卒 業 研 究 報 告 書

題　目　　バイトコードとソースコードにおける
不具合予測結果の差異の分析

指導教員　　水野 修　教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　14122007

氏　　名　　植村 佳治

平成30年2月14日提出

バイトコードとソースコードにおける不具合予測結果の差異の分析

平成 30 年 2 月 14 日　　14122007　植村 佳治

概　要

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，それらのモデルを構築するために，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る．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．バイトコードはソースコードをコンパイルして得られる写像であることから，本質的な内容は同じである．しかし，両者を用いた不具合予測の結果には違いがあることがわかっている．具体的な実験として，対象とする二つの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得た結果のうち,Fault-prone モジュール予測結果が同じものと違うものの差異について分析する．

この実験を通して，バイトコードとソースコードの予測結果の傾向を示し，また，バイトコードとソースコードの予測結果の差異に起因するメトリクスの確認，および，２つのプロジェクト間での予測結果の差異に起因するメトリクスが存在することを示した．

目　次

1. 緒言

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，有用であるだろうという考えのもとに採用した．バイトコードはソースコードをコンパイルして得られる写像であることから，本質的な内容は同じである．しかし，両者を用いた不具合予測の結果には違いがあることがわかっている．そして，ソースコードによる予測と比べてどのように差異があるのかを，U 検定によって，２つのプロジェクトを用いて分析する．本研究では，バイトコードとソースコードの Fault-prone モジュール予測において，予測結果に差異があること，そして，予測結果が異なるものに共通する

メトリクスが存在することを示した．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章では研究手法を説明する．本実験に必要な前提事項や対象となるプロジェクト，および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．４章では，本実験の条件と評価尺度について言及し，結果を述べる．５章では，本実験の結果を基に考察を行う．そして６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

従来行われてきた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，有用であることが挙げられる．バイトコードはソースコードをコンパイラによって変換した写像として考えることができるが，本質的には含んでいる情報は同一である．しかしながら，それぞれを対象としてテキスト情報ベースの不具合予測モデルを構築すると，その予測結果が異なる現象が観測されている．本研究では，この予測結果の違いが何によって生じるのかを解明すべく，同一のデータから生成するソースコードとバイトコードに対してFault-prone filteringを適用する．そして，そこで得られた予測結果について統計分析を行い，違いに対する分析を行う．

3. 研究手法

3.1 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．

3.1.1 Fault-prone フィルタリング

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.

(1) スパムフィルタ

汎用的な用途として，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.

(2) CRM114

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.

3.1.2 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．

3.1.3 Git

プログラムのソースコードなどの変更履歴を記録・追跡するための分散型バージョン管理システムである．個人的なソフトウェアの開発から大規模なオープンソースソフトウェアの開発まで，様々なプロジェクトが Git 上で管理されている．(注 1)

3.1.4 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．(注 2)

3.1.5 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．

(注 1)：https://github.com
(注 2)：http://bxr.su/OpenBSD/usr.bin/lex

3.1.6 マンホイットニーの U 検定

これは，ノンパラメトリックな統計学的検定の 1 つであり，独立な 2 組の標本の有意差検定として用いられ，変数は順位としてとれば，すなわち 2 つを比較してどちらが大きいかが分かっていればよい．二つの観察された分布の間の重なりの度合が偶然で期待されるよりも小さいかどうかを,「両標本が同じ母集団から抽出された」との帰無仮説に基づいて検定する方法である．

また，正規分布の混合といった非正規分布については t 検定よりも有効性が高く，正規分布についても t 検定に近い有効性を示す．

3.2 実験

本章では，実験の対象・準備・方法について説明する．

本実験の流れを図 3.1 に示す．Fault-prone フィルタリングの入力をバイトコード，ソースコードとし，Fault-prone filtering を適用する．そして得られた結果のうち，バイトコードとソースコードで，Fault-prone か Non-Fault-prone かの予測結果が同じものと違うものとで比較を行う．また，これらが一般性を持つかを調べるために，Apache Ant，Apache Forrest の 2 つのプロジェクトに対して比較を行う．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．

