修 士 論 文

題　目　　複数のソフトウェア不具合予測手法の
定量的比較分析

主任指導教員　　寶珍　輝尚　教授

指導教員　　水野　修　准教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　11622025

氏　　名　　中井　道

平成25年2月8日提出

様式 2 号

学位論文の要旨（和文）

平成 25 年 2 月 8 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　　情報工学専攻
平成 23 年入学
学生番号　　11622025
氏　　名　　中井　道　㊞

（主任指導教員　　寶珍　輝尚　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

複数のソフトウェア不具合予測手法の定量的比較分析

2.　論文内容の要旨（400 字程度）

ソフトウェアにおける不具合の予測はソフトウェア工学の中でも最も重要な問題の一つである．従来，多くの研究が成されてきてはいるが，同一のプロジェクトに対して同一の条件で予測手法間の定量的な比較を行った研究はほとんど無い．これは，同一のデータに対して複数の手法を適用することが難しいことも要因と考えられる．我々は共有データリポジトリで公開されたデータを元に，新たなデータをこれに追加することで複数の手法を同一の条件で実行することに成功した．本論文では複数の手法での予測結果の違いについての考察を行う．

比較実験には，3 つの不具合予測手法を利用した：Random Forest 手法，Fault-prone フィルタリング手法，PMD を用いた Fault-prone フィルタリング手法，である．これらの手法は入力として与えるのはソフトウェアのモジュールであるものの，利用するメトリクスは異なる．本実験では，複数のプロジェクトを用いて，プロジェクト横断的な予測を実施した．これにより，手法とプロジェクトの違いが予測結果にどのような影響を及ぼすかを評価した．

Quantitative Evaluatiion of Fault-prone Module Prediction Techniques

2013 11622025 *Michi NAKAI*

Abstract

Prediction of Fault-prone modules in software is one of the most important areas in software engineering. Although much research has been done so far, not so many studies on quantitative comparison of such Fault-prone module prediction approaches have been done. One reason is the difficulty of the comparison such as preparing the same condition and contexts for experiments. We obtained the data of several OSS projects from a public data repository. Using this data, we can conduct experiments with the same condition and contexts.

For the comparison, we used three approaches for Fault-prone module prediction: the Random Forests, the Fault-prone filtering, and Fault-prone filtering with PMD. The Random Forests technique is one of the best machine learning methods for Fault-prone module prediction. Fault-prone filtering and Fault-prone filtering with PMD are our original approaches, which use text information in the source code or the output of the PMD.

The result of comparison showed that our original approaches have better prediction results than the Random Forest approach in many cases. We also found that the data from specific projects make lower the prediction accuracy in any prediction approaches.

目　次

1. 緒言

ソフトウェアの開発において，大規模な開発になればなるほどモジュールに不具合が混入することは避けられないこととなる．そのため，いかにして効率的に不具合を発見し，取り除くかがソフトウェア開発を行う上で重要な鍵となる．

そのために，不具合を含みそうなモジュール（Fault-proneモジュール）を予測することができれば，不具合を効率よく取り除くことができ，ソフトウェア開発にかかるコストを削減することができると考えられている．そのため，いかにFault-proneモジュールを高い精度で予測するかという研究が数多く行われている．

従来の研究では，ソフトウェアの不具合はソフトウェアの構造の複雑さに関連していると仮定し，ソフトウェアの複雑さを定量化したメトリクスを利用することによって，不具合を推定するモデルを構築していた．中でもRandom Forest[1]は非常に優秀な機械学習であり，Fault-proneモジュールの予測にも利用できることが知られている[2].

これに対し，我々は異なるアプローチとしてFault-proneフィルタリング[3]という手法，及び，静的解析ツールの出力を利用したFault-proneフィルタリング[4, 5]という手法を提案している．これらの手法は従来手法と違い，ソースコードや静的解析ツールから得られる警告文といったテキスト情報を用いて予測を行う．

しかし，このように数多くの研究が行われているにもかかわらず，それらの手法を定量的に比較した研究は少ない．その1つの要因となっているのは同一の実験環境を用意することが困難であるからである．我々は公共のリポジトリからオープンソースのプロジェクトを入手し，同様の実験環境を用意した．

そこで本研究ではRandom Forest，Fault-proneフィルタリング，及び静的解析を応用したFault-proneフィルタリングの3つの手法について定量的比較を行う．

本論文の以降の構成を述べる．第2章では，比較の対象となるRandom Forest，Fault-proneフィルタリング，及び静的解析を応用したFault-proneフィルタリングの3つの不具合予測手法について説明する．第3章では，各予測手法の比較実験を行う．実験対象や実験方法，評価指標など，そして実験結果について説明する．第4章では，実験結果に対する考察を述べる．最後に第5章では，本研究のまとめ及び，今後の課題について述べる．

2. 準備

本章では本研究で比較する 3 つの予測手法を紹介する．

2.1 ソフトウェアメトリクスを用いた予測 (Random Forest)

2.1.1 概要

Random Forest[1] は，集団学習により高精度の識別，回帰，クラスタリングを行う機械学習アルゴリズムである．研究 [2] において Random Forest が Fault-prone の予測に非常に有効な手法であるということが実証されたため，代表的な不具合予測手法として比較の対象とした．この手法では決定木の集団学習することで高精度の分類や予測を行うことができる．

(1) 決定木

決定木とは木構造の条件分岐を用いて入力パターンをクラス分類する分類器の一種である．データから決定木を作成する機械学習を決定木学習というが，これはデータマイニングの分野でも良く用いられる．その場合，決定木の葉が分類を表し，葉に至るまでの通り道となる枝がその分類の特徴を表す．決定木は目的変数に影響が大きい変数や境界値，順序を算出する．

決定木の分岐基準としては CART(Classification and Regression Trees) を用いる．CART は分割統治アルゴリズムの一種である．CART では各ノードでの分岐基準となる説明変数は，木全体の Gini 係数を一番減少させる説明変数という基準で選択する．Gini 係数とは，本来は社会における所得分配の均衡，不均衡を表す尺度である．Gini 係数は 0 から 1 の値をとり，0 に近いほど平等な社会であり，反対に 1 に近いほど格差が大きく，不平等 (高所得者への所得集中) な社会であるとされている．この Gini 係数はデータマイニングにおけるデータ分類にも利用できる．Gini 係数が 0 の状態をこれ以上分割できないを意味する．逆に分類を行う前のデータの Gini 係数は 1 に近い．そのため，CART では少しでも多くの Gini 係数を減少させて 0 へ近づけるように説明変数を選択するのである．Gini 係数は式 (2.1) で表される．ここで $p(k|A)^2$ はノード A でクラス k をとる確率を表す．

$$Gini\text{ 係数} = 1 - \sum_{k} p(k|A)^2 \tag{2.1}$$

決定木には分類木と回帰木の 2 つに分けられる．分類木は目的変数が分類変数のものである．回帰木は目的変数が連続数値変数のものである．なお，本研究では不具合があるかないかの 2 値の分類を行うため，分類木として用いる．

