# バイトコードの機械学習に基づく不具合予測手法の提案

藤原 剛史
京都工芸繊維大学 工芸科学部 情報工学課程
m5622042@edu.kit.ac.jp

水野 修
京都工芸繊維大学 大学院工芸科学研究科 情報工学部門
o-mizuno@kit.ac.jp

## 要旨

不具合を含んでいそうなモジュール（*Fault-prone* モジュール）の予測には，複雑度メトリクスに基づいたモデルが用いられることが多い．しかし，それらのモデルを構築するには，メトリクスを測定するための環境整備やツールへの慣れが必要であり，現場への適用は簡単ではない．そこで，メトリクス測定を行わない *Fault-prone* モジュール検出手法として，「*Fault-prone* フィルタリング」というものが提唱されている．この手法は，スパムフィルタリングの理論を用いたものであり，ソースコードへの簡単な適用のみによって *Fault-prone* モジュールを検知できる．

本研究では，*Fault-prone* フィルタリングによる *Fault-prone* モジュール検出のより高い効果を得ることを目的として，*Fault-prone* フィルタリングをバイトコードへ適用した場合とソースコードへ適用した場合の比較実験を行う．具体的には，対象とするプロジェクトのバイトコードおよびソースコードから単語を抽出し，スパムフィルタに通して結果を得たのち，比較を行う，

本研究ではこの実験を通して，バイトコードによる不具合予測が従来の不具合予測と比べて同等以上の精度を得ることが可能であることを示した．

## 1. はじめに

今やソフトウェアの果たす社会的な役割は大きく，ソフトウェアの品質を高く保つことはソフトウェア工学における重要な目標である．しかし実際には，時間的・人的な理由が障害となって，それらの品質を高く保持するのは難しく，より効率的かつ簡易な手法が求められている．コードを作成した時点で不具合を含んでいそうなモジュール（Fault-prone モジュール）の検出が可能であることは，レビューやデバッグに費やすコストの削減につながる．そのため，これまでも 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][2][3][4][5][6][7]．従来の手法では，主にモジュールの複雑さや変更頻度等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であり，現場への適用を難しくしている一因となっている．

こうした状況の中，メトリクス等の測定の必要なく，ソースコードのみを入力として Fault-prone モジュールの予測が可能である「Fault-prone フィルタリング」と呼ばれる手法が水野らによって提唱されている [8][9]．この手法は，迷惑メール検出に用いられるスパムフィルタリングの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 Fault-prone モジュールを予測する．そして，適用が簡便であるにも関わらず，従来の手法と比較しても劣らない程の高い予測精度が示されている手法である．

先に示した Fault-prone フィルタリングの研究では，ソースコードを入力として Fault-prone モジュールの予測が行われているが，実は Fault-prone フィルタリングに用いる入力は，テキスト情報であればソースコードでなくてもよい．そこで，本研究では，この Fault-prone フィルタリングをバイトコードへ適用する．バイトコードを用いる理由としては，ソースコードに比べ開発者のコードの書き方等に依存しにくく，精度が保証されれば有用であるだろう、という考えのもとに採用した．そして，ソースコードによる予測と比べてどのように精度が違うのかを，ソースコードへ適用した場合との比較実験を通じて調べる．

本報告書の構成を以下に示す．２章では，本研究を行

う目的について述べる．３章では本実験への準備として，前提知識として必要な事項を説明する．４章では，本実験の対象となるプロジェクトおよび本実験の準備の手順を述べるとともに，実験の方法について説明する．５章では，予測精度の評価尺度について言及し，本実験の結果を述べる．６章では，本実験の結果を基に考察を行う．そして７章では，本研究のまとめと今後の課題について述べる．

## 2. 研究の目的

これまで行われてきた Fault-prone フィルタリングの研究においては，ソースコードを入力とした不具合予測で高い成果を収めてきた．しかし，Fault-prone フィルタリングの入力はテキスト情報であればソースコードである必要はない．何か別のテキスト情報を入力とすることによって，より良い結果が得られる可能性がある．