3.2.1 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．

Apache 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.1.3，rev.1.4，rev.1.5，rev.1.6，rev1.7 の 5 つのリビジョンを対象とする．

Apache Forrest

Java で開発されたパブリッシュフレームワークで，複数のデータソースを入力にとり、1 つまたは複数のフォーマットに変換したデータを出力することができる．静

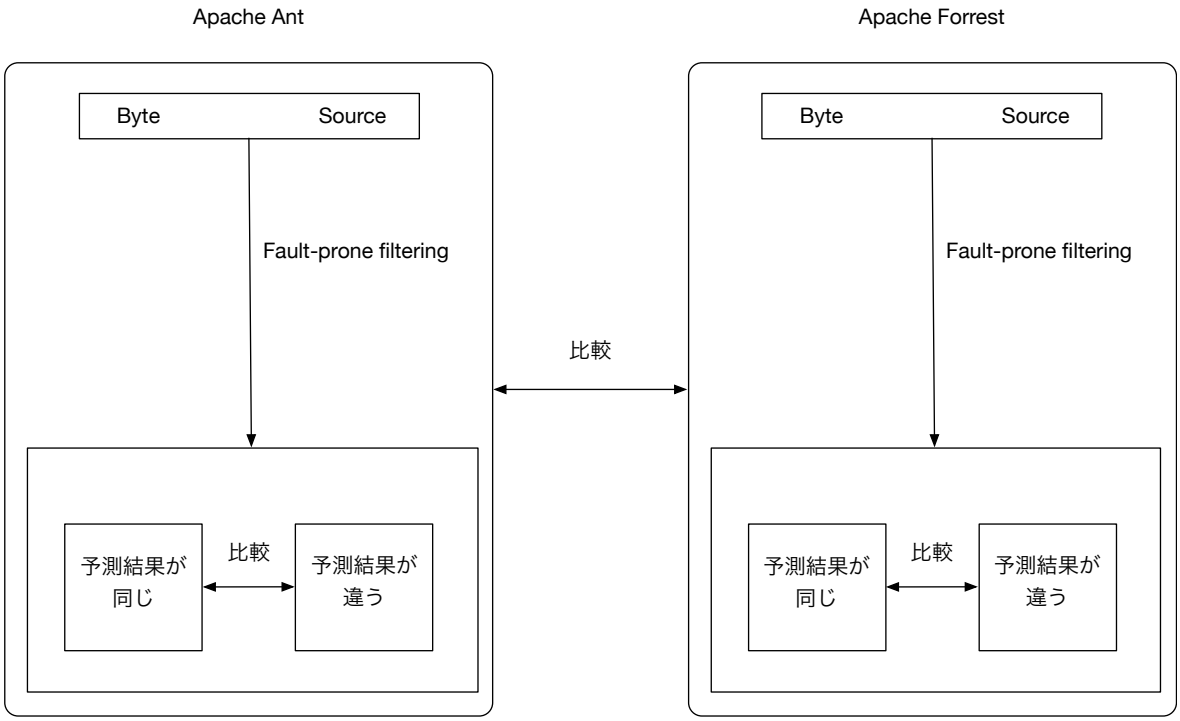
Apache Ant
Apache Forrest
Byte
Source
Fault-prone filtering
比較
予測結果が
同じ
比較
予測結果が
違う
Byte
Source
Fault-prone filtering
予測結果が
同じ
比較
予測結果が
違う

