情報処理学会論文誌 Vol.61 No.4 895–907 (Apr. 2020)

コーディング規約違反メトリクスに基づき ソフトウェア変更に対して不具合混入を予測する手法

名倉 正剛[1,a)] 田口 健介[2,†1] 高田 眞吾[2,b)]

受付日 2019年7月31日, 採録日 2020年1月16日

概要：ソフトウェアの変更に対する既存の不具合予測手法には，不具合を混入した可能性のあるコミットの予測の際に，教師データの利用有無にかかわらずプロジェクト固有の特徴を利用するものがある．これらの利用には対象プロジェクト自体のプロジェクトデータの分析が必要であり，変更が十分に実施された後でないと適用が難しい．プロジェクトに依存せず不具合混入に関連するメトリクスであれば，予測対象プロジェクトと異なるプロジェクトデータの学習により，対象プロジェクト自体のプロジェクトデータの分析の必要なく不具合予測を実施できる．我々はそのようなメトリクスとして，コーディングスタイルに対する規約違反に基づくメトリクスに着目した．本研究では，予備実験によりこのメトリクスの利用でプロジェクトによらず不具合混入を予測できることを明らかにしたうえで，このメトリクスを利用した不具合予測手法を提案する．提案手法は，OSS プロジェクトからこのメトリクスにより算出した値と不具合混入との関連を教師データとして利用し，教師あり学習で予測する．評価実験により，予測対象プロジェクトと異なるプロジェクトデータの学習によって，先行研究に近い性能で不具合予測を実施できることを確認した．

キーワード：不具合予測，コーディング規約違反，メトリクス変化，ソフトウェア変更

A Defect Prediction Method for Software Changes Based on Coding Style Violation Metrics

Masataka Nagura[1,a)] Kensuke Taguchi[2,†1] Shingo Takada[2,b)]

Received: July 31, 2019, Accepted: January 16, 2020

Abstract: Existing some defect prediction methods for software changes use project dependent characteristics to predict defect prone commits, whether they use supervised learning or not. This means it is necessary to analyze data, i.e., changes, from the target project, which makes it difficult when there are not enough changes committed within the target project. If we can use defect-related metrics which are independent of the target project, we may be able to predict defects based on data from projects other than those of the target project. In this paper, we focus on coding style violation metrics. We first clarify if we can use these metrics to predict the defects independently of the target project. We then propose a defect prediction method using these metrics. Our proposed method predicts defects by supervised learning using metrics values and defects. The evaluation showed that defect prediction accuracy of our method is close to existing methods.

Keywords: defect prediction, coding style violations, metrics value changes, software changes

1. 序論

ソフトウェア開発では，レビューやテストなどの品質確保に必要な活動を効率的に実施する必要がある．この際に重点的に実施すべきモジュールを予測する目的で，不具合予測手法が研究されている [1]．不具合予測とは，過去のソフトウェア開発の履歴情報やソースコードから取り出した

[1] 南山大学
Nanzan University, Nagoya, Aichi 466–8673, Japan
[2] 慶應義塾大学
Keio University, Yokohama, Kanagawa 223–8522, Japan
[†1] 現在，ヤフー株式会社
Presently with Yahoo Japan Corporation
[a)] nagura@nanzan-u.ac.jp
[b)] michigan@ics.keio.ac.jp

特徴から，ソフトウェア変更により不具合が混入しそうなソフトウェアモジュールやファイルを予測する手法である．

プログラム変更前後のソースコードや開発履歴情報と不具合混入の関係を利用して，不具合混入を予測する手法が古くから提案されている [2], [3], [4], [5], [6]．プロジェクトの進行と不具合混入の関係性は，対象プロジェクトの属性（開発チームの構成，プロジェクトの規模，開発期間など）に影響を受ける．したがって，これらの手法では予測対象プロジェクトのプロジェクトデータの変化に対する不具合混入有無をあらかじめラベル付けし，それを教師データとして利用し訓練させ，不具合混入を予測する．一方で近年では，実際の不具合混入有無によってラベル付けした教師データを利用せずに，変更前後のソフトウェアメトリクスや，ソースコード片自体を分類することで不具合混入を予測する手法も提案されている [7], [8], [9], [10]．これらの方法は予測対象プロジェクトに対するラベル付けを必要としないが，十分な量の変更に関するプロジェクトデータを利用した訓練が必要になる．このように，これらの既存研究で提案される手法による不具合混入コミットの予測ではプロジェクト固有の特徴を利用する．教師データの利用有無にかかわらず対象プロジェクト自体のプロジェクトデータの分析が必要であり，プロジェクトに変更が十分に実施された後でないと適用が難しい．

