情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-SE-170 No.10
2010/11/12

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

平田幸直[†1]　水野修[†1]

テキスト分類に基づくfault-proneモジュール検出法であるfault-proneフィルタリングではこれまでコード行とコメント行の区別がなされていなかった．しかし，モジュール中のコード行はコンピュータに望む処理が記述されたものであり，コメント行はコードの説明や利用法などの記述が主であるなど，コード行とコメント行は異なる役割を果たすものである．そこで，本研究ではコード行とコメント行を区別して扱いEclipseのデータセットに対してfault-proneフィルタリングを行った．その結果，コメント行を利用した方がコード行を利用した検出よりも高い再現率と適合率を示した．

Analysis of Influence of Comment Lines in Fault-prone Module Detection Based on Text Categorization

Yukinao Hirata [†1] and Osamu Mizuno [†1]

Comment lines in the software source code includes descriptions of codes, usage of codes, copyrights, unused codes, comments, and so on. From the viewpoint of fault-prone module prediction, comment lines may have useful information for faulty modules as well as in code lines. In fault-proneness filtering approach, which we have been proposed based on text filtering technique, comment lines and code lines in a source code modules are regarded as text without any distinction. For better prediction results of fault-prone modules, the effects of the comment lines on prediction should be investigated. This study conducts experiments using Eclipse data sets to reveal the effects of comment lines on the fault-proneness filtering. The result of experiments was somehow unexpected: prediction using comment lines only shows better recall and precision.

†1 京都工芸繊維大学 大学院工芸科学研究科
Graduate School of Science and Technology, Kyoto Institute of Technology

1. はじめに

ソフトウェア開発において，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，バグを効率よく除去できるため，開発コストを削減することができる．そのため，FPモジュールを予測するための研究が数多く行われてきた[1),2)]．

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複雑さに関連しているとの仮定を置き，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，その不具合を推定するモデルを構築していた．

これに対し，我々は不具合がソフトウェアモジュール中の語や文脈に関連すると考え，モジュールをテキストとして扱いスパムフィルタによって不具合の有無を判定する「fault-proneフィルタリング法」を提案した[3)–5)]．

これまでのfault-proneフィルタリングの研究では，モジュールを1つのテキストとして扱い，モジュール中のコメント行とコード行を区別することなく扱ってきた．しかし，ソースコード中におけるコメント行とコード行ではその役割が異なるものであるため，これらを区別してfault-proneフィルタリングを行うことで，コメント行やコード行に特化した結果が得られるのではないかと期待する．

そこで，本研究ではコメント行とコード行を区別して実験を行い，その結果を比較することによって，コメント行のみを用いたfault-proneフィルタリング法の有効性などを評価する．本研究ではコメント行とコード行での予測精度の違いに重点を置くために，従来のものとは別に新たにfault-proneフィルタを試作した．

具体的には，まず，コメント行とコード行を区別した辞書を学習する．あるソフトウェアにおける不具合モジュールの情報を含む公開データセット[2)]を利用して，対象としたソースコードモジュールを，不具合有りと無しとを区別して取得した．次に，不具合有りと分類されているソースコードモジュールからコメント行とコード行を別々に抽出する．そして，コメント行とコード行のそれぞれについてテキストのトークンを抜き出して，不具合有りのコメント行辞書，不具合有りのコード行辞書のそれぞれに登録する．不具合無しモジュールからも同様にトークンを抽出し，不具合無しの辞書に登録する．こうしてコメント行，コード行のそれぞれに対して作成した辞書を得る．その後，学習した辞書を用いて，新しいモジュールの予測実験を行う．コメント行とコード行を抽出したものをトークン化し，試作したfault-proneフィルタによって不具合の有無を予測する．最後に，予測の精度をコメント行とコード行を用いた実験の間で比較する．

本論文の構成を紹介する．まず，第 2 節では fault-prone フィルタリングの概念について説明する．第 3 節では今回の実験のために試作した fault-prone フィルタについて説明する．第 4 節ではオープンソースのプロジェクトへの適用実験について述べる．第 5 節では実験の結果に対する考察を述べる．最後に，第 6 節では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

Fault-prone フィルタリング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考え方を導入したものである．

現在使われている一般的なスパムフィルタでは，その基礎にベイズの定理を用いたものが主流となっている．こうしたスパムフィルタでは，受信したメールを学習していくことによって，新たに受信したメールを本人に有用なメール（ham）と迷惑なメール（spam）に分類することができる．