図 3.1　実験概要

的なコンテンツを処理させることができるほか，動的な処理を行うサーバとしても利用することが可能である．

今回の実験では rev.0.6，rev.0.7，rev.0.8 の 3 つのリビジョンを対象とする．

3.2.2 研究設問

実験は以下の研究設問に回答するために設計される．

設問 1:ソースコードとバイトコードによる Fault-prone Filtering の予測結果に差が見られるか？

設問 2:ソースコードとバイトコードで予測結果が食い違ったモジュールに共通する特徴はあるか？

3.2.3 実験の準備

実験を行う前にまず，”THE APACHE ANT PROJECT”(注 3) から Apache Ant のバイトコードとソースコードを，”THE APACHE FORREST PROJECT”(注 4) から Apache Forrest 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，Apache Forrest 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．

(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 ファイルもフォーマット上は，zip ファイルと同じ構造なので，zip ファイルを扱うのと同様に処理することができる．よって，python にて zip ファイルを展開するメソッドを作成して展開した．

(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，

(注 3)：http://ant.apache.org/git.html
(注 4)：https://forrest.apache.org

クラスファイルに逆アセンブル(注 5)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．

(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．手段としては，1 つのソースファイルに対して，正規表現を用いて，2 つ目またはそれ以上のクラス定義が存在するかを調べて，存在する時はクラス名に".java_"の拡張子を付して，ソースファイルを生成する．よって，ソースコードモジュールの実験データは全てこの".java_"を拡張子に持つソースファイルとなる．

(4) 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(注 6)を抽出するトークナイザとソースコードから単語(注 7)を抽出するトークナイザの 2 つを作成する．

(5) 実験用データの抽出

実験に必要なバイトコード，およびソースコードは PROMISE に記されているモジュールのみである．PROMISE に記されている各モジュールのパスには，Apache

(注 5)：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
(注 6)：オペコードと 1 対 1 対応している人間が可読な文字列
(注 7)：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．

Ant,Apache Forrest から取得・展開したものの中に存在しないものや，実験には必要のないものもある．存在しないモジュールを実験データに用いると不具合が生じるため，実験に必要なモジュールのみをバイトコード，ソースコードのそれぞれについて抽出し用意する．

3.2.4 実験方法

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．
(5) 予測結果より，バイトコードとソースコードで予測結果が違ったものと同じものの２つに分類し，それぞれのモジュールの情報を PROMISE のデータから抽出する．
(6) 分類した２つのデータのそれぞれのモジュールのメトリクスに対して，マンホイットニーの U 検定を行う．

以上の (1)〜(6) 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．これを Apache Ant，Apache Forrest の２つのプロジェクトに対して行うことをもって実験の方法とする．

4. 結果

本章では，実験によって得られた結果を示す.

4.1 実験の条件

本実験に行う上での条件を次に示す.

4.1.1 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.

4.2 評価尺度

4.2.1 実験結果の凡例

表 4.1 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 4.1 では便宜上，Fault-prone を FP，Non-Fault-prone を NFP と省略する．また，得られた予測結果に対して，精度（Accuracy），適合率（Precision），再現率（Recall）の 3 つの指標を算出する.

4.2.2 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.

$$Accuracy = \frac{N_1 + N_4}{N_1 + N_2 + N_3 + N_4} \tag{4.1}$$

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.

表 4.1　実験結果の凡例

		予測	
		FP	NFP
実測	FP	N_1	N_2
	NFP	N_3	N_4

4.2.3 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．

$$Precision = \frac{N_1}{N_1 + N_3} \tag{4.2}$$

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．

4.2.4 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．

$$Recall = \frac{N_1}{N_1 + N_2} \tag{4.3}$$

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．

4.2.5 マンホイットニーの U 検定

マンホイットニーの U 検定の結果の p 値が有意水準以下であれば有意差があると言える．本実験では 5 %，1 % と設定した時の結果を示す．

4.3 実験結果

4.3.1 Apache Ant の実験結果

[**3.3 実験方法**] によって得られた Fault-prone フィルタリングの最終的な予測結果を表 4.2，表 4.3 に，実験に用いたデータのバイトコードとソースコードの最終学習データ，最終予測データに含まれるユニークなトークンの数を表 4.4 に，U 検定の結果を表 4.5 に示す．