そこで本研究では，Fault-prone フィルタリングの入力をバイトコードにすることによって不具合予測を行い結果を得る．バイトコードを用いる理由としては，バイトコードは最低限の命令列で記述されているため，開発者のコードの書き方等に依存しにくいので，精度が保証されれば有用であることが挙げられる．また，ソースコードを入力とした不具合予測との比較を行い，どのような違いが生じるのかを調べる．

## 3. 準備

本章では実験を行う前に，実験の前提となる事項を紹介する．

### 3.1. Fault-prone フィルタリング

迷惑メール検出に用いら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 Fault-prone モジュールを予測する手法である．

#### 3.1.1 スパムフィルタ

現在使用されている一般的なスパムフィルタは，その仕組みにベイズの定理を利用しているものが多い．これらのスパムフィルタでは受信したメールの特徴を学習し，新たにメールを受信した際に，それらの学習に従ってメールを本人に有用なメール (ham) と迷惑なメール (spam) に分類するといった仕組みを取っている．基本的なスパムフィルタの動作は，メールを学習するステップと，学習した結果を用いてメールが ham メールか spam メールかを分類するステップの２ステップで構成されている．

#### 3.1.2 CRM114

Yerazunis らによって開発されたスパムフィルタであり [10] ，Fault-prone フィルタリングに使用されている．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等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 Marcovmodel (OSB)”を使用する．OSB は任意の連続する５単語の組み合わせのうち，２単語からなるものだけをトークンとする手法である．

### 3.2. PROMISE

ソフトウェア工学のためのデータセットを公開しているウェブサイトである．PROMISE 内で公開されているデータセットの中には，不具合情報や，提案手法との比較に利用することができる，

### 3.3. Lex

Lex は，レキシカルアナライザ（字句解析プログラム）を生成するプログラムである．テキスト中の文字列の変換，カウント，抽出などさまざまな目的に使われ，その応用領域は，コンパイラやコンバータの作成を筆頭に，自然言語処理や簡単な整形まで幅広い．

本研究では，バイトコードおよびソースコードの単語抽出に使用する．

### 3.4. 誤判定時のみ学習 (TOE)

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，対象の分類を行い，その分類結果が実際の結果と異なっていたと判断した時点でのみ学習を行う手法である．この手法を用いることで，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が可能となる．

本研究では，実際は Fault-prone であるバイトコードおよびソースコードで示されたモジュールを Non-Fault-prone であると判断したとき，または実際は Non-Fault-prone であるモジュールを Fault-prone であると判断したときのみに学習する．

## 4. 実験

本章では，実験の対象・準備・方法について説明する．

本実験では，Fault-prone フィルタリングの入力をバイトコードとすることで，Fault-prone モジュールの予測を行う．また，ソースコードを用いた場合の予測との変化を確認するため，入力をソースコードとした Fault-prone モジュール予測との比較を行う．

以下，この実験においてはクラスをモジュール単位として扱う．

### 4.1. 実験対象

本実験では，次に示すオープンソースプロジェクトを対象とする．

- **Apatch Ant**

  Apache Ant は，GNU make の Java 版ともいえるものであり，OS など特定の環境に依存しにくいビルドツールである．記述言語は Java である．

  今回の実験では rev.1.5，rev.1.6，rev1.7 の３つのリビジョンを対象とする．

### 4.2. 実験の準備

実験を行う前にまず, ”THE APACHE ANT PROJECT”[1] から Apache Ant のバイトコードとソースコードをリビジョン毎に用意する．また，Apache Ant に関するリビジョン毎のデータセットを PROMISE から取得し，以降の操作を行う．

#### 4.2.1 jar ファイル展開

用意したバイトコードは jar（Java アーカイブ）形式で圧縮されているため，クラスファイルに展開する必要がある．jar 形式のファイルを展開する手段は JDK （Java Development Kit）で提供されており, jar コマンドの -x オプションと-f オプションを使用することで，バイトコードが記述されたクラスファイルを取得することが可能である．