ここでは，スパムフィルタの動作について説明する．基本的なスパムフィルタはメールの特徴を学習するステップと学習した結果を利用してメールを分類するステップからなる．学習ステップでは，学習用に ham メールか spam メールかがわかっているメール群を用意する．そして，各メールから単語（トークン）を抽出して辞書に記述する．この際，ham メールから抽出したトークンは ham メール用の辞書に，spam メールから抽出したものは，spam 用の辞書に記述する．これにより，2 つの辞書が作成される．以上が学習ステップである．分類ステップでは学習ステップにおいて作成した 2 つの辞書を利用して，新しく受信したメールを ham と spam のどちらかに分類する．

このようなスパムフィルタの動作は，ham と spam にはそれぞれ異なる特徴がある，つまり，文中に含まれる語や文の傾向が異なるという仮定に基づいている．

我々は，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て，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，それを「fault-prone フィルタリング法」と名付けた．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 メールと見なし，スパムフィルタの処理と同様に辞書を学習する．そして，学習した辞書を用いて新しいモジュールが不具合有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判定を行う．

Fault-prone フィルタの有効性については，これまでにも実験を行い，議論を進めてきている[3)–5)]．本研究では，その過程で明らかになってきた 1 つの課題「コメント行の扱い」に対する仮説の実証実験を行う．

2.1 Fault-prone モジュール検出におけるコメント行の利用

本研究では，モジュール中のコメントに該当する部分を「コメント行」と呼び，その他を「コード行」と呼ぶ．コード行とはコンピュータで実現したい動作が記述されたものであり，プログラミング言語の構文に従って記述されている部分である．これに対し，コメント行は，主に複雑なソースコードの動作の説明やメソッドの利用法の説明などが言語の構文にとらわれず記述されるものである．

Fault-prone フィルタリングは，モジュール中から抽出したトークンに基づいてその分類を行うが，過去の研究 4) においてはコード行とコメント行の区別は行わずにトークンを抽出し分類が行われてきた．また，研究 5) においては，コメントを削除した状態での実験が行われている．

しかしながら，コード行とコメント行ではその持つべき役割が異なっているため，これらを区別して扱うことで fault-prone モジュールの予測精度に対して及ぼす影響を評価することが重要だと考える．

たとえば，コメントはコードが複雑であると開発者が判断したために記述されることが多い．このような，人間の主観的判断は客観的な複雑性メトリクスなどよりも場合によっては不具合を検出する指標としての信頼性が高い可能性がある．このため，コメント行を効率よく利用することは不具合の検出精度に良い影響を与えることがあると考えられる．そこで，本研究ではコード行とコメント行を区別して扱いながら fault-prone モジュールの検出を行うことによって，コメント行が fault-prone フィルタの分類精度に対して与える影響を調査する．

2.2 コメント行の定義

本節では，Java 言語におけるコメントとその扱いについて述べる．Java 言語には次に挙げる 3 種類のコメントが存在する．

(1) “//”で始まるコメント
(2) “/*”で始まり“*/”によって閉じられるコメント
(3) “/**”で始まり“*/”によって閉じられるコメント

ここで，3 つ目のコメントはドキュメンテーションコメントと呼ばれ，その他のコメントとは利用目的が異なるが，本研究では各コメントを区別しないものとする．また，コメントの内容がコードに説明を加えるものであるのか，コードをコメントアウトしたものであるか

等の区別も行わないものとする．

3. Fault-prone フィルタの実装

本節では，実験で利用した fault-prone フィルタについて説明する．この fault-prone フィルタはベイズの定理に基づいた単純なものである．学習ステップと分類ステップのアルゴリズムについては，文献 6) 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になっている．

3.1 トークン分割

Fault-prone フィルタの入力となるソースコードモジュールは，全てトークンと呼ばれる単位に分解される．ここでは，コード行とコメント行におけるトークンの分割方法について述べる．

まず，コード行の分割について説明する．コード行の分割ではコードを以下の 4 種類のトークンに分割する．

(1) アルファベット，数字，ドットからなる文字列
(2) Java 言語の演算子
(3) シングルクオートに囲まれた文字列
(4) ダブルクオートに囲まれた文字列

以上のようなトークンに分割する際に，空白や括弧，セミコロンなどの上記トークンの構成要素ではない文字についてはトークンを区切るためのみに利用されるためトークンとしては出力されない．

