public static int fact(int x) {
        if (x < 0) {
            return -1;
        } else {                                     NFP
            return(x<=1?1:x*fact(x--));
        }
    }
    public static int sigma(int y) {
        return(y<=0?0:y+sigma(y--));                 NFP
    }
    public static void main(String[] args) {
        int n = 100;
        System.out.printf("%d\n",Hello.fact(n));     NFP
        System.out.printf("%d\n",Hello.sigma(n));
    }
}
```

Hello.java.1.5

```
public class Hello {
    public static int fact(int x) {
        return(x<0?-1:(x<=1?1:x*fact(x--)));         NFP
    }
    public static int sigma(int x) {
        return(x<=0?0:x+sigma(x--));                 NFP
    }
    public static void main(String[] args) {
        int n = 128;
        System.out.printf("%d\n",Hello.fact(n));     NFP
        System.out.printf("%d\n",Hello.sigma(n));
    }
}
```

図 5　対象とするソースコード履歴

1.5 では NFP を NFP と正しく予測している．

- `sigma()` については，Rev. 1.3 で FP を FP と正しく予測しているが，Rev. 1.4 で NFP を FP と誤判定している．

以下では誤判定が発生した部分について詳しく説明する．

表 2　TOE による予測結果

Rev.	モジュール名	実際	予測	正誤
1.1	main()	NFP	NFP	正
1.1	fact()	FP `fact(x++)` が間違い 本来は `x*fact(x--)`	NFP	誤
1.2	fact()	FP `fact(x--)` が間違い 本来は `x*fact(x--)`	FP	正
1.3	main()	NFP	FP	誤
1.3	fact()	FP 条件判定に `x<0` がない	FP	正
1.3	sigma()	FP `fact(y++)` が間違い 本来は `y+sigma(y--)`	FP	正
1.4	sigma()	NFP	FP	誤
1.4	fact()	NFP	FP	誤
1.5	main()	NFP	NFP	正
1.5	fact()	NFP	NFP	正
1.5	sigma()	NFP	NFP	正

(1)　Rev. 1.1 `fact()`: 直前の Rev. 1.1 `main()` において学習が行われないので FP, NFP ともに辞書は空である．そのため，事前に決められた値である NFP へと予測をしている．結果としてこの予測は間違いなので，Rev. 1.1 `fact()` の内容が FP 辞書に学習される．

(2)　Rev. 1.3 `main()`: (1) で FP 辞書が学習されたことにより，`main()` と FP の辞書内に一致点が発生する．そのため，`main()` が FP と誤判定される．その結果 NFP 辞書に Rev. 1.3 `main()` の内容が学習される．

(3)　Rev. 1.4 `sigma()`: バグは取り除いたが FP 辞書のほうに一致点が多いため FP と誤判定され，内容が FP 辞書に学習される．

(4)　Rev. 1.4 `fact()`: こちらも (3) と同様バグは取り除いたが，FP の辞書に一致点が多いため FP と誤判定され，内容が FP 辞書に学習される．

4. 大規模プロジェクトへの適用実験

4.1 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 Modeling Framework(EMF), Eclipse Project(EP) [7] の 2 種類の開発データについて TOE を用いた実験を行った結果を示す．

TOE 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 [18] に示されているアルゴリズムを用い

表 3 対象プロジェクトの概要

名称	EMF	EP
開発言語	Java	
リポジトリのサイズ	962MB	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	4,042	44,600
CVS のログから発見した不具合数	2,832 (70.1%)	24,344 (54.6%)
FP モジュールの数	10,636	73,902
NFP モジュールの数	152,821	1,289,463

表 4 実験結果の凡例

		予測	
		NFP	FP