#### 4.2.2 逆アセンブル

クラスファイルの状態だと学習に利用できないため，オペコードを取得する必要がある．しかし, バイナリデータからオペコードを抽出するのは困難であるため，クラスファイルに逆アセンブル[2] を行う．クラスファイルを逆アセンブルする手段は JDK で提供されており，javap コマンドの -c オプションをクラスファイルに適用するだけで，アセンブリコードの命令列（以下アセンブリコードと呼ぶ）を得ることが可能である．

#### 4.2.3 ソースファイル分割

１つの Java ソースファイル（以下ソースファイルと呼ぶ）には複数のクラスが定義されていることがある．クラスを１つのモジュールとして扱うと，１つのソースファイルに複数のモジュールが存在することとなり，実験を行うにあたって不都合である．よって，１つのソースファイルには１つのモジュール（クラス）のみが存在するようにソースファイルを分割する．

#### 4.2.4 タグ付け

PROMISE のデータセットに準拠して，各モジュールが Fault-prone であるか Non-Fault-prone であるかを決定する．本実験において Fault-prone なモジュールとは，バグを少なくとも１つ含んでいるモジュールであると定義

[1] http://ant.apache.org/git.html

[2] 実行可能なバイナリデータを人間が可読なアセンブリ言語に変換すること

し，Non-Fault-prone なモジュールとは，バグを１つも含んでいないモジュールであると定義する．

タグ付けの結果, Fault-prone なモジュールが 291, Non-Fault-prone なモジュールが 1,096, 合計 1,387 のモジュールが得られた．

#### 4.2.5 トークナイザ作成

Fault-prone フィルタリングを行うにあたって，テキスト情報を単語に分割するトークナイザが必要となる．本実験では，Lex を用いて作成した字句解析プログラムをもってトークナイザとする．具体的には，正規表現を用いて，アセンブリコードからニーモニック [3] を抽出するトークナイザとソースコードから単語 [4] を抽出するトークナイザの２つを作成する．

### 4.3. 実験方法 1

本実験では，Fault-prone フィルタリングを適用するにあたって，誤判定時のみ学習（TOE）を採用する．TOE を用いた Fault-prone フィルタリングの具体的な手順を以下に示す．

(1) 当該プロジェクトのモジュール群を古いリビジョンのものから順にソートする．ただし，同リビジョン内のモジュール群の順に関しては，ランダムに決定する．

(2) 並び替えたモジュール群から１つ取り出して適当なトークナイザを適用し，Fault-prone であるか Non-Fault-prone であるかをスパムフィルタにより予測する

(3) モジュールの予測が完了した時点で，その結果とタグを比較し，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，その後 (2) へ戻る．

以上の (1)〜(4) の手順を, アセンブリコードおよびソースコードで記述された，それぞれのモジュール群に対して適用することをもって, 実験の方法とする．このとき，モジュールが Fault-prone であるか Non-Fault-prone であるかの予測を決定するための確率の閾値は，0.5 に設定している．

[3] オペコードと１対１対応している人間が可読な文字列

[4] ここで述べる「単語」とは，変数名，演算子，キーワード等のソースコードを構成する要素のことである．ただし，本研究では区切り文字を示す” ; ”は含まない．

### 4.4. 実験方法 2

より具体的な比較をするため，アセンブリコードおよびソースコードで記述されたそれぞれのモジュール群に対して，閾値を動かして Fault-prone モジュールを予測し，結果を評価する．予測の手順に関しては，先に示した TOE を用いた Fault-prone フィルタリングの手順 (1)〜(4) を用いる．また，閾値は 0.05 から 0.95 まで 0.05 刻みで動かして実験を行う．このとき，TOE という手法は予測の順への依存が大きいため，閾値を変えて測定し直す際に，最初にランダムに決定した同リビジョン内のモジュール群の順を変えないように注意する．