(2)　集団学習

集団学習とはサンプルや重みの異なる複数の学習モデルを生成して，各モデルの評価結果を統合することで精度や汎用性をを向上させる機械学習のアルゴリズムである．結果の統合方法に関しては，分類の場合は多数決を用い，回帰の場合には平均値を用いる．

2.1.2　アルゴリズム

Random Forest は決定木の集団学習によって分類や予測を行うが，大きく分けて学習と評価の 2 つのステップに分けられる．

(1)　学習ステップ

まず，ブーストラップサンプリングを用いて，学習データから重複を許してランダムに B 組のサンプル集合を抽出する．そして各サンプルにおいて M 個の説明変数の中から m 個の変数をランダムに抽出して決定木を作成する．その結果 B 本の決定木が作成される．

(2)　評価ステップ

学習ステップで作成した B 本の全ての決定木で調査したいデータを分類，回帰予測を行う．そして得られた B 個の結果を分類なら多数決で，回帰予測なら平均で統合する．

2.1.3 特徴

Random Forest の長所としては次の 4 つなどが挙げられる．

- サポートベクターマシン (SVM) などの他の分類器と比較して分類，予測が高精度である．
- 説明変数が数百，数千など膨大な数になっても効率的に作動する．
- 目的変数に対する説明変数の重要度が推定できる．
- データに欠損値が存在しても動作可能である．

逆に短所は意味のある説明変数が意味のない説明変数よりも極端に少ない場合にうまく評価が行えないことなどである．

2.2 Fault-prone フィルタリング

ソフトウェアの不具合は従来の研究では主にソフトウェアの構造の複雑さに関連していると仮定し，ソフトウェアの複雑さを定量化したメトリクスを利用することによって，不具合を推定するモデルを構築していた．

これに対し，我々は Fault-prone フィルタリング [3] という手法を提案している．この手法では，不具合はソフトウェアモジュール中に含まれる語や文脈に関連していると考え，与えられるモジュールをテキスト情報として扱う．そして，スパムフィルタリングで利用されるベイズ理論を用いて，あらかじめ不具合が混入していたものを分類して学習させた辞書に基づいて，あるモジュールに対する不具合混入確率を計算する．この確率に基づいて，与えられたモジュールが Fault-prone モジュールであるのか，否かを判定する．

2.2.1 概要

Fault-prone フィルタリングとはモジュールの不具合の検出にスパムフィルタリングの理論を導入したものである．

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，メールの特徴を学習するステップと，学習した結果を用いてメールを分類するステップの 2 ステップで構成されている．

(1) 学習ステップ

1. 学習用に ham メールなのか spam メールなのかが予め分かっているメール群を用意する．
2. 各メールから単語（トークン）を抽出し，それらのトークンを登録した辞書を作成する．この際に，ham メールから抽出したトークンと spam メールから抽出したトークンは別々の辞書に登録する．したがって，ham メール用と spam メール用の 2 つの辞書が作成されることになる．

(2) 分類ステップ

3. 作成した 2 つの辞書を用いて，新しく受信したメールを ham メールであるか，spam メールであるかに分類する．

このようなスパムフィルタの動作は，ham メールと spam メールには文中に含まれる語や文の傾向といったものにそれぞれ異なる特徴が見られるといった仮定に基づいている．

水野らは，この仮定が不具合を含むソースコードと含まないソースコードに対しても当てはまると考え，スパムフィルタを Fault-prone モジュールの検出に導入したものを提案し，「Fault-prone フィルタリング法」と名付けた [6][7]．Fault-prone フィルタリングでは，不具合を含まないモジュールを ham メール，不具合を含むモジュールを spam メールとそれぞれ対応させ，スパムフィルタの処理と同様に辞書を学習していく．そして，学習した辞書を用いることで，新しいモジュールが不具合ありモジュールである確率を計算し，Fault-prone モジュールであるかの判定を行う．

Fault-prone フィルタリングに対しては，これまでにも，いろいろな観点から研究が進められている [6][7][8][9].

2.2.2 Fault-prone フィルタの実装

本研究では，文献 [9] で用いられた Fault-prone フィルタを使用する．この Fault-prone フィルタはベイズの定理を利用して作成された単純なものである．学習ステップおよび分類ステップのアルゴリズムは文献 [10] で提案されたスパムフィルタの基礎理論を踏襲したものとなっている．次にトークン分け及び，学習，分類の両ステップについて説明する．

トークンの分割

Fault-prone フィルタリングの入力となるモジュール，本研究ではソースコードを用いるが，それらは全てトークン単位に分割される必要がある．

以下は Java を例にして説明する．ソースコードモジュールは Java のコードが書かれている部分と，コメントが書かれている部分で構成されている．しかし，コメント部分はコードの部分と違い，厳密な構文が決まっておらず人間が自由に記述できるということに注意する必要がある．制約が緩いため，ダブルクォート等が正しく綴じられていない場合などが生じる．また，アポストロフィなのかシングルクォートで囲まれた文字列なのか判定が困難であると言ったような問題も生じる．そのため，ダブルクォートやシングルクォートで囲むといった規則が正しく機能しない可能性がある．したがって，コメントであるか否かでトークンの分割規則を変える必要がある．

そのためには，まずコメントである部分を検出する必要がある．Java 言語には次の 3 種類のコメントが存在する．

- “//”で始まるコメント
- “/*”で始まり，“*/”で終わるコメント
- “/**”で始まり，“*/”で終わるコメント

よって，モジュール中にこれらが検出されたときにコメントであると判断し，コメント用の規則を使用することとする．それ以外の部分はコード用の規則を使用することとする．

コードの部分のトークン分割規則は次のように定義する．

- アルファベット，数字，ドットからなる文字列
- Java 言語の演算子
- シングルクォートで囲まれた文字列
- ダブルクォートで囲まれた文字列

コメントの部分のトークン分割規則は上記の理由により，次のように定義する．

- アルファベット，数字，ドットからなる文字列
- Java 言語の演算子

2.2.3 学習ステップ

このステップは後の分類ステップで必要となる辞書を作成するステップである．

ここでは不具合を含むモジュールと不具合を含まないモジュールからトークンを抽出し，それぞれを別々にトークンの出現数を数える．そして，それにより次に定義する 2 つのハッシュテーブルを得る．

- $faulty(t)$：全ての Fault-prone モジュールにおけるトークン t の出現数
- $nonfaulty(t)$：全ての Fault-prone でないモジュールにおけるトークン t の出現数