次に，コメント行の分割について説明する．コメント行の分割もコード行の分割と基本的に同じである．ただし，コメントはコードのようにプログラミング言語の構文に従ったものではないため，ダブルクオート等が正しく閉じられていない場合や，アポストロフィなのかシングルクオートに囲まれた文字列であるのかを判定することが困難な場合などがある．そのため，シングルクオートとダブルクオートについては区切り文字として扱う．よって，文字列「`output("Hello, world!");`」が与えられたとき，コード行としては `output`，`"hello world!"`の 2 つに分割され，コメント行としては `output`，`hello`，`world` の 3 つに分割されるものとする．

3.2 学習ステップ

Fault-prone フィルタの学習ステップについて示す．このステップでは，後の分類ステップで必要な辞書を作成する．

Step 1 まず，不具合有り (ft と表記する) モジュールと不具合無し (nf と表記する) モジュールからトークンを抽出する．そして，ft, nf ごとにトークンの出現数を数える．これにより，次の 2 つのハッシュテーブルを得る．

- $faulty(t)$： 全 FP モジュールにおけるトークン t の出現数．
- $nonfaulty(t)$： 全 NFP モジュールにおけるトークン t の出現数．

Step 2 トークン t が含まれるモジュールが ft モジュールである確率 $P_{ft|t}$ を得るハッシュテーブルを作成する．ここで，N_{ft} は ft モジュールの数，N_{nf} は nf モジュールの数である．確率 $p_{ft|t}$ は式 (1) で与えられる．

$$
\begin{aligned}
r_{nf} &= \min\left(1, \frac{2 \times nonfaulty(t)}{N_{nf}}\right) \\
r_{ft} &= \min\left(1, \frac{faulty(t)}{N_{ft}}\right) \\
P_{ft|t} &= \max\left(0.01, \min\left(0.99, \frac{r_{ft}}{r_{nf} + r_{ft}}\right)\right)
\end{aligned} \tag{1}
$$

3.3 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 ft モジュールであるか (すなわち，fault-prone か)，そうでないのかの判定を行う．

Step 1 分類を行う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，式 (2) の値が大きいほど特徴的であるとする．ただし，今までに，一度も出現していないトークンであった場合は $P_{ft|t} = 0.4$ とする．

$$
abs(0.5 - P_{ft|t}) \tag{2}
$$

なお，今回の実験では n 個の特徴的なトークンを利用せず，全てのトークンを利用したため，上の Step 1 は実質的に省略されている．

Step 2 式 (3) によってモジュールが ft モジュールである確率が計算される．あるモジュールが fault-prone モジュールであるか否かは，$P_{ft} \geq 0.9$ であるか否かで判定する．ここで使っている 0.9 は変更可能な閾値である．

$$
P_{ft} = \frac{\prod_{i=1}^{n} P_{ft|t_i}}{\prod_{i=1}^{n} P_{ft|t_i} + \prod_{i=1}^{n}(1 - P_{ft|t_i})} \tag{3}
$$

4. 適 用 実 験

この実験では，fault-prone フィルタリングにおいて，モジュール中のコメント行がモジュー

Vol.2010-SE-170 No.10
2010/11/12

表 1 Eclipse の各バージョンにおけるモジュールの詳細
Table 1 Specification of modules in the Eclipse

Project	Eclipse		
Version	2.0	2.1	3.0
Number of modules	6,729	7,888	10,593
Number of faulty modules	975 (14.5%)	854 (10.8%)	1,568 (14.8%)
Number of non-faulty modules	5,754	7,034	9,025

ルの分類に与える影響を調べることを目的とする．そこで，コード行によるモジュールの分類実験とコメント行による分類実験の 2 通りを行う．

4.1 実験対象

実験の対象としたのは，オープンソースプロジェクトである Eclipse の異なる 3 つのバージョンから得られたデータである．実験を行うためには，各バージョンに含まれるソフトウェアモジュールのソースコードだけではなく，各ソフトウェアモジュールについての不具合情報が必要である．この情報として，Zimmermann ら[7] によって作成された *promise-2.0a* と呼ばれるデータセットを用いる．

データセット *promise-2.0a* にはリリース前後 6ヶ月に発見された不具合の情報と 31 種類のメトリクスを測定したデータが記録されている．これらの情報のうち，本実験ではファイルごとに測定されたリリース後の 6ヶ月間に発見された不具合の情報（*post*）を用いる．