## 5. 実験結果

本章では，実験によって得られた結果を示す．

### 5.1. 実験の条件

本実験に行う上での条件を次に示す．

#### 5.1.1 スパムフィルタ

本実験では，スパムフィルタとして CRM114 を使用する．また分類の手法として，CRM114 のデフォルトの”Othogonal Sparce Bigrams Marcovmodel（OSB）”を使用する．

#### 5.1.2 閾値（Threshold）

対象となるモジュールに CRM114 を適用すると，0〜1 の値が得られる．この値はベイズ識別によって得られた，そのモジュールが Fault-prone である確率である．閾値は，そのモジュールが Fault-prone か Non-Fault-prone かの予測を決定するための境界値であり，Fault-prone である確率が閾値より大きければ Fault-prone と予測され，閾値以下であれば Non-Fault-prone と予測されるということである．

当実験においては，実験方法１では閾値を 0.5 に設定し，実験方法２では閾値を 0.05〜0.95 間を 0.05 刻みで動かしながら実験を行う．

### 5.2. 評価尺度

表 1 は本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．またこの表 1 では便宜上，Fault-prone を FP，Non-Fault-prone を NFP と省略する．本実験で得られた結果の評価尺度として以下の，精度（Accuracy），適合率（Precision），再現率（Recall）の３つの指標を使用する．

表 1. 実験結果の凡例

| | | 予測 | |
|---|---|---|---|
| | | FP | NFP |
| 実測 | FP | $N_1$ | $N_2$ |
| | NFP | $N_3$ | $N_4$ |

#### 5.2.1 精度（Accuracy）

精度は，全モジュールのうち，予測が正しかった割合を示す．よって，精度は以下のように定義される．

$$Accuracy = \frac{N_1 + N_4}{N_1 + N_2 + N_3 + N_4} \tag{1}$$

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，実測値の偏り等に大きく影響を受ける指標である．そのため，この値のみで予測の良さを判断するのは危険である．

#### 5.2.2 適合率（Precision）

適合率は，予測が Fault-prone であるモジュールのうち，実測が Fault-prone であったものの割合を示す．よって，適合率は以下のように定義される．

$$Precision = \frac{N_1}{N_1 + N_3} \tag{2}$$

適合率は，直感的には１つの不具合を見つけるのにどのくらい無駄なモジュールを調べる必要があるかを示す指標である，すなわち，テストのためのコストを示している．

#### 5.2.3 再現率（Recall）

再現率は，実測が Fault-prone である全てのモジュールのうち，正しく Fault-prone 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．

$$Recall = \frac{N_1}{N_1 + N_2} \tag{3}$$

再現率は，直感的に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す指標である．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いえる．

### 5.3. 実験結果１

[ **4.3　実験方法１** ] によって得られた Fault-prone フィルタリングの経過のグラフを図 1，図 2 に，最終的な予測結果を表 2，表 3 に示す．