2.2.4 分類ステップ

このステップでは学習ステップで作成した辞書を用いて，与えられたモジュールが Fault-prone モジュールであるのか，そうでないのかの判定を行う．

このステップは次の 3 つの手順から構成されている．

1. まず，トークン t を含むモジュールが Fault-prone モジュールである確率を $P_{ft|t}$ と定義し，それを得るハッシュテーブルを作成する．不具合を含むモジュールの数を N_{ft}，不具合を含まないモジュールの数を N_{nf}，不具合を含むモジュール中にトークン t が出現する確率を r_{ft}，不具合を含まないモジュール中にトークン t が出現する確率を 2 倍したものを r_{nf} とそれぞれ定義することで，確率 $P_{ft|t}$ は式 (2.4) で表される．

$$r_{nf} = \min\left(1, \frac{2 \times nonfaulty(t)}{N_{nf}}\right) \tag{2.2}$$

$$r_{ft} = \min\left(1, \frac{faulty(t)}{N_{ft}}\right) \tag{2.3}$$

$$P_{ft|t} = \max\left(0.01, \min\left(0.99, \frac{r_{ft}}{r_{nf} + r_{ft}}\right)\right) \tag{2.4}$$

2. 次に，分類されるモジュールからトークンを抽出し，n 個の特徴的なトークン $t_1 \cdots t_n$ を決定する．ここで，これらのトークンについて，式 (2.5) の値が大きいほど特徴的であると定義する．また，今までに一度も出現したことのないトークンであった場合は $P_{ft|t}$ は 0.4 と定義する．

$$\mathrm{abs}\left(0.5 - P_{ft|t}\right) \tag{2.5}$$

しかし，本実験では n 個の特徴的なトークンを全てのトークンとしている．

3. そして，式 (2.6) を用いて，分類されるモジュールが Fault-prone モジュールである確率を計算する．

$$P_{ft} = \frac{\prod_{i=1}^{n} P_{ft|t_i}}{\prod_{i=1}^{n} P_{ft|t_i} + \prod_{i=1}^{n}\left(1 - P_{ft|t_i}\right)} \tag{2.6}$$

判定基準として，式 (2.6) で求めた P_{ft} を用いる．P_{ft} が 0.9 以上の値を取れば，分類されるモジュールは Fault-prone モジュールであると判定する．なお，ここで使用した 0.9 は変更可能な閾値である．

2.3 静的解析を応用した Fault-prone フィルタリング

Fault-prone フィルタリングは，モジュール中から検出したトークンに基づいて分類を行うものであるが，過去の研究 [7] ではモジュール中の全てのトークンを用いて分類が行われていた．また，研究 [8] ではモジュール中のコメントに該当する部分を削除して実験が行われた．そして，研究 [9] では研究 [8] とは逆にコメントに該当する部分のみのトークンに対して実験が行われた．

こうした中，これまで Fault-prone フィルタリングの研究で，学習・分類の対象となるモジュールはテキスト情報であればどのようなものでも構わないにもかかわらず，常にソースコードモジュールが用いられていたことに着目し，他のテキスト情報として，静的コード解析ツールの 1 つである PMD[11] によって得られる警告文を用いることを提案した．

2.3.1 静的コード解析の概要

静的コード解析とはソフトウェアの解析手法の1つであり，ソフトウェアを実際に実行せずに解析を行い，プログラムの問題点や不具合を発見する手法である．

解析は一部例外としてオブジェクトコードに対して行うものがあるが，基本的にソースコードに対して行われる．ソースコードを構造的に解析することで，潜在的な不具合や保守性の低下の原因となるコーディング規約違反，性能劣化の原因となるコードなどを検出する．静的コード解析は，ソフトウェアの安全性や信頼性，保守性などを向上させる手法であり，ソフトウェア品質向上や，保守にかかるコストの削減を目指すことができる．

こういったプログラムの問題点や不具合を発見する作業はテストの役割だと思われがちだが，テストでは発見できないものも存在する．テストは設計どおりにソースコードが組まれているか，つまり不具合が存在しないかを確認する作業とも言えるためである．例えば，ソースコードのコーディング規則違反や，保守性に関する問題も多くがテストでは発見できない．

また，性能劣化の問題などは大抵の場合すべてのソフトウェアを結合し，テストを行うことで発見されるものであるので，そのときになって問題を発見していたのでは，手戻りが非常に多くなってしまう．そのため，静的コード解析を開発早期に用いることでテストの前に各種問題や不具合を発見し，開発にかかるコストや時間を削減することに繋がる．

近年，潜在するセキュリティホールの検出などの必要性が増していることで，静的コード解析が重要視されている．

静的コード解析を行うツールは，非商用，商用様々で，対象となるプログラミング言語も様々なものがある．

2.3.2 静的コード解析ツール PMD

静的コード解析ツールの1つとして，PMD というソフトウェアが存在する．

PMD はオープンソースとして提供されているソフトウェアで，Java を用いて実装されたものである．対象となる言語は Java である．PMD は，Java のソースコードを解析して，未使用の変数や空の catch ブロック，到達不可能なステートメントなど

の，潜在する不具合の原因となるコードを検出するし警告文を出力することができる．PMD には多種多様なルールセットが用意されており，使用するルールセット次第でコーディング規約の検査から，潜在的な不具合を検出まで幅広い用途で使用することができる．

また，独自のルールセットを作成することができるため，特殊なルールを使用することも可能である．なお，ルールセットの作成には XPath か Java のクラスファイル形式を用いる．

PMD にはオープンソースの統合開発環境である Eclipse のプラグインも公開されており，Eclipse で Java 開発を行う際に利用すると効果的に用いることができるとされている．

PMD はソフトウェアの不具合を未然に防ぐことを目的としたツールであるため，得られる警告文は不具合を検出するための強力な特徴であると考えられる．しかし，一方で PMD の警告文は冗長であり，かつ，非常に多くの情報を含むために本当に必要な情報が人間の目に入らないという問題を抱えている．

2.3.3 PMD ルールセット

PMD に標準で用意されているルールセットの中で，本研究では以下に示す 10 個のルールセットを利用する．これらのルールはソフトウェアの品質を調べる観点でよく利用されるものとなっている [12].