表 1 にデータセットから得られた Eclipse の各バージョンにおけるモジュールの詳細を示す．表中の括弧内の数値は全モジュールに対する ft モジュールの割合である．

4.2 実験方法

この節では，コード行のみを用いた分類実験とコメント行のみを用いた分類実験との 2 つの実験の方法について説明を行う．

まず，不具合有り (ft) モジュールと不具合無し (nf) モジュールの定義について説明する．この実験においてモジュールとは 1 つのソースファイルを指すものとする．そして，各モジュールが ft モジュールであるか nf モジュールであるかはデータセット *promise-2.0a* におけるリリース後の不具合（*post*）の数値によって定める．もし，*post* が 0 より大きい場合，そのモジュールを ft モジュールと決定し，そうでなければ nf モジュールと決定する．ここで決定された値を今回の実験における真値とする．

ここで，モジュール M からコード行のみを取り出したものを M_{code} とする．そして，M からコメント行のみを取り出したものを M_{com} とする．コード行による分類の実験においては M_{code} のみを用い，コメント行による分類の実験においては M_{com} のみを用いる．また，上で述べた方法で，それぞれのモジュールに対して不具合有り (ft) であるか，不具合無し (nf) であるかの決定を行う．以上の概略を図 1 に示す．

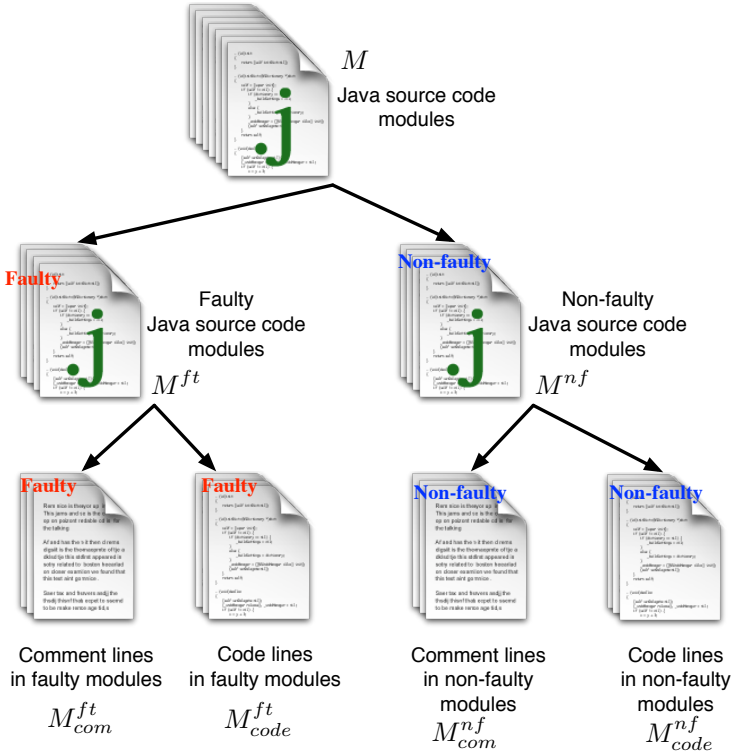

図 1 モジュール取得の流れ
Fig. 1 Getting modules

次に，学習と分類を行う実験をコード行とコメント行に対して実施する．コード行による分類の実験は次のような流れになる．

学習ステップ 図 2 に学習ステップの概略を示す．

まず，学習に用いるバージョン（*train*）を決定する．次に，全ての不具合有り (ft) モ

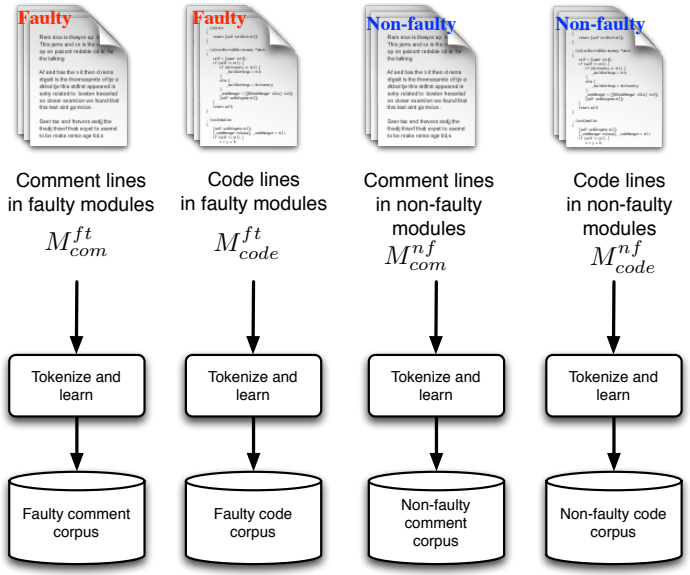

図 2 学習ステップの概略
Fig. 2 Learning procedure

ジュールから M_{code}^{ft} を取り出し，トークンを抽出して ft 辞書に学習させる．さらに，残りの不具合無し (nf) モジュールからも同様に M_{code}^{ft} を取り出し，nf 辞書に学習させる．

分類ステップ 図 3 に分類ステップの概略を示す．