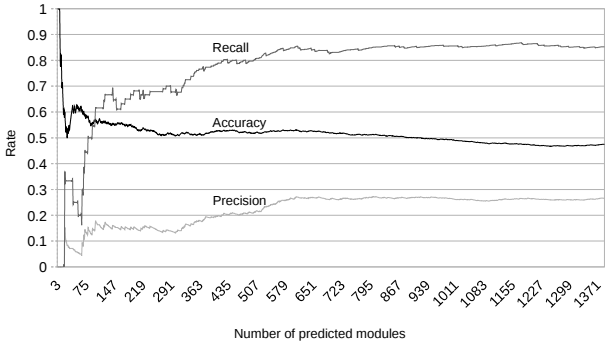


図 1. バイトコード入力時における各指標の推移

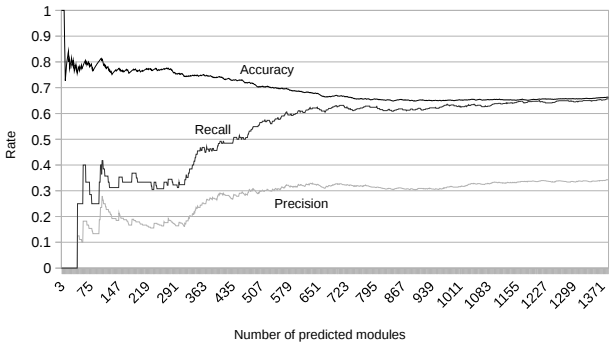


図 2. ソースコード入力時における各指標の推移

表 2. バイトコードモジュールの最終予測結果

| | | 予測 | |
|---|---|---|---|
| | | FP | NFP |
| 実測 | FP | 248 | 43 |
| | NFP | 686 | 410 |
| 精度（accuracy） | | 0.474 | |
| 適合率（precision） | | 0.266 | |
| 再現率（recall） | | 0.852 | |

表 3. ソースコードモジュールの最終予測結果

| | | 予測 | |
|---|---|---|---|
| | | FP | NFP |
| 実測 | FP | 192 | 99 |
| | NFP | 388 | 708 |
| 精度（accuracy） | | 0.649 | |
| 適合率（precision） | | 0.331 | |
| 再現率（recall） | | 0.660 | |

図 1，図 2 のグラフについて説明する．図 1 のグラフは，バイトコードモジュールに Fault-prone フィルタリングを適用したときの各指標の推移経過を，また図 2 のグラフはソースコードモジュールに Fault-prone フィルタリングを適用したときの各指標の推移経過を示している．TOE では無学習の状態から 1 つずつモジュールを予測していき，その予測結果が間違っていればその時点で学習を行うため，予測し終えたモジュールの数によって予測精度が変わる．これらのグラフは，第 n 個目までのモジュールの予測が完了した時点での 3 つの指標の値をプロットしたものである．このグラフの縦軸は精度（Accuracy），適合率（Precision），再現率（Recall）の各指標の値を表す．また，横軸は精度，適合率，再現率の各指標を算出した時点での，既に予測が完了しているモジュールの数を表している．

次に，表 2，表 3 について説明する．これらの表は，対象としたプロジェクトの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．特に，表 2 はバイトコードモジュールを予測したときの予測結果であり，表 3 はソースコードモジュールを予測したときの予測結果である．

これらの結果からは，バイトコードモジュールの Fault-prone フィルタリングによる予測は，ソースコードモジュールの予測に比べて精度，適合率が低いことが確認できる．特に適合率は 0.266 と，1 つの Fault-prone モジュールを見つけるのに余分なモジュールを 3 つも調べなければならないことを示している．しかし再現率は，ソースコードモジュールのものと比べても 0.852 と著しく高く，これはおよそ 85 %の Fault-prone モジュールを検出できていることを意味する．