実測	NFP	N_1	N_2
	FP	N_3	N_4

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 3 に示す．なお，モジュールの抽出に要する時間はプロジェクトの規模によって異なるが，Eclipse Project(EP) については約 17 時間であった．

4.2 予測精度の評価尺度

表 4 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 4 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．

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

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．

上記指標を利用すると 3.2 節における小規模な実験についても予測精度の評価を行うことができる．表 2 より，精度は 0.636, 再現率は 0.75, 適合率は 0.50 と計算できる．

4.3 適 用 結 果

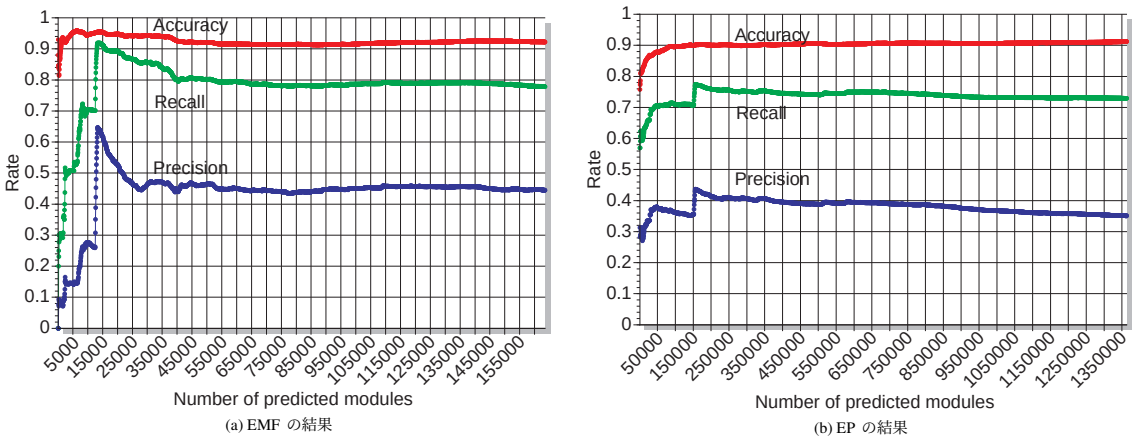
図 6(a), (b) に TOE 実験の経過を示すグラフを，表 5(a), (b) に最終的な予測結果を示す．

図 6(a),(b) のグラフについて説明する．このグラフの縦軸は精度，再現率，適合率の値を表す．また，横軸は作成時刻の時系列順にソートした全モジュールの通し番号を表している．TOE においては，古いものから順に分類・学習を行うため，一度全てのモジュールを作成時刻でソートしておき，古いものから若い番号を付けてある．例えば横軸の 10000 はこの開発において 10000 番目に作成されたモジュールであることを表し，その時点までのモジュールを予測した結果としての 3 つの指標の値をプロットしたものがグラフになっている．

表 5 は，各プロジェクトについて全てのモジュールを予測し終わった時点での予測と実測をクロス集計したものである．表 5 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．再現率に関しては共に 0.75 前後の値を示しており，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 0.40 前後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，これら 2 つの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．

なお，TOE 実験に要する時間は単純にモジュールの数に比例する．最大の規模をもつ EP においては約 20

☆ 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．



(a) EMF の結果　(b) EP の結果

図 6　TOE 実験における各指標の推移

表 5　TOE 実験での予測結果

(a) EMF の予測結果

		予測	
		NFP	FP
実測	NFP	142,482	10,339
	FP	2,358	8,278
精度 (accuracy)		0.922	
再現率 (recall)		0.778	
適合率 (precision)		0.445	

(b) EP の予測結果

		予測	
		NFP	FP
実測	NFP	1,189,740	99,723
	FP	19,985	53,917
精度 (accuracy)		0.912	
再現率 (recall)		0.730	
適合率 (precision)		0.351	

時間を要したが，1 つのモジュールの予測・学習に要する時間は 0.5 秒前後である．

5. 議　　論

5.1 実験結果の考察

まず，表 5 に示す予測結果について考察する．2 つのプロジェクトに対して適用した結果を比較すると，2 つのプロジェクトについてほとんど同様の傾向を示している．その傾向とは，(1) 再現率はおよそ 0.75 付近であること，(2) 適合率はおよそ 0.40 付近であること，(3) 精度はおよそ 0.90 付近であること，である．

再現率についてみると，0.75 前後という値は実際に発生する 4 つの不具合のうち 3 つまでは補足できるということを表しているため，率としては高いと考えられる．一方，適合率についてみると，0.40 前後という値は，1 つの不具合モジュールを発見するためにもう 1 つか 2 つは実際には不具合のないモジュールを調べなければならないことを表している．

実は，FP と NFP を判定する確率の閾値を動かすことで，再現率と適合率はある程度操作することが可能である．例えば，FP と判定される可能性を高く閾値を設定すれば，再現率は向上するが，適合率は下落する．再現率と適合率はトレードオフの関係になるため，品質を確保するためにどの程度までコストを費やせるかという問題になる．その解は実際に使う状況に依存すると思われる．

次に，図 6 について考察する．2 つのグラフからは共に実験開始時は精度，再現率，適合率の値が激しく乱高下するのが確認できる．これは TOE の手順では全くの無学習から実験を行うためである．ある程度の数の学習を行った時点で指標の値が安定することも確認できる．例えば，EMF の例では実験開始から 15,000 モジュール程度の分類・学習を行った後に実用的な精度が得られている．一般的にスパムメールのフィルタリングでは高々数十通程度メールを学習させることで実用に耐える精度が得られることが確認されている．しかし，ソースコードに適用した場合の精度の向上は比較的遅いため，初期段階での精度向上が今後の課題である．