まず，分類に用いるためのバージョン（$test$）を決定する．これには，学習ステップで用いたバージョンとは異なるものを選択する．次に，分類するモジュールから M_{code} を取り出してトークンを抽出し，学習ステップで作成した辞書を利用して分類を行う．これを分類対象のバージョンの全てのモジュールに対して行う．3.3 節において述べたように，分類をするためには利用するトークンの数 n を決定する必要があるが，この実験では分類対象の M_{code} 内の全てのトークンを用いることにしている．

コメント行による分類の実験は，モジュールからコード行を取り出した M_{code} を利用するのではなく，コメント行を取り出した M_{com} を利用することを除いてコード行による分類実験と同じである．

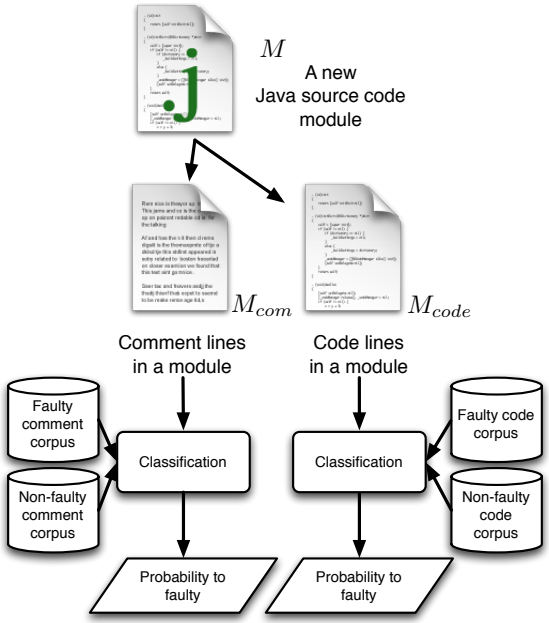

図 3 分類ステップの概略
Fig. 3 Classification procedure

この実験では，Eclipse の 3 つのバージョンを用いるが，学習には分類対象のバージョンより古いバージョンのものを用いる．これは，実用的な面を考えると古いバージョンで新しいバージョンの予測ができることが望ましいからである．従って，バージョン 2.0 で学習を行う場合は，バージョン 2.1 と 3.0 の分類を行い，バージョン 2.1 で学習を行う場合はバージョン 3.0 の分類を行う．3 通りの学習・分類の組み合わせに対して，コメント行・コード行による予測を実施するため，全部で 6 通りの実験結果を得ることになる．これらの 6 通りの実験を 3 項組，(利用する情報, 学習バージョン, 分類バージョン) で表すことにする．

4.3 評価指標

モジュールの予測結果は表 2 のようにまとめられる．得られた結果の評価指標として、正答率（Accuracy），再現率（Recall），適合率（Precision），F_1 値を用いる．

Vol.2010-SE-170 No.10
2010/11/12

4.3.1 正 答 率

正答率（Accuracy）は，全モジュールのうち，実測が不具合であるファイルを fault-prone (FP)，不具合ではないモジュールを fault-prone ではない (NFP) と正しく予測できたモジュールの割合を示す．よって，凡例の表 2 の値を用いて式 (4) のように定義される．

$$Accuracy = \frac{N_1 + N_4}{N_1 + N_2 + N_3 + N_4} \tag{4}$$

正答率はデータの偏りなどに影響されやすい指標であり，この値のみをもって予測精度を評価することは難しい．そのため，下に示す再現率，適合率，F_1 値なども併用した評価をするのが一般的である．