### 5.4. 実験結果 2

[ **4.4　実験方法 2** ] によって得られた閾値を動かして行った Fault-prone フィルタリングの結果のグラフを図 3，図 4 に示す．

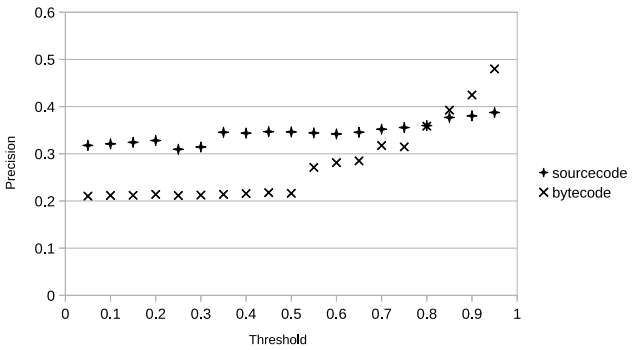


図 3. 閾値の変化における適合率の推移

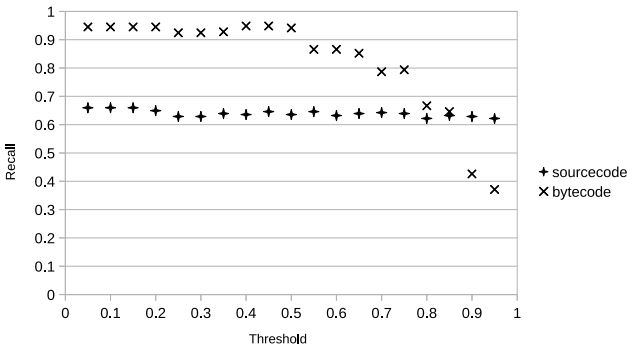


図 4. 閾値の変化における再現率の推移

図 3，図 4 のグラフについて説明する．このグラフは，閾値（Threshold）を変化させた場合の Fault-prone フィルタリングの適合率（Precision）と再現率（Recall）の推移について示している．図 3 のグラフの縦軸は適合率の値，図 4 のグラフの縦軸は再現率の値を表し，これらのグラフの横軸は閾値を表している．

ここで注意すべきは図 4 のグラフにおいて，閾値の増加に伴い再現率の値が増加することがあることである．通常，すべてのデータを学習する場合は閾値の増加によって再現率が上がることはない．しかし，本実験で

は TOE という手法を使っており，閾値を変えることによって予測が変われば，今までの閾値で学習されていたモジュールが学習されなくなる，また逆に学習されていなかったモジュールが学習されるようになるということが起こりうる．よって，ある程度の再現率の揺れは十分にあり得るものであると考えられる．

図 3，図 4 では，閾値を 0.8 あるいは 0.85 としたときにバイトコードモジュールの予測結果による適合率と再現率双方の値がソースコードモジュールの予測結果の値を上回っている，あるいは同等であることが確認できる．このことからバイトコードモジュールによる予測精度は，閾値を変えることによってソースコードモジュールの予測精度を上回ることが可能である．

## 6. 考察

本章では，実験に対する考察を行う

### 6.1. 妥当性の検証

本実験で得られた結果の妥当性を検証する．

- **対象プロジェクトの不足**

  本実験では，Apache Ant を対象のプロジェクトとして特定の結果を得た．しかし，対象としたプロジェクトが 1 つしかなく，本実験で示した結果が他の多くのプロジェクトに対して，一般的に当てはまるとは限らない．また，Apache Ant はオープンソースプロジェクトであり，商業プロジェクトに今回の手法を適用しても同等の結果が得られる確証はない．

- **Perl スクリプトの不具合**

  本実験を行うにあたって，当環境では複数の処理を一括して行うために，Perl スクリプトを作成して実験を行っている．得られた実験結果に関してはある程度の根拠があり，目下のところ特別な問題は見受けられないが，作成した Perl スクリプトに何らかの不備が存在して実験の処理の手順を誤っている可能性がある．

- **トークナイザの不具合**

  本実験において，Lex を用いて作成した字句解析プログラムをもってトークナイザとしたが，この字句解析プログラムに不具合がある可能性がある．いくつかのモジュールデータに対して適用して確認をしてみてはいるが，人手で確認するにはデータの量が膨大なためすべてのデータに対して確認ができておらず，字句解析プログラムが不具合なく動いているという確証はない．

- **モジュール群の適用順**

  本実験を行うにあたって，同リビジョン内のモジュール群の順をランダムにソートした．これは，ダウンロードしてきたバイトコードおよびソースコードからは詳細な作成日時が不明であったためである．今回実験に用いた手法である TOE においては適用順によって学習内容が変わるので，最悪の場合と最良の場合とで大きな差が出る可能性がある．