Basic

もっとも基本的なルールセット．catch，if，while，switch などの空ブロックや必要のない文字列変換処理，連続して記述された単項演算子，IP アドレスのハードコーディングの検出など，幅広く多種多様なルールが用意されている．

Braces

括弧に関するルールセット．if や else，for や while などステートメントに対して，括弧が使用されてないものを検出するルールが用意されている．

Code Size

コードのサイズに関するルールセット．複雑なメソッドや異常に長いメソッドの検出や，大量の public なメソッドやフィールドを含むクラスなど検出するルールが用意されている．

Coupling

カップリングに関するルールセット．オブジェクト-パッケージ間のカップリングが高すぎたり，不適切なカップリングがないかを検査する．例えば，大量のインポート・ステートメントが定義されているクラスや，使用しているフィールドやローカル変数，戻り値の型の種類が多すぎるクラス，インタフェースではなく実体クラスを宣言しているなどの検出を行うルールが用意されている．

Design

コードの設計に関するルールセット．設計を良くするための様々な原則を検査する．深すぎる if ブロックのネストや，default ブロックのない switch ステートメント，メソッドのパラメータへの代入などを検出するルールが用意されている．

Naming

標準的な名前付けに関するルールセット．ループ変数以外で極端に短い，または長い名前の変数，クラス名と同名のメソッド，先頭が大文字でないクラス名などを検出するルールが用意されている．

Optimizations

最適化に関するルールセット．1 回しか代入されないのに final として宣言されていないローカル変数，ループ内でのオブジェクトの生成，不必要なラッパーオブジェクトの使用などを検出するルールが用意されている．

Strict Exception

例外に関する厳密なルールセット．フロー制御に例外を使用している，Throwable を捕らえている，Exception を投げている，java.lang.Error を継承したクラスなどを検出するルールが用意されている．

Strings

文字列に関するルールセット．重複した文字列定義，String インスタンスの生成，String オブジェクトに対する toString() の呼び出し，StringBuffer インスタンスの生成方法が適切でないところなどを検出するルールが用意されている．

Unused Code

使用されていないコードに関するルールセット．使用されないローカル変数や，private フィールド，メソッドなどを検出するルールが用意されている．

2.3.4 Fault-prone フィルタリングへの PMD の利用

この PMD を利用した Fault-prone フィルタリングを実行するために，テキスト情報として用いられていたソースコードを全て PMD の警告文に置き換える．

そのための処理としてソースコードを Fault-prone フィルタリングに与える前に一度 PMD に与え，得られた警告文を Fault-prone フィルタリングに与える．

トークンの分割

PMD から得られた警告文もソースコードと同様にトークンに分割する必要がある．警告文はソースコードの存在するディレクトリパス，警告の原因が存在する行番号，警告文本体である英文で構成されている．

この中でディレクトリパスと行番号をまず始めに消去する．ディレクトリパスはソースコードの存在している場所を表しているだけであり，警告文本体でも用いられる単語が用いられている場合は逆に結果に悪影響を与えるからである．行番号も

何行目に不具合が存在するかはソースコードによって違うため，ディレクトリと同様に結果に悪影響を与えるからである．

次に生成された警告文本体をトークンに分割する．これは英文で構成されているため，空白文字を区切り文字としてトークンに分割する．

3. 比較実験

本実験では，ソフトウェアメトリクスを用いた予測 (Random Forest)，Fault-prone フィルタリング及び，静的解析を応用した Fault-prone フィルタリングの 3 つの手法でそれぞれ同様のプロジェクトに対し不具合予測を行う．そして，その結果を比較して各手法を評価する．

3.1 実験対象プロジェクト

本実験では次の 7 つのプロジェクトを利用する．

- Apache Ant (注 1)(Ant)(version 1.3, 1.4, 1.5, 1.6, 1.7)：Java プログラムのビルドツール．
- jEdit (注 2)(version 3.2, 4.0, 4.1, 4.2, 4.3)：Java で作成されたテキストエディタ
- Apache Lucene (注 3)(Lucene)(2.0, 2.2, 2.4)：全文検索エンジンの Java ライブラリ．
- Apache POI (注 4)(POI)(version 1.5, 2.0, 2.5, 3.0)：Microsoft Office のファイルの読み書きをする Java ライブラリ．
- Apache Velocity (注 5)(Velocity)(version 1.4, 1.5, 1.6)：テンプレートエンジンの Java ライブラリ．
- Apache Xalan (注 6)(Xalan)(version 2.4, 2.5, 2.6, 2.7)：XML を他の形式へ変換する XSLT プロセッサ．
- Apache Xerces (注 7)(Xerces)(version 1.2, 1.3, 1.4, init)：XML 文書のパースと操作を行うためのソフトウェアパッケージ

これらのプロジェクトには複数のバージョンが存在するが，本実験ではそれらを統合して用いる．各プロジェクトの各ソースコードモジュールの情報を PROMISE[13] のリポジトリから取得する．

PROMISE からは次のメトリクス情報を取得することができる [14]．

(注 1)：http://ant.apache.org/
(注 2)：http://jedit.org/
(注 3)：http://lucene.apache.org/
(注 4)：http://poi.apache.org/
(注 5)：http://velocity.apache.org/
(注 6)：http://xalan.apache.org/
(注 7)：http://xerces.apache.org/

- **Weighted methods per class (WMC)**：各クラスの全メソッドの複雑さの合計
- **Depth of Inheritance Tree (DIT)**：クラスにおける継承度の深さ
- **Number of Children (NOC)**：クラス直下の子クラスの数
- **Coupling between object classes (CBO)**：それぞれのクラスと結合しているクラスの合計
- **Response for a Class (RFC)**：クラスに対して呼び出し可能なメソッドの合計
- **Lack of cohesion in methods(LCOM)**:属性によるメソッドの不同性 (凝集度の欠如)
- **Afferent couplings (Ca)**：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の数
- **Efferent couplings (Ce)**：パッケージ外のクラスに依存するパッケージ内のクラスの数
- **Number of Public Methods (NPM)**：アクセスレベルが public なメソッドの数
- **Lack of cohesion in methods (LCOM3)**：式 (3.1) で表される値 (m:クラス内のメソッドの数，a：クラス内のフィールドの数，$\mu(A)$：フィールド A を参照するメソッドの数)
- **Lines of Code (LOC)**：コードの行数
- **Data Access Metric (DAM)**：クラス内の全フィールド中のアクセスレベルが private と protected のフィールドの割合
- **Measure of Aggregation (MOA)**：ユーザが定義したクラスのフィールドの数
- **Measure of Functional Abstraction (MFA)**：クラス内でアクセス可能なメソッド内の継承されたメソッドの割合
- **Cohesion Among Methods of Class (CAM)**：クラス内のメソッドの凝集
- **Inheritance Coupling (IC)**：クラスが結合する親クラスの数
- **Coupling Between Methods (CBM)**：全ての継承されたメソッドが結合している新規又は再定義されたメソッドの数
- **Average Method Complexity (AMC)**：クラスのコードの数の平均
- **McCabe's cyclomatic complexity (CC)**：循環的複雑度 (本実験では CC の最大値と算術平均値の 2 つを用いる)

- **Faulty**：クラスに含まれるバグの総数 (本研究ではバグの数ではなく，バグの有無を用いるためにバグの有無で二値化する)

$$LCOM3 = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m} \tag{3.1}$$

各プロジェクトのモジュール数を表 3.1 に示す.