4.3.2 再 現 率

再現率（Recall）は実測が不具合である全てのモジュールのうち，正しく FP と予測できた割合を示す．よって，再現率は式 (5) のように定義される．

$$Recall = \frac{N_4}{N_3 + N_4} \tag{5}$$

直感的には再現率は実不具合を予測できる網羅率を表している．そのため，fault-prone モジュールの検出という場面では非常に重要な指標となる．

4.3.3 適 合 率

適合率（Precision）は FP と予測したモジュールのうち，実測が不具合であったモジュールの割合を示す．適合率は式 (6) のように定義される．

$$Precision = \frac{N_4}{N_2 + N_4} \tag{6}$$

直感的には，適合率は実不具合を発見するために必要なコストを表していると考えられる．すなわち，この値が低いと実際は不具合ではないものまで調べる手間が増えるためである．

4.3.4 F_1 値

再現率と適合率はトレードオフの関係にある．つまり，どちらか一方を下げる代わりに，他方を上げることができる．そこで，再現率と適合率の調和平均である F_1 を式 (7) により定義する．

$$F_1 = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{7}$$

F_1 値は再現率と適合率を総合的に判断するための指標である．一般にこの値が高いほど予測精度が高いと判断できるが，やはりデータの偏りなどの影響は受ける．

表 2 実験結果の凡例
Table 2 Classification result matrix

		予測	
		NFP	FP
実測	nf	N_1	N_2
	ft	N_3	N_4

4.4 実 験 結 果

表 3 から表 8 に 6 つの実験におけるモジュールの分類結果を示す．また，表 9 に分類結果をまとめて評価指標を算出したものを示す．

表 9 に示したコメント行とコード行による予測の結果について比較すると，Recall については，コメント行による結果とコード行による結果でどちらが必ず高いといった特徴は見られない．これに対し，Precision では，コメント行による結果がコード行による結果よりも高い．また，Recall と Precision を組み合わせた値である F_1 値についてもコメント行の

表 3 予測結果: (コメント行, 2.0, 2.1)
Table 3 Prediction result: (comment, 2.0, 2.1)

		予測	
		NFP	FP
実測	nf	5,952	1,082
	ft	508	346

表 4 予測結果: (コード行, 2.0, 2.1)
Table 4 Prediction result: (code, 2.0, 2.1)

		予測	
		NFP	FP
実測	nf	5,711	1,323
	ft	492	362

表 5 予測結果: (コメント行, 2.0, 3.0)
Table 5 Prediction result: (comment, 2.0, 3.0)

		予測	
		NFP	FP
実測	nf	7,625	1,400
	ft	835	733

表 6 予測結果: (コード行, 2.0, 3.0)
Table 6 Prediction result: (code, 2.0, 3.0)

		予測	
		NFP	FP
実測	nf	7,477	1,548
	ft	934	634

表 7 予測結果: (コメント行, 2.1, 3.0)
Table 7 Prediction result: (comment, 2.1, 3.0)

		予測	
		NFP	FP
実測	nf	7,563	1,462
	ft	986	582

表 8 予測結果: (コード行, 2.1, 3.0)
Table 8 Prediction result: (code, 2.1, 3.0)

		予測	
		NFP	FP
実測	nf	7,883	1,142
	ft	1,133	435

表 9 実験結果のまとめ（閾値 = 0.9）
Table 9 Summary of experimental results (Threshold = 0.9)

学習 (Training)	分類 (Test)	利用情報 (Target)	正答率 (Accuracy)	再現率 (Recall)	適合率 (Precision)	F_1
2.0	2.1	コメント行	**0.798**	0.405	**0.242**	**0.303**
		コード行	0.770	**0.424**	0.215	0.285
2.0	3.0	コメント行	**0.789**	**0.467**	**0.344**	**0.396**
		コード行	0.766	0.404	0.291	0.338
2.1	3.0	コメント行	0.769	**0.371**	**0.285**	**0.322**
		コード行	**0.785**	0.277	0.276	0.277

みを利用した実験の場合の方が高い値を示している．

3.3 節で示したように，fault-prone モジュールと判定するための閾値は 0.9 で実験を行ったが，この値は現実的には高すぎるとも考えられる．そこで，閾値を 0.5 とした時の実験結果を表 10 に示す．FP モジュールと判定する基準を下げたため，閾値 0.5 の結果は FP と判定した数が増え，Recall が増加している．逆に Precision に関しては低下している．しかし，予測の全体的な傾向に特に大きな差は見られない．