### 6.2. 実験結果の考察

まず，実験方法 1 の結果を考察する，図 1，図 2，表 2，表 3 はバイトコードモジュールおよびソースコードモジュールの fault-prone フィルタリングを用いた不具合予測における，精度，適合率，再現率の各指標の推移と最終結果を示したものである．図 1 と表 2 からはバイトコードモジュールの不具合予測の結果を知ることができる．バイトコードモジュールの不具合予測は精度と適合率はあまり高くないが再現率だけは非常に高い，このことの理由がこれらの図表から確認できる．その理由とは，バイトコードモジュールの不具合予測ではモジュールの多数を Fault-prone であると予測しているということである．特に，実測が Non-Fault-prone なモジュールでは 1,096 モジュール中 686 ものモジュールが Fault-prone と誤判定されていることがわかる．しかし一方，Fault-prone モジュールは 291 モジュール中 248 ものモジュールの予測に成功しており，図 1 のグラフからもわかるように，TOE による学習は一定の成果を上げている．このようなことが起こる理由として考えられるのは，バイトコードモジュールでは Fault-prone なモジュールは Fault-prone であると識別できる何か決定的な特徴を持ちながら，その他の特徴は Non-Fault-prone なモジュールと似通っている，そして，Non-Fault-prone なモジュールは Non-Fault-prone であると識別できる決定的な特徴を持っていないということである．一方，図 2 と表 3 からはソースコードモジュールの不具合予測の結果を知ることができる．これらの図表からは，ソースコードモジュールの不具合予測は精度と再現性が比較的高いこ

とが確認できる．バイトコードモジュールの予測と反して，$N_1 : N_2$ と $N_4 : N_3$ がともにおよそ 2 : 1 の比になっていることからも，バランスの良い安定した学習と予測が行われていると考えられる．

次に，実験方法２の結果を考察する．図 3，図 4 は確率の閾値の変化における適合率と再現率の推移を示している．図 3，図 4 を見てまず注意すべきことは，バイトコードモジュールの不具合予測では閾値を変えると適合率と再現率が大きく変化するのに対し，ソースコードモジュールの不具合予測では閾値を変えても再現率がほぼ変わらず，適合率もさほど変わらないことである．この結果が示すことは，バイトコードモジュールの不具合予測では，Fault-prone であるか Non-Fault-prone であるか明確でないモジュールが多いのに対して，ソースコードモジュールの不具合予測ではモジュールが Fault-prone であるか Non-Fault-prone であるかが明確であるということである．特に，閾値を上げても再現率がほぼ変化していないことから，Fault-prone であると予測されたモジュールのほぼすべてにおいて Fault-prone である確率が１に近いことを表している．これは，バイトコードから抽出された単語（ニーモニック）は種類が少なく，ユニークなトークンが乏しくなる一方で，ソースコードから抽出された単語は，変数名や記号が含まれることから種類も多く，トークンがユニークになりやすいことに起因すると考えられる．つまり，バイトコードモジュールでは特徴が似たようなものが多くなり，Fault-prone である確率が中程の値を取りやすいのに対して，ソフトコードモジュールでは特徴がユニークであるので，多くのモジュールにおいて Fault-prone である確率は 0 か 1 に近く，閾値を変えても予測がほとんど変わらないものであると予想される．

実験方法１，実験方法２の結果を通して述べた通り，バイトコードモジュールの不具合予測においては Fault-prone である確率がどっち付かずとなることが多く，実践的な適用に向いてないように思われるかもしれない．しかし，図 1 と表 2 が示すように，閾値が 0.5 のときに高い再現率を持つことから決定的な識別要因が存在しているものと考えられる．それを裏付けるように，バイトコードモジュールの不具合予測では適切な閾値を設定してやれば，ソースコードモジュールの不具合予測の精度を上回る可能性があることが図 3，図 4 からわかる．さらに言うならば，実は適合率と再現率はトレードオフの関係にあるので，閾値の変化によって適合率と再現率の大きく変わるバイトコードモジュールの不具合予測は状況に合わせた使用が可能であると考える．