表 4.2　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	124	42
	NFP	211	368
精度（accuracy）		0.6604	
適合率（precision）		0.3701	
再現率（recall）		0.7470	

表 4.3　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	127	39
	NFP	395	184
精度（accuracy）		0.4174	
適合率（precision）		0.2433	
再現率（recall）		0.7651	

表 4.4　バイトコードモジュールとソースコードモジュールに含まれるユニークなトークン数

	バイトコード	ソースコード
最終学習データ	16987	101094
最終予測データ	9007	41829

表 4.5　各メトリクスの中央値と U 検定の結果

metrics	same	different	有意差$^{*:level=0.05}_{**:level=0.01}$
wmc	7	7	-
dit	3	2	-
noc	0	0	-
cbo	7	6	-
rfc	20.5	26	-
lcom	6	8	-
ca	2	1	**
ce	4	5	-
npm	6	6	-
lcom3	0.8	0.9	*
loc	136.5	156	-
dam	1	0.9	*
moa	0	0	**
mfa	0.7	0.7	-
cam	0.4	0.4	-
ic	0	0	-
cbm	0	0	-
amc	16.4	16.3	-
max_cc	3	3	-
avg_cc	1.2	1.1	-
bug	0	0	**

表 4.2，表 4.3 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．表 4.2 はバイトコードモジュールを予測したとき，表 4.3 はソースコードモジュールを予測したときの精度，適合率，再現率の最終結果である．表 4.2，表 4.3 からは，バイトコードモジュールとソースコードモジュールの実測が NFP のうち，予測の FP と NFP を入れ替えたような結果となっていることがわかる．なので，予測の NFP が大きいバイトコードモジュールの方が精度は高くなる．一方，予測の FP が大きいソースコードモジュールの方が適合率は高くなる．次に，表 4.4 について説明する．表 4.5 は，バイトコードモジュールとソースコードモジュールで，予測結果が同じものと違うものの２標本の各メトリクスに対して，マンホイットニーの U 検定を行った結果を示している．”same”は分類結果が同じもの，”different”は分類結果が違うものであり，値はそれぞれのメトリクスの中央値である．”有意差”は，有意水準を５％，１％とした時の有意差の有無を示したものであり，”*”は有意水準を５％とした時に有意差がある，”**”は有意水準を１％とした時に有意差があることを示している．そして，以下のメトリクスに有意差が確認された．

ca(Afferent Couplings)

これは，パッケージ内のクラスに依存するパッケージ外のクラス数，すなわち，被依存クラス数である．この値が大きいと，自身の仕様変更が他に影響を及ぼしやすいことを表す．予測結果が同じものの方が高い傾向にある．

lcom3(Lack of cohesion in methods3)

これは，クラスの凝集度を示しており，低いことは，高い凝集度が実現されておりいい設計と言える．逆に高いとカプセル化の減少および複雑性の増加を示し，それによりエラーの可能性が増加することを示す．予測結果が異なる方が高いのは，凝集度が低く，複雑性が増加しているからであると考えられる．

dam(Data Access Metrics)

これは，クラスで宣言された属性の総数に対する private または protected として宣言された属性の比を示しており，高い方が良い設計と言える．予測結果が異なる方が高いのは，アクセスを厳しくする分，複雑性が増加しているからであると考えられる．

moa(Measure of Aggregation)

これは，開発者が定義したクラスが変数として定義された数を示している．中央値が互いに 0 でありながら，有意差がある理由としては，開発者が定義したクラスであるから，含まれるモジュールに偏りなどがあると予測結果に影響を与えることが考えられる．

bug

これは，モジュールに含まれるバグの数を示す．ソースコードはユニークなトークン数が多いことから，NFP を誤って FP と予測することが多いが，バイトコードは FP は FP である決定的な特徴を持ち，その他は NFP と似通うことから，有意差が生じると考えられる．