異なるプロジェクトの分析により得られたメトリクス変化の傾向から教師データを作成し，予測対象プロジェクトに適用することで不具合を予測できるならば，予測対象プロジェクトのプロジェクトデータを利用した訓練の必要がない．そして対象プロジェクトに対して，予測対象となる変更前後のソースコードだけを与えるだけで不具合予測を実施できる．この実現には，対象プロジェクトの属性による影響を受けにくい（プロジェクト共通で不具合混入に対して同じような傾向を示す）メトリクスが必要になる．先行研究として，コード行数などの基本的なソースコードメトリクスやCKメトリクスなどのメトリクスを利用し，異なるプロジェクトの分析結果によって不具合混入を予測する手法もある [11], [12]．本研究ではそのようなメトリクスとして，コーディングスタイルに対する規約違反の増加量に基づくメトリクス（コーディング規約違反メトリクスと呼ぶ）に着目した．一般的にソースコードの「書き方」が整っていないと記述を理解しにくく，別の開発者や，ときには開発した本人も記述の意図を見誤ることがある．我々は，先行研究が利用する行数などの基本的なソフトウェアメトリクスの変化や，ソースコード内の識別子などの情報でなく，コーディング規約違反メトリクスによって，プロジェクトに関係なく不具合混入を説明できると考えた．そして以前に4つのオープンソースソフトウェア（OSS）プロジェクトを対象に観察し，特定のコーディング規約についてのコーディング規約違反メトリクスと不具合混入に関連が見られることを報告した [13]．

本研究ではまず30のOSSプロジェクトの分析を通した予備実験により，ソフトウェア変更時のコーディング規約違反メトリクスと，不具合混入の傾向を明らかにする．そして明らかにした傾向を利用し，変更前後のソースコードから，不具合の混入を判別する手法を提案する．提案手法は，予備実験で明らかにしたコーディング規約違反と不具合混入の関係に従い，OSSプロジェクトを対象にあらかじめ分析した予測モデルを利用する．これにより，予測対象プロジェクトに対する訓練なしで，不具合混入を予測する．

以降，まず2章で関連研究を，3章でコーディング規約違反メトリクスと予備実験を，4章で提案手法を，5章で評価実験を示し，6章で考察し，7章で結論を述べる．

2. 関連研究

2.1 不具合予測手法

ソフトウェア開発プロジェクトで過去に実施された変更に対し，変更前後のソースコードや履歴情報と不具合混入有無の関係をあらかじめラベル付けし，それを教師データとして利用する不具合予測手法が古くから提案されている．ソフトウェアの変更に対する特徴量ベクトルとの関係を利用する方法 [2]，変更前後のメトリクス変化との関係を利用する方法 [3], [4]，変更部分のコードに含まれる構文要素との関係を利用する方法 [5]，変更前後のソースコード構造の変化との関係を利用する方法 [6] がある．

このような教師あり学習により不具合予測を行う研究では，予測対象のプロジェクトについての過去の不具合に関する教師データが必要となる．そこで，近年では教師なし学習により不具合予測を行う手法も提案されている．それらは変更時のコミットに含まれる変更差分情報を分類し，その分類結果に従って不具合混入を予測する．変更前後のソフトウェアメトリクスの分類による方法 [7]，変更時のコミットに含まれる変更を表現するメトリクス [8] の分類による方法 [9]，変更ソースコード片自体の分類による方法 [10] がある．これらの利用には不具合混入有無によりラベル付けした教師データの作成が不要であるが，予測対象プロジェクトの特徴を学習する．したがって対象プロジェクト自体の十分な量の変更に関するプロジェクトデータを利用した訓練が必要である．

これらの不具合予測手法の利用には，教師データの利用有無にかかわらず対象プロジェクト自体の十分な量の変更に関するプロジェクトデータが必要である．したがって，プロジェクトが十分に変更された後でないと適用が難しい．前述の文献 [7] では，類似プロジェクトの利用や，メトリクスを変換する必要性があることも述べられている．

一方で異なるプロジェクトから集めたプロジェクトデータを利用し，不具合予測を実施する手法も提案されている．そのような不具合予測手法として24種類の手法を対象に，

予測性能の比較を目的にベンチマークを行った研究 [11] もあり，この研究でベンチマークの対象としたどの手法の予測性能にも大きな差がないことを，結論として述べている．対象としている不具合予測手法群は，コード行数などの基本的なソースコードメトリクスや CK メトリクスなどのメトリクスを利用し，不具合混入を予測する手法である．一例として，コード追加行数や削除行数や変更行数や複雑度を利用して不具合混入を予測する手法 [12] があげられている．