3.2 評価指標

本実験では各モジュールを評価した結果，表 3.2 に示すように，True negative，False positive，False negative，True positive の 4 つに分類される．True negative は評価の結果，Fault-prone ではないと予測され，且つ実際に不具合が含まれていないモジュールが分類される．False positive には評価の結果，Fault-prone であると予測されたが，実際には不具合が含まれていなかったモジュールが分類される．そして False negative は，Fault-prone ではないと予測されたが，実際には不具合が含まれていたモジュールが分類される．最後に True positive には，Fault-prone であると予測され，且つ実際に不具合が含まれていたモジュールが分類される.

本実験ではこの 4 つに分類されたモジュールの数を用いて予測の評価指標を計算する．今回は予測の評価指標として正答率（Accuracy），再現率（Recall），適合率（Precision），F_1 値の 4 つを用いる.

正答率（Accuracy）

正答率（Accuracy）とは，全モジュールに対して，実際に不具合を含むモジュールを Fault-prone である，不具合を含まないモジュールを Fault-prone でないと正しく予測できた割合を示す．したがって，表 3.2 の凡例の値を用いると式 (3.2) のように定義される.

$$Accuracy = \frac{TN + FP}{TN + FP + FN + TP} \tag{3.2}$$

この値は用いるデータの偏りなどの影響を受けやすい指標である．そのため，この値のみではなく以下の再現率，適合率，F_1 値といった評価指標を併用する.

表 3.1　各プロジェクトのモジュール数

	不具合あり	不具合なし	合計
Ant	330	1227	1557
jEdit	302	1146	1448
Lucene	437	248	685
POI	706	655	1361
Velocity	366	224	590
Xalan	908	1399	2307
Xerces	336	334	670

表 3.2　実験結果の凡例

		予測	
		Not Fault-prone	Fault-prone
実測	不具合を含まない	True negative(TN)	False positive(FP)
	不具合を含む	False negative(FN)	True positive(TP)

再現率（Recall）

再現率（Recall）とは，実際に不具合を含むモジュールを Fault-prone であると予測できた割合を示す．したがって，表 3.2 の凡例の値を用いると式 (3.3) のように定義される．

$$Recall = \frac{TP}{FN + TP} \tag{3.3}$$

この値が高いということは実際に存在する不具合の多くを予測できることを示すもので，非常に重要な評価指標である．

適合率（Precision）

適合率（Precision）とは，Fault-prone であると予測したモジュールの内，実際に不具合を含んでいたモジュールの割合を示す．したがって，表 3.2 の凡例の値を用いると式 (3.4) のように定義される．

この値が低いということは，実際には不具合でないものを Fault-prone であると誤った予測をする確率が高いことを示し，本来調べる必要のないモジュールを調べる手間が増えることに繋がる．そのため，この値は不具合を発見するために必要なコストを表す指標であるといえる．

$$Precision = \frac{TP}{FP + TP} \tag{3.4}$$

F_1 値

再現率と適合率の調和平均を F_1 として式 (3.5) と定義する．

$$F_1 = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{3.5}$$

再現率と適合率はトレードオフの関係にあるため，両者を総合的に判断するための指標としてこの F_1 値を用いる．

3.3 実験方法

実験の方式としてはプロジェクト横断型予測方式を採用する．その理由としては，現在行われている不具合予測の研究ではそのほとんどが予測したいプロジェクトの

過去の情報を用いて学習に用いていることが挙げられる．そのため，新規のプロジェクトや過去の情報を参照できない状態のプロジェクトでは利用することができなかった．本研究ではそのような過去の履歴を参照できないプロジェクトでも予測が可能なプロジェクト横断型予測方式を用いることとする．

各手法それぞれに共通する実験の流れを以下に示す．

1. 3.1 節で示した 7 つのプロジェクトをそれぞれ学習して辞書を作成する．
2. 作成した 7 つの辞書それぞれを用いて自身以外のプロジェクトをそれぞれ評価する．(例：Ant の辞書を用いた場合は jEdit，Lucene，POI，Velocity，Xalan，Xerces を評価する．)

次に各手法ごとの実験の流れについて説明する．

3.3.1 ソフトウェアメトリクスを用いた予測 (Random Forest)

Random Forest の実行には R 言語 (注 8) の randomForest パッケージを用いる．入力とするデータセットには 3.1 節で示したメトリクス群を用いる．Random Forest はその名が示すようにランダム性の高い評価手法であるため，実験を試行するたびに結果が変わる．よって，100 回の試行の平均を用いることとする．

3.3.2 Fault-prone フィルタリング

2.2 節で述べたアルゴリズムに従って Fault-prone フィルタリングを実行する．

1. SQLite3 を用いて学習用プロジェクトのデータベースから全てのソースコードモジュール及び不具合情報を取得する．
2. 各モジュールをトークンに分割する．
3. 分割したモジュールからトークンを抽出して，不具合情報と共に辞書に学習させる．
4. 学習用プロジェクトと同様の手順で評価用プロジェクトのモジュールをトークンに分割する．
5. 作成した辞書を用いて評価用プロジェクトのトークンを評価する．

(注 8)：http://www.r-project.org/

3.3.3 静的解析を応用した Fault-prone フィルタリング

Fault-prone フィルタリングの実行手順に PMD を挿入する．

1. SQLite3 を用いて学習用プロジェクトのデータベースから全てのソースコードモジュール及び不具合情報を取得する．
2. 得られたソースコードモジュールに対して PMD を実行し，警告文を得る．
3. 得られた警告文を 2.3.4 節に従って必要のない部分を切り取り，警告文本体のみを残した後に空白文字でトークンに分割する．
4. 分割された警告文からトークンを抽出して，不具合情報と共に辞書に学習させる．
5. 学習用と同様の手順で評価用プロジェクトのソースコードモジュールから警告文を取得しトークンに分割する．
6. 作成した辞書を用いて評価用プロジェクトから得られた警告文のトークンを評価する．

3.4 実験結果

3.4.1 全体の傾向

実験結果として，表 3.3 から表 3.14 に各予測手法によって行われた予測の正答率（Accuracy），再現率（Recall），適合率（Precision）及び F_1 値をそれぞれ示す．

始めに，各評価指標について全てのプロジェクトの組み合わせの平均値を比較する．正答率に着目すると，Random Forest が他の 2 手法と比べて高い値を示している．Fault-prone フィルタリング (FP フィルタリング) と PMD を用いた Fault-prone フィルタリング (PMD-FP フィルタリング) は同程度の結果であるが僅差だが PMD-FP フィルタリングが高い値を示している．再現率に着目すると PMD-FP フィルタリングが一番高い値を示し，Random Forest が一番低い値を示している．適合率では，Random Forest が一番高い値を示している．FP フィルタリングと PMD-FP フィルタリングは同程度の結果を示したが，僅かに PMD-FP フィルタリングが高い値を示した．F_1 値に関しては FP フィルタリングと PMD-FP フィルタリングが Random Forest よりも高い値を示し，僅差ではあるが PMD-FP フィルタリングが一番高い値を示した．