これらの結果から，”ca”，”moa”,”bug”は有意水準 1 %，”lcom3”,”dam”は有意水準 5 %と設定した時，有意差が見られることがわかる．

4.3.2 Apache Forrest の実験結果

Apache Ant と同様に示す．最終的な予測結果を表 4.6，表 4.7 に，実験に用いたデータのバイトコードとソースコードの最終学習データ，最終予測データに含まれるユニークなトークンの数を表 4.8 に，U 検定の結果を表 4.9 に示す．

対象としたプロジェクトが Apache Forrest であること以外は，[**4.3.1 Apache Ant の実験結果**] と同様であるため，説明を省略する．表 4.6，表 4.7 から，バイトコードモジュールとソースコードモジュールの予測結果にほとんど差がないが，特筆すべき事としては，バイトコードモジュールの実験において，実測が FP であるモジュールを一つも正しく予測できなかったことがわかる．ただし，バイトコードモジュールの予測において，バグをバグと正しく予測できなかったため，精度，適合率，再現率は算出できなかった．

表 4.9 は，バイトコードモジュールとソースコードモジュールで，予測結果が同じものと違うのものの 2 標本の各メトリクスに対して，マンホイットニーの U 検定を行った結果を示している．有意差が見られるメトリクスに着目して考察する．また，[**4.3.1 Apache Ant の実験結果**] にて既出のメトリクスについては，メトリクスの説明は省略する．

rfc(Response for a Class)

これは，あるクラスに存在する，メソッド呼び出しの種類の数．値が大きいほど，

表 4.6　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	0	2
	NFP	4	26

精度（accuracy）	-
適合率（precision）	-
再現率（recall）	-

表 4.7　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	1	1
	NFP	6	24

精度（accuracy）	0.7812
適合率（precision）	0.1429
再現率（recall）	0.5000

表 4.8　バイトコードモジュールとソースコードモジュールに含まれるユニークなトークン数

	バイトコード	ソースコード
最終学習データ	1329	3155
最終予測データ	964	1824

表 4.9　各メトリクスの中央値と U 検定の結果

metrics	same	different	有意差$^{*:level=0.05}_{**:level=0.01}$
wmc	4	11	-
dit	3	3	-
noc	0	0	-
cbo	10	17	-
rfc	24	53	*
lcom	1	20	-
ca	1	0	-
ce	9	16	-
npm	3	6	-
lcom3	0.75	0.75	-
loc	138	522	**
dam	1	1	-
moa	0	1	*
mfa	0.7	0.5	-
cam	0.5	0.4	-
ic	0	0	-
cbm	0	0	-
amc	29.7	56.6	-
max_cc	2	2	-
avg_cc	1	0.9	-
bug	0	0	-

他のクラスへの依存度が高く，処理も複雑であることを表している．中央値を見ると，予測結果が異なる方が，高くなっている．複雑性が増加する分，予測も難しくなり，有意差が生じたと考えられる．

loc(Line of Code)

これは，モジュールのコード行数を示す．中央値を見ても，予測結果が異なる方が高いことがわかる．行数が多いと，それだけユニークなトークン数が増え，複雑性も増加する．また，バイトコードは最低限の命令列であるのに対し，ソースコードは変数名や記号など多くの情報が含まれる．よって，なのでコードの行数は予測結果の差異にも影響を与えると考えられる．

moa(Measure of Aggregation)

これは，予測結果が異なる方が中央値が高いことがわかる．

これらの結果から，”loc”は有意水準 1 %，”rfc”,”moa”は有意水準 5 %と設定した時，有意差が見られることがわかる．また，”moa”メトリクスのみが，両プロジェクトで共通して有意差に起因しているという結果が得られた．