2.2 コーディング規約とソフトウェア品質

コーディング規約とは，ソフトウェア開発におけるコードの書き方に関する統一的な定義である [14]．プログラムの設計や実装に対する属人性を排除する目的で，プロジェクト開発ではコーディング規約が規定されることが多い．プロジェクトメンバ全体の統一的な認識に基づきコーディング規約を定め遵守することで，メンバ間でのプログラムの理解を促進し保守性や移植性を高めることが可能になる．

しかしコーディング規約として考えられるコードの書き方に対する特徴は非常に多い．すべてを正しく理解し遵守することが難しく，規約によっては特定の開発者のみに属人的に利用される場合もあることが指摘されている [15]．そこで，組織や開発グループで守るべきコーディング基準を定めて，それを規約として遵守しているケースも多いという調査結果も存在する [16]．

コーディング規約の遵守がソフトウェアの品質に関連する可能性も示されている．C 言語による開発プロジェクトを対象にした分析では，コーディングスタイルに基づく規約（組込みソフトウェア設計に利用される標準規格 MISRA-C）に対する違反と不具合含有率に相関があるという調査結果が示されている [17]．また，Java プログラムを対象にした調査により，識別子名とスコープの長さなどのコーディングスタイルに対して測定したメトリクスと不具合混入に関連があることも報告されている [18], [19]．

2.3 検査ツールの解析結果を利用する保守開発支援手法

本研究の提案手法では，コーディングスタイルの検査ツールを利用して得られた結果から不具合混入を予測する．不具合混入の予測とは直接的に関連しないが，提案手法と同様にプログラムの静的解析による検査ツールの解析結果を利用し，保守開発を支援する方法も提案されている．

先行研究 [20] では，Java エラー検査ツール FindBugs による警告が実際に欠陥を示しているかどうかをユーザに指定させ，その結果により警告をランキングする手法を提案している．先行研究 [21] では，Convolutional Neural Networks（CNNs）を利用し，エラー検査ツールが示す False Positive な警告を分類する手法を提案している．また先行研究 [22] では，本研究でも利用する Checkstyle などの静的な検査ツールを利用することで得られたプログラム解析結果が，意図に対して間違えているかどうかをユーザが指定することにより，False Positive な解析を抑制し解析精度の向上を狙うことのできるエコシステムを提案している．

3. コーディング規約違反と不具合混入の関連

3.1 コーディング規約違反メトリクスの定義

熟練した開発者であれば，開発メンバ間でのプログラムの保守性や移植性を保ち，さらに開発者自身に対する可読性の確保をも目的として，一般的にコーディング規約を守りながら開発する．2.2 節では組織や開発グループで守るべきコーディング基準を定めそれを規約として遵守する場合があることを述べた．さらに，組織や開発グループに関係なく熟練した開発者が共通的に遵守しているコーディング規約を，オープンソース/フリーソフトウェアコミュニティでガイドラインとして規定している場合もある[*1]．

2.2 節で示した関連研究 [18], [19] では，これらのガイドラインに規約として定められるようなコーディングスタイルに対して測定したメトリクスと，不具合混入に関連があることを報告している．このようにコーディング規約の遵守とソフトウェアの品質には関連があることが期待できる．我々も以前に 4 つの OSS のプロジェクトデータを観察し，Java コーディング規約準拠検査ツール Checkstyle [23] が検出するコーディング規約違反のうち，特定の規約違反と不具合の混入になにかしらの関連が見られることを報告した [13]．この観察では，ソフトウェア変更時のコーディング規約違反の増加量を，コーディング規約違反メトリクスとして定義した．コーディング規約違反メトリクスは，ある変更に関連するファイル群 F（$F = \{F_1, F_2, F_3, \ldots, F_n\}$）について，変更前後のコーディング規約違反の増加量の総和として算出する．コーディング規約違反の増加量は，その変更で編集操作が行われた編集量に影響を受ける．そこで，コーディング規約 R に対するコーディング規約違反の増加量の総和を編集量で正規化した正規化コーディング規約違反メトリクス $NormV_R$ を，式 (1) のように定義した．

$$NormV_R = \frac{\sum_{i=1}^{n} \phi(V_R(F_{i_{aft}}) - V_R(F_{i_{bef}}))}{\sum_{i=1}^{n}(LA_{F_i} + LD_{F_i})} \quad (1)$$

$$\phi(x) = \begin{cases} 0 & (x < 0 \text{ の場合}) \\ x & (\text{それ以外}) \end{cases}$$

ここで，$V_R(F_i)$ は，ファイル F_i のコーディング規約 R に対する規約違反数であり，$F_{i_{bef}}$ はファイル F_i の変更前の

[*1] たとえば，次のようなガイドラインがある（2019/7/31 閲覧）：
1) “GNU Coding Standards” https://www.gnu.org/prep/standards/standards.html
2) “Google Java Style Guide” http://google.github.io/styleguide/javaguide.html
3) “Code Conventions for the Java Programming Language” http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