また，TOE の過程で EMF の 13,000 モジュール付近 (図 6(a)) や，EP の 150,000 モジュール付近 (図 6(b)) のように，再現率と適合率の値が激しく上昇することがあるのが確認できる．この現象は該当する期間に出現するモジュールの予測結果が全て正しくなることによって発生している．全ての予測結果が正しくなる理由については現在調査中であるが，リファクタリングなどにより，従来と全く同内容のソースコードがリポジトリに登録された結果このような現象が発生してい

るのではないかと予想している．

5.2 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 [1,3,6,10–12,17].

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，Briand 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 Briand 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 5 から精度の最大値は 0.922, 再現率の最大値は 0.778(共に表 5(a) より) 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，2%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．

5.3 現場への適用

本研究の成果は，開発の現場において次のように利用できると考えている．まず，モジュールの作成・変更の際に Fault-prone フィルタリングを適用することにより，各モジュールにおける不具合の含有可能性を確率として得ることが可能になる．フィルタリングに利用するものはソースコードのみであるため，適用に必要な手間は少ないと考えている．得られた確率の情報から各モジュールについての不具合の出やすさが分かるため，例えば高い不具合確率が示されたモジュールに関しては，以降の開発においてそのモジュールに関連する部分を十分に精査することが可能となる．また，テストやレビューによる不具合発見時にはその不具合に関連するモジュールの情報を学習という形でフィードバックすることにより，不具合検出の精度を改善していくことが可能になる．

ただし，現時点での本手法はあくまでモジュールの中に不具合が存在する可能性を指摘するのみであるため，具体的な不具合の検出などには別の手法との組合せが有効であると考える．例えば，本手法は多くのモジュール中で特に不具合の含まれそうなものをおおまかに選択するものとし，具体的な不具合の検出には静的解析ツールなどを併用する，などの方法が考えられる．

将来的には，あるモジュールについて不具合を含む確率と共にそのモジュールに関する過去の不具合情報を提供することで，本手法単体での有用性も高められると期待している．

6. まとめと今後の課題

本論文では，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 フィルタリングとその実用的な適用手法，Training Only Errors(TOE) について述べた．また，2 つのオープンソースソフトウェアプロジェクトに対して本手法を適用した実験についても述べた．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.

☆ 第二種の過誤の率は表 4 の表記では $\frac{N_3}{N_1+N_2+N_3+N_4}$ となる．

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) 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, S. Ikami, S. Nakaichi, and T. Kikuno. Fault-prone filtering: Detection of fault-prone modules using spam filtering technique. In *Proc. 1st International Symposium on Empirical Software Engineering and Measurement (ESEM2007)*, (to appear). Madrid, Spain.
14) O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *The 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.
15) *POPFile*. http://popfile.sourceforge.net/.
16) 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.
17) 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.
18) 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.