5. 考察

本章では，実験に対する考察を行う．

5.1 研究設問に対する考察

設問 1:バイトコードとソースコードによる Fault-prone filtering の予測結果に差が見られるか？

ソースコードの実測が NFP のものの予測結果の傾向に大きな差は見られる．この理由をバイトコードとソースコードの特徴から考察する．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ことが表 4.4 からわかる．逆に，ソースコードにおいては，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なものが多くなる．Fault-prone filtering においては，一般に不具合ありとして学習されるものの方が少なく，不具合なしのものが多い．そのため，ユニークなトークンが少なくなることで，学習時に不具合の特徴を表すトークンが少なくなることが予想される．この結果，NFP を NFP と予測はできるものの，FP を FP と予測できなかった例が多いものと考えられる．ただし，Apache Forrest はデータ数が少ない上に，実測が FP のモジュールが少ないので，実測，予測ともに NFP が多い傾向になったと考えられる．また，バイトコードの方はユニークなトークンが少ないことに加えて，データ数が少なかったことから，FP を正しく予測するだけの十分な学習が行えなかったことが理由として考えられる．しかし，データ数が少ないながらも，ソースコードモジュールの予測結果では，正しく 1 つの FP を予測できていることがわかる．こうした理由により，バイトコードとソースコードの予測結果に差が出たと考えられる．

設問 2:バイトコードとソースコードで予測結果が食い違ったモジュールに共通する特徴はあるか？

予測結果が食い違ったモジュールには，”moa”メトリクスが高いという共通する特徴が見られた．この理由として考えられるのは，ソースコードでは開発者が定義しているクラスとして，ユニークなトークンとなり得るものの，バイトコードではそうしたトークンは存在しないため，予測結果が異なるものと考えられる．

5.2 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 と Apache Forrest の２つを対象のプロジェクトとして特定の結果を得た．しかし，Apache Forrest に関しては，モジュール数が少なく，無学習状態から学習を始める本実験においては，有意な結果を得られる実験データでなかった可能性がある．よって，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 も Apache Forrest も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差が出る可能性がある．

- **U 検定の適用条件**

U 検定は，２標本の大きさが同程度で，データ数があまりに少ないと適用できない．Apache Forrest の予測結果には偏りがあり，データ数が少ないことから２標本の大きさに差が生じてしまい，U 検定に適した標本でなかった可能性がある．

5.3 今後の課題

十分な数のデータがある複数のプロジェクト間で不具合予測を行った場合，予測結果の差異の要因を確認できると考える．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，特にソースコードにおいてはユニークな情報が多くなる．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．しかし，違うプロジェクト間でもモジュールの持つメトリクスは測定できるため，より有意差の生まれる予測結果が得られると予想できる．その予測結果に対して，差異の要因となっているメトリクスを分析できれば，バイトコードでの Fault-prone モジュール予測をより実用的にするために，着目すべきメトリクスが発見でき，今後への大きな貢献となることが期待できる．

6. 結言

本研究では，スパムフィルタ CRM114 を用いた Fault-prone filtering の手法を用いて，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不具合予測を行う実験を行った．また，得られた結果から予測結果の差異を分析した．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．また，バイトコードとソースコードのそれぞれのモジュールの予測結果の差異には，有意差を生むメトリクスが存在していることも確認できた．今後の課題としては，十分なデータ数を持つ，より多くのプロジェクトに対して実験を行い，バイトコードおよびソースコードモジュールによる不具合予測の結果の差異に起因するメトリクスが，他のプロジェクトにおいても有意差の要因とな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，そのメトリクスが差異の要因である一般性を検証することを今後行うべき課題としたい．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学・人間科学系水野修教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻原田 禎之先輩，黒田 翔太先輩，中川 要先輩，西浦 生成先輩，田中 健太郎先輩，小林 勇揮先輩，近藤 将成先輩，洪浚通先輩，情報工学課程 北村 紗也加さん，中村 勝一君，広瀬 早都希さん，渡邊 大輝君をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

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.