3.4.2 正答率

結果を詳細に見る．まず正答率に着目すると，Random Forest では学習に Ant を用いて jEdit を評価した場合，jEdit で学習し Ant を評価した場合に他のプロジェクトの組み合わせと比べて高い結果が得られることが分かった．FP フィルタリングでは，Xerces 以外で学習した場合，jEdit を評価すると低い値が得られた．また，Lucene で学習し Ant を評価した場合も値が低くなっている．PMD-FP フィルタリングでは FP フィルタリングと同様に Xerces 以外で学習して jEdit を評価すると低い値が得られた．また，Xerces で学習すると Lucene を評価した場合にも低い値が得られている．

3.4.3 再現率

次に再現率に着目すると，全ての手法において jEdit を用いて学習した場合に低い値を示している．特に Random Forest では非常に低い値を示している．逆に PMD-FP フィルタリングは他の 2 手法と比べると値の低下が抑えられている．Random Forest では Ant を学習用に用いた場合も非常に低い値が得られた．その中で jEdit を評価した場合は他と比べて高い値を示してはいるが，他のプロジェクトを学習用に用いた場合と比べると大きく下回る値を示している．また，Velocity で学習して Ant を評価した場合も低い値を示している．FP フィルタリングでは，100 % という最高値を記録している箇所が多く見られる．Lucene で学習した場合は全体的に非常に高い値を示している．しかし，Xerces で学習して jEdit を評価した場合は他と比べ著しく低い値が得られた．また，Lucene 以外で学習した場合，Xerces を評価すると低い値を示している．PMD-FP フィルタリングでは，100 % といった最大値を示しているものはないが，全体的に安定して高い値を示している．ただし，Xerces を学習か評価に用いた場合は低い値をが得られた．特に学習で利用した場合は非常に低い値が得られた．

3.4.4 適合率

次に適合率に着目する．この値は再現率とトレードオフの関係にあるので，全体的に見て，再現率が低かった箇所は逆に値が高く，再現率が高かった箇所は値が低い傾向が読み取れる．全ての手法において Ant と jEdit を評価した場合は低い値を

示している．Random Forest では特に Velocity で学習して jEdit を評価した場合は低い値を示している．また，jEdit で学習して Lucene と Velocity を評価した場合に最高値の 100 %を記録している．FP フィルタリングと PMD-FP フィルタリングには全手法に見られる特徴以外には特殊な値は読み取れない．

3.4.5 F_1 値

最後に F_1 値に着目する．Random Forest では，Ant 又は jEdit で学習した場合に低い値が得られた．特に jEdit で学習した場合は著しく低い値を示している．ただし Ant で学習した場合でも，jEdit を評価した場合は値に大きな低下は見られず，むしろ jEdit を評価した中では一番高い値を示している．FP フィルタリングと PMD-FP フィルタリングでは，Ant 又は jEdit を評価した場合に低い値を示している．FP フィルタリングでは特に Xerces で学習し jEdit を評価した場合に非常に低い値が得られた．また，jEdit で学習した場合，Ant を評価した場合を除いて全て低い値を示している．逆に Ant を評価した場合は jEdit で学習したものが一番高い値を示している．PMD-FP フィルタリングでは，Xerces で学習した場合，低い値を示している．

表 3.3　Random Forest の正答率 (Accuracy)

正答率 (Accuracy)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.8106	0.4914	0.5120	0.4405	0.5044	0.6273	0.5644
	jEdit	0.7998	-	0.4426	0.4928	0.4296	0.4680	0.6049	0.5396
	Lucene	0.4393	0.4371	-	0.6407	0.5875	0.5631	0.5543	0.5370
	POI	0.5994	0.5614	0.6250	-	0.5365	0.5486	0.5677	0.5731
	Velocity	0.5301	0.4363	0.5757	0.5073	-	0.4926	0.5707	0.5188
	Xalan	0.5728	0.4496	0.6012	0.6369	0.5349	-	0.5998	0.5659
	Xerces	0.4161	0.3846	0.6209	0.5644	0.5899	0.5137	-	0.5149
	平均	0.5596	0.5133	0.5595	0.5590	0.5198	0.5151	0.5874	0.5448

表 3.4　Random Forest の再現率 (Recall)

再現率 (Recall)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.4225	0.1029	0.0751	0.0447	0.1387	0.1206	0.1507
	jEdit	0.0765	-	0.0048	0.0124	0.0068	0.0317	0.0121	0.0240
	Lucene	0.9134	0.9262	-	0.8275	0.6590	0.7372	0.5509	0.7690
	POI	0.8386	0.8070	0.5171	-	0.3909	0.5112	0.4523	0.5862
	Velocity	0.5833	0.5095	0.7616	0.6724	-	0.4212	0.4584	0.5677
	Xalan	0.8585	0.8890	0.6446	0.8069	0.5167	-	0.4771	0.6988
	Xerces	0.8454	0.8970	0.7425	0.7399	0.7425	0.6734	-	0.7735
	平均	0.6859	0.7419	0.4623	0.5224	0.3934	0.4189	0.3452	0.5100

表 3.5　Random Forest の適合率 (Precision)

適合率 (Precision)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.4503	0.9039	0.7418	0.7035	0.7359	0.6790	0.7024
	jEdit	0.6328	-	1.0000	0.9338	1.0000	0.7652	0.7164	0.8413
	Lucene	0.2582	0.2258	-	0.6106	0.6359	0.5770	0.4510	0.4597
	POI	0.3208	0.2565	0.7348	-	0.6639	0.5999	0.4565	0.5054
	Velocity	0.2392	0.1557	0.5947	0.5152	-	0.5434	0.4606	0.4181
	Xalan	0.3086	0.2248	0.6438	0.6106	0.6128	-	0.4971	0.4829
	Xerces	0.2406	0.2064	0.6391	0.5568	0.6192	0.5427	-	0.4675
	平均	0.3334	0.2533	0.7527	0.6615	0.7059	0.6274	0.5434	0.5539