表 10 実験結果のまとめ（閾値 = 0.5）
Table 10 Summary of experimental results (Threshold = 0.5)

学習 (Training)	分類 (Test)	利用情報 (Target)	正答率 (Accuracy)	再現率 (Recall)	適合率 (Precision)	F_1
2.0	2.1	コメント行	**0.789**	0.419	**0.234**	**0.300**
		コード行	0.755	**0.438**	0.204	0.279
2.0	3.0	コメント行	**0.776**	**0.492**	**0.328**	**0.394**
		コード行	0.747	0.415	0.270	0.327
2.1	3.0	コメント行	0.738	**0.416**	**0.260**	**0.320**
		コード行	**0.768**	0.292	0.254	0.272

5. 考　　察

表 9 と表 10 の結果よりこの実験結果の特筆すべき点は，コメント行のみを用いた予測実験の結果と，コード行のみを用いた実験の結果がほとんど同じ値を示していることにある．さらに，F_1 値を比較すると，コメント行を利用した予測結果の方がコード行を利用した結果よりもわずかではあるが高くなっている．このことから，fault-prone フィルタリングにおいてコメント行を利用することは予測精度をやや向上させていることが分かる．

コード行による結果とコメント行による結果に差が余り無かったのは，どちらの場合についても学習データが十分でなかったためであると考えられる．これは，スパムフィルタが正しく動作するには十分な量の学習データが必要であり，スパムフィルタを利用する fault-prone フィルタリングにおいても同様だからである．従来の手法では，過去数年間の履歴といった巨大なデータを学習させていたため，学習量は十分であった．しかし，今回の実験では，学習には単一のバージョンのみを用いたため，学習量という点では過去の研究と比べても非常に少ない．このため，十分な量の学習データを利用した場合には，コード行とコメント行のどちらによる分類がより良い結果を出すのかを調査する必要がある．

6. ま と め

本研究では FP フィルタリングにおいてコメント行のみを用いて予測する実験とコード行のみを用いて予測する実験を行った．その結果，コメント行のみを用いた実験ではコード行のみを用いた実験よりもわずかではあるが，F_1 値が高くなることがわかった．

今後の課題として，他のプロジェクトに適用した場合の結果の調査やどのようなトークンが不具合予測に与える影響が大きいかなどを調査する必要がある．

また，今回作成した fault-prone フィルタは非常に原始的な実装になっており，今後はソフトウェアモジュールの特性を考慮した手法へと発展させていく必要があると考えられる．一例として，今回の分析での結果を基に，コメント行やコード行，また，構成管理システムのログなどを区別した辞書により不具合を予測し，その予測結果を突き合わせて最終的な予測結果を導き出す手法の開発などが可能だと考えている．

参 考 文 献

1) Khoshgoftaar, T.M. and Seliya, N.: Comparative Assessment of Software Quality Classification Techniques: An Empirical Study, *Empirical Software Engineering*, Vol.9, pp.229–257 (2004).
2) Zimmermann, T., Premrai, R. and Zeller, A.: Predicting defects for Eclipse, *Proc. of 3rd International Workshop on Predictor models in Software Engineering* (2007).
3) Mizuno, O. and Kikuno, T.: 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 foundations of software engineering*, pp.405–414 (2007).
4) Mizuno, O. and Kikuno, T.: Prediction of Fault-Prone Software Modules Using a

Vol.2010-SE-170 No.10
2010/11/12

Generic Text Discriminator, *IEICE Trans.on Information and Systems*, Vol.E91-D, No.4, pp.888–896 (2008).

5) Hata, H., Mizuno, O. and Kikuno, T.: Fault-Prone Module Detection Using Large-Scale Text Features Based on Spam Filtering, *Empirical Software Engineering*, Vol.15, No.2, pp.147–165 (2010).

6) Graham, P.: *Hackers and Painters: Big Ideas from the Computer Age*, chapter8, pp.121–129, O'Reilly Media (2004).

7) Boetticher, G., Menzies, T. and Ostrand, T.: *PROMISE Repository of empirical software engineering data repository*, http://promisedata.org/, West Virginia University, Department of Computer Science (2007).