今回の実験は単一のプロジェクト内の複数リビジョンで行っているためこのような結果が出たが，複数のプロジェクト間で不具合予測を行った場合，バイトコードモジュールによる不具合予測が勝るものと予想できる．根拠としては，別プロジェクトにおいては開発者が違えば文脈も違い，また変数名や関数名も違うためユニークな情報が役に立ちにくいと考えるからである．一方，バイトコードは余分な情報を排除した純粋な命令列であるため，違うプロジェクト間においても機能すると考えられる．もしこの考えが成り立つならば，あるプロジェクトで蓄積した学習情報を違うプロジェクトの初期のリビジョンでも適用することも可能となり，さらに適用も比較的簡単であるので，今後への大きな貢献となることが期待できる．

## 7. 結言

本研究では，スパムフィルタ CRM114 を用いた Fault-prone フィルタリングの手法を用いて，バイトコードモジュールの不具合予測を行う実験を行った．また，ソースコードモジュールに対しても同様の不具合予測を行い，バイトコードモジュールの予測結果と比較することで予測の精度の違いに関して調べた．実験の結果，バイトコードモジュールを Fault-prone フィルタリングに適用して不具合予測を行うことによって，従来の予測をわずかに上回る，もしくは同等な精度を得ることが可能であることが示された．

今後の課題としては，より多くのプロジェクトに対して実験を行い，バイトコードモジュールによる不具合予測が他のプロジェクトにおいても高い精度を得ることができるかを調べることが考えられる．また，プロジェクト間に対しても実験を行うことで，あるプロジェクトのバイトコードモジュールの学習結果を他プロジェクトに適用できるかどうかを調べることを今後行うべき課題としたい．

## 参考文献

[1] P. Bellini and I. Bruno and P. Nesi and D. Rogai “Comparing Fault-Proneness Estimation Models”, *Proc. of 10th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 205–214, 2005.

[2] Lionel C. Briand and Walcelio L. Melo and Jurgen Wust “Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects”, *IEEE Trans. on Software Engineering, Vol. 28, No. 7*, pp. 706–720, 2002.

[3] Giovanni Denaro and Mauro Pezze “An Empirical Evaluation of Fault-Proneness Models’, *Proc. of 24th International Conference on Software Engineering*, pp. 241–251, 2002.

[4] Lan Guo and Bojan Cukic and Harshinder Singh “Predicting Fault Prone Modules by the Dempster-Shafer Belief Networks”, *Proc. of 18st International Conference on Automated Software Engineering*, pp. 249–252, 2003.

[5] Taghi M. Khoshgoftaar and Naeem Seliya “Comparative Assessment of Software Quality Classification Techniques: An Empirical Study”, *Empirical Software Engineering*, pp. 229–257, 2004.

[6] Tim Menzies and Jeremy Greenwald and Art Frank “Data Mining Static Code Attributes to Learn Defect Predictors”, *IEEE Transactions on Software Engineering*, pp. 2–13, 2007.

[7] Naeem Seliya and Taghi M. Khoshgoftaar and Shi Zhong “Analyzing Software Quality with Limited Fault-Proneness Defect Data”, *Proc. of 9th IEEE International Symposium on High-Assurance Systems Engineering*, pp. 89–98, 2005.

[8] Osamu Mizuno and Shiro Ikami and Shuya Nakaichi and Tohru Kikuno “Fault-Prone Filtering: Detection of Fault-Prone Modules Using Spam Filtering Technique”, *Proc. of 1st International Symposium on Empirical Software Engineering and Measurement*, 2007.

[9] Osamu Mizuno and Tohru Kikuno “Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter”, *Proc. of 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the founda- tions of software engineering*, pp. 405–414, 2007.

[10] S. Chhabra and William S. Yerazunis and C. Siefkes “Spam filtering using a Markov random field model with variable weighting schemas”, *Proc. of 4th IEEE International Conference on Data Mining*, pp. 347–350, 2004.