表 3.6　Random Forest の F_1 値

F_1 値		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.4360	0.1848	0.1363	0.0841	0.2334	0.2049	0.2132
	jEdit	0.1364	-	0.0096	0.0244	0.0135	0.0608	0.0238	0.0448
	Lucene	0.4026	0.3631	-	0.7027	0.6473	0.6474	0.4959	0.5432
	POI	0.4641	0.3893	0.6070	-	0.4921	0.5520	0.4544	0.4931
	Velocity	0.3393	0.2385	0.6679	0.5834	-	0.4745	0.4595	0.4605
	Xalan	0.4540	0.3588	0.6442	0.6951	0.5607	-	0.4869	0.5333
	Xerces	0.3746	0.3356	0.6869	0.6355	0.6753	0.6010	-	0.5515
	平均	0.3618	0.3535	0.4667	0.4629	0.4121	0.4282	0.3542	0.4057

表 3.7　Fault-prone フィルタリングの正答率 (Accuracy)

正答率 (Accuracy)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.2113	0.6380	0.5445	0.6203	0.4586	0.4119	0.4808
	jEdit	0.6821	-	0.5124	0.5026	0.4424	0.6290	0.4448	0.5355
	Lucene	0.2119	0.2086	-	0.5195	0.6119	0.4170	0.5015	0.4117
	POI	0.5260	0.2251	0.6394	-	0.6186	0.5813	0.4284	0.5031
	Velocity	0.5010	0.2189	0.6409	0.6716	-	0.5813	0.3955	0.5015
	Xalan	0.3597	0.2093	0.6380	0.5790	0.6203	-	0.3940	0.4667
	Xerces	0.3012	0.5925	0.5547	0.4578	0.6441	0.4135	-	0.4940
	平均	0.4303	0.2776	0.6039	0.5458	0.5929	0.5134	0.4294	0.4848

表 3.8　Fault-prone フィルタリングの再現率 (Recall)

再現率 (Recall)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	1.0000	1.0000	0.9674	0.9973	0.9780	0.6696	0.9354
	jEdit	0.6667	-	0.3501	0.1926	0.1475	0.3811	0.2619	0.3333
	Lucene	1.0000	1.0000	-	1.0000	0.9836	0.9978	1.0000	0.9969
	POI	0.5121	1.0000	1.0000	-	0.9973	0.4868	0.1667	0.6938
	Velocity	0.7939	1.0000	0.9657	0.7691	-	0.7544	0.3869	0.7783
	Xalan	0.9364	1.0000	1.0000	0.9476	1.0000	-	0.5060	0.8983
	Xerces	0.8212	0.2086	0.5904	0.6402	0.7760	0.8965	-	0.6555
	平均	0.7884	0.8681	0.8177	0.7528	0.8169	0.7491	0.4985	0.7559

表 3.9　Fault-prone フィルタリングの適合率 (Precision)

適合率 (Precision)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.2091	0.6380	0.5336	0.6207	0.4195	0.4429	0.4773
	jEdit	0.3636	-	0.7537	0.5597	0.7606	0.5406	0.4151	0.5655
	Lucene	0.2119	0.2086	-	0.5191	0.6175	0.4028	0.5015	0.4102
	POI	0.2265	0.2121	0.6389	-	0.6197	0.4692	0.3522	0.4198
	Velocity	0.2698	0.2107	0.6462	0.6566	-	0.4797	0.3951	0.4430
	Xalan	0.2405	0.2087	0.6380	0.5552	0.6203	-	0.4146	0.4462
	Xerces	0.2085	0.1522	0.6719	0.4829	0.6893	0.3927	-	0.4329
	平均	0.2535	0.2002	0.6644	0.5512	0.6547	0.4508	0.4202	0.4564

表 3.10　Fault-prone フィルタリングの F_1 値

F_1 値		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.3459	0.7790	0.6878	0.7652	0.5871	0.5332	0.6164
	jEdit	0.4706	-	0.4781	0.2866	0.2471	0.4470	0.3212	0.3751
	Lucene	0.3498	0.3451	-	0.6834	0.7587	0.5740	0.6680	0.5632
	POI	0.3141	0.3499	0.7797	-	0.7644	0.4778	0.2263	0.4854
	Velocity	0.4028	0.3481	0.7743	0.7084	-	0.5865	0.3910	0.5352
	Xalan	0.3827	0.3453	0.7790	0.7002	0.7657	-	0.4558	0.5714
	Xerces	0.3325	0.1760	0.6285	0.5505	0.7301	0.5461	-	0.4940
	平均	0.3754	0.3184	0.7031	0.6028	0.6719	0.5364	0.4326	0.5201

表3.11　PMDを用いたFault-proneフィルタリングの正答率(Accuracy)

正答率 (Accuracy)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.2749	0.6599	0.5922	0.6203	0.4781	0.5239	0.5249
	jEdit	0.4258	-	0.6161	0.6165	0.6051	0.4950	0.4642	0.5371
	Lucene	0.3487	0.2666	-	0.6098	0.6186	0.4530	0.5239	0.4701
	POI	0.3346	0.2604	0.6686	-	0.6119	0.4478	0.5090	0.4720
	Velocity	0.3847	0.3246	0.6190	0.5165	-	0.5020	0.5045	0.4752
	Xalan	0.3610	0.2928	0.6657	0.6062	0.6271	-	0.4672	0.5033
	Xerces	0.5845	0.5504	0.4219	0.4372	0.4441	0.5323	-	0.4951
	平均	0.4066	0.3283	0.6085	0.5631	0.5879	0.4847	0.4988	0.4968

表3.12　PMDを用いたFault-proneフィルタリングの再現率(Recall)

再現率 (Recall)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.9934	0.9382	0.9632	0.9098	0.9615	0.7887	0.9258
	jEdit	0.8848	-	0.7094	0.8669	0.6639	0.6971	0.5536	0.7293
	Lucene	0.9545	0.9007	-	0.9476	0.8825	0.9306	0.7798	0.8993
	POI	0.9545	0.9106	0.9359	-	0.8634	0.9471	0.7798	0.8986
	Velocity	0.9182	0.9272	0.8879	0.7805	-	0.9361	0.7054	0.8592
	Xalan	0.9697	0.9768	0.9359	0.9688	0.9016	-	0.6756	0.9048
	Xerces	0.3818	0.4768	0.3364	0.1530	0.3962	0.6597	-	0.4006
	平均	0.8439	0.8642	0.7906	0.7800	0.7696	0.8554	0.7138	0.8025

表 3.13　PMD を用いた Fault-prone フィルタリングの適合率 (Precision)

適合率 (Precision)		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.2226	0.6656	0.5624	0.6355	0.4275	0.5166	0.5050
	jEdit	0.2544	-	0.6951	0.5885	0.6884	0.4156	0.4709	0.5188
	Lucene	0.2397	0.2086	-	0.5752	0.6396	0.4134	0.5168	0.4322
	POI	0.2358	0.2085	0.6727	-	0.6384	0.4123	0.5068	0.4457
	Velocity	0.2455	0.2265	0.6467	0.5228	-	0.4379	0.5043	0.4306
	Xalan	0.2452	0.2248	0.6705	0.5710	0.6420	-	0.4779	0.4719
	Xerces	0.2214	0.2261	0.5810	0.3913	0.5754	0.4375	-	0.4055
	平均	0.2403	0.2195	0.6553	0.5352	0.6365	0.4240	0.4989	0.4585

表 3.14　PMD を用いた Fault-prone フィルタリングの F_1 値

F_1 値		評価							
		Ant	jEdit	Lucene	POI	Velocity	Xalan	Xerces	平均
学習	Ant	-	0.3636	0.7787	0.7102	0.7483	0.5919	0.6243	0.6362
	jEdit	0.3951	-	0.7022	0.7010	0.6759	0.5208	0.5089	0.5840
	Lucene	0.3832	0.3387	-	0.7159	0.7417	0.5725	0.6216	0.5623
	POI	0.3782	0.3393	0.7828	-	0.7340	0.5745	0.6143	0.5705
	Velocity	0.3875	0.3641	0.7483	0.6261	-	0.5967	0.5881	0.5518
	Xalan	0.3914	0.3656	0.7813	0.7185	0.7500	-	0.5598	0.5944
	Xerces	0.2803	0.3067	0.4261	0.2200	0.4693	0.5261	-	0.3714
	平均	0.3693	0.3463	0.7032	0.6153	0.6865	0.5637	0.5862	0.5529

4. 考察

4.1 予測手法の比較

ここでは実験において得られた評価値について，予測手法間での平均を比べる．表 3.6, 表 3.10, および表 3.14 から，最も F_1 値が高かった手法が PMD を用いた Fault-prone フィルタリングであったことがわかる．

また，Fault-prone フィルタリングとその派生である PMD を用いた Fault-prone フィルタリングでは，再現率が非常に高くなる現象が確認される．これは，手法が提案された当初より文献 [7] でも観測された事象であり，適用するプロジェクトが変わっても概ね傾向が変化しないことを裏付けている．

今回の実験は，1 つのプロジェクト全体を学習させ，まったく別のプロジェクトに対して予測実験を行っている．学習対象と予測対象には論理的なつながりが薄く，特に文書の類似性を評価して予測する Fault-prone フィルタリングには不利な実験であると思われたが，予想に反して Fault-prone フィルタリングの予測結果は悪くなかった．この結果から，Fault-prone フィルタリング法が適用環境の変化に対してロバストであることが確認できる．

4.2 プロジェクト間の比較

実験結果でも述べているが，jEdit を学習対象として利用し，手法を Random Forest とした場合に予測精度が大きく悪化する現象が観察されている．また，jEdit を予測対象として選んだ場合でも再現率と適合率の比が大きくなる現象が確認される．このため，jEdit において収集されたメトリクスの値が他のプロジェクトから得られたものに比べ，大きく傾向を異にするものだと推察される．各プロジェクトから得られたデータの記述統計を比較しても，特別に特異な点を確認することもできなかったため，jEdit の特性分析は今後の課題として残る．

5. 結言

本論文では複数のソフトウェア不具合予測手法での予測結果の違いについての考察を行う．実験にあたり，我々は共有データリポジトリで公開されたデータを元に，新たなデータをこれに追加することで複数の手法を同一の条件で実行することに成功した．比較実験で利用したのは 3 つの不具合予測手法である．

- Random Forest 手法
- Fault-prone フィルタリング手法
- PMD を用いた Fault-prone フィルタリング手法

これらの手法は入力として与えるのはソフトウェアのモジュールであるものの，利用するメトリクスは異なる．本実験では，複数のプロジェクトを用いて，プロジェクト横断的な予測を実施した．これにより，手法とプロジェクトの違いが予測結果にどのような影響を及ぼすかを評価した．

実験の結果，PMD を用いた Fault-prone フィルタリング手法が平均的に最も高い F_1 値を達成することが確認された．また，プロジェクトの特性に依るものとして，一部のプロジェクトを学習・予測に利用すると，特に Random Forest 法において予測精度が悪化する現象を確認した．

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，本論文の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野修准教授に厚く御礼申し上げます．本論文執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻梁軍偉君，出原真人君，川本公章君，椋代凛君，大西哲朗君，学生生活を通じて筆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] L. Breiman, “Random forests,” Machine learning, vol.45, no.1, pp.5–32, 2001.

[2] L. Guo, Y. Ma, B. Cukic, and H. Singh, “Robust prediction of fault-proneness by random forests,” Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, pp.417–428, nov. 2004.

[3] 水野　修，菊野　亨，“Fault-prone 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，” SEC journal，vol.4，no.1，pp.6–15，Feb. 2008.

[4] M. Nakai and O. Mizuno, “Fault-prone module prediction by filtering warning messages of static code analyzer,” Proc. of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/MENSURA2011), Fast abstracts, pp.18–21, Nov. 2011. Nara, Japan.

[5] O. Mizuno and M. Nakai, “Can faulty modules be predicted by warning messages of static code analyzer?,” Advances in Software Engineering, vol.2012, no.924923, p.8 pages, May 2012.

[6] O. Mizuno and T. Kikuno, “Training on errors experiment to detect fault-prone software modules by spam filter,” The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007), pp.405–414, Sept. 2007. Dubrovnik, Croatia.

[7] O. Mizuno and T. Kikuno, “Prediction of fault-prone software modules using a generic text discriminator,” IEICE Trans. on Information and Systems, vol.E91-D, no.4, pp.888–896, April 2008.

[8] H. Hata, O. Mizuno, and T. Kikuno, “Fault-prone module detection using large-scale text features based on spam filtering,” Empirical Software Engineering, vol.15, no.2, pp.147–165, April 2010.

[9] 平田幸直，水野　修，“テキスト分類に基づく fault-prone モジュール検出法におけるコメント行の影響の分析,” 情報処理学会研究報告 ソフトウェア工学 (SE)，vol.2010-SE-170，no.10，pp.1–8，Nov. 2010．大阪大学．

[10] P. Graham “Hackers and painters: Big ideas from the computer age,”, chapter 8, pp.121–129, O’Reilly Media, 2004.

[11] T. Copeland, PMD Applied, Centennial Books, Arexandria, VA, 2005.

[12] O. Mizuno and H. Hata, “A hybrid fault-proneness detection approach using text filtering and static code analysis,” International Journal of Advancements in Computing Technology, vol.2, no.5, pp.1–12, Dec. 2010.

[13] G. Boetticher, T. Menzies, and T. Ostrand “Promise repository of empirical software engineering data repository,”, http://promisedata.org/, West Virginia University, Department of Computer Science, 2007.

[14] M. Jureczko and L. Madeyski, “Towards identifying software project clusters with regard to defect prediction,” Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pp.9:1–9:10, PROMISE ’10, ACM, New York, NY, USA, 2010.