状態のファイルを，$F_{i_{aft}}$ は変更後の状態のファイルを示す．またコーディング規約違反メトリクスは規約違反の増加量を表す．したがって，ある変更に対する規約違反の増加量 $V_R(F_{i_{aft}}) - V_R(F_{i_{bef}})$ が負であった場合，式 (1) ではその値を 0 となるように計算している．なお，LA_{F_i}，LD_{F_i} はそれぞれ対象コミットでのファイル F_i に対する追加行数，削除行数でありこの和がファイル F_i に対する編集量を示す．式 (1) ではその総和によって正規化している．

3.2 予備実験：不具合に関連するコーディング規約の調査

3.2.1 概要

本研究ではまず 30 の OSS プロジェクトの分析を通した予備実験により，ソフトウェア変更時に特定のコーディング規約違反が増加した場合に不具合混入の傾向があることと，そのコーディング規約違反の種類を明らかにする．

この分析では，各コミットとそれにより不具合が混入したかどうかの関連を，Rosen らの手法 [24] によって分析した結果を利用した．この手法では，コミットメッセージに含まれる単語によって不具合修正コミットを検出し，そのコミットが示す修正箇所のコードを最後に変更したコミットを，不具合を混入したコミットとしてラベル付けする．

Rosen らは提案手法を Web アプリケーションとして実装し公開している[*2]．利用者による Git リポジトリ URL の指定により，そのプロジェクトの各コミットと不具合混入との関連を分析でき，その分析結果も公開している．ただし多数のプロジェクトを Commit Guru の Web アプリケーションに分析させることは，Commit Guru Web アプリケーションの可用性に影響を及ぼす．そこで，公開された分析結果から各コミットの不具合混入有無に関する情報を収集した．なお Java 開発プロジェクトを対象にし，Java ファイルが占める割合の極端に少ないプロジェクトを収集対象から除外した．そして 2019 年 2 月から 4 月の間に Commit Guru に分析結果が存在していた 30 プロジェクトに対する分析結果を収集した．

対象としたプロジェクトのプロファイルを，**表 1** に示す．ここでは GitHub に登録されている全プロジェクトデータを対象にした調査結果 [25] に報告されているコミット数とコミッタ数についても記載することによって，一般的な OSS プロジェクトの規模を参考として示した．

そして収集した分析結果に対応する各プロジェクトの全コミットに対し，正規化コーディング規約違反メトリクス $NormV_R$ を全コーディング規約に対して算出した．そしてこのメトリクスと不具合混入との関連を，Commit Guru データセットから分析した．その手順を，3.2.2 項に示す．

3.2.2 不具合に関連するコーディング規約違反の特定手順

Java 言語を対象にしたコーディング規約準拠検査ツールである Checkstyle には，検査対象として 150 種類を超えるコーディング規約が用意されている．本研究では Checkstyle によるコーディング規約違反の検出結果と，Commit Guru の分析結果に含む不具合混入コミットのラベルにより，次の手順で不具合混入とコーディングスタイルに基づく規約違反の共起傾向を明らかにした．

(1) Commit Guru 分析結果に含む全コミットのコミットログに現れる各ソースファイルに対し変更前後のソースファイルを取得し，Checkstyle により 152 種類[*3]のコーディング規約の検査を実施した．検査対象のコーディング規約が属性値を必要とする場合は，Checkstyle で推奨されるデフォルト値を利用した．そして，各コーディング規約に対し正規化コーディング規約違反メトリクス $NormV_R$ を算出した．

(2) Commit Guru 分析結果に不具合混入としてラベル付けされたコミット ID に対応するコミットの，全コーディング規約に対する正規化コーディング規約違反メトリクスを，不具合混入としてラベル付けした．そして，規約ごとに各コミットの不具合混入有無と正規化コーディング規約違反メトリクスの値に有意差があるかどうかを，マン・ホイットニーの U 検定により検定し，帰無仮説 H_0 を「不具合有無で分類した 2 群のメトリクス値に差がない」とし p 値を算出した．

(3) ここまでの手順を全プロジェクトに対して実施し，その結果得られた全プロジェクトに対する p 値の集合に対して，コーディング規約ごとに次の手順を実施した：

- (a) 該当のコーディング規約違反が現れなかったプロジェクトを除外．
- (b) p 値の集合に対して外れ値除去を実施することで，特異なプロジェクトを除外．1 サンプルの T 検定を利用し，有意水準 5%での外れ値を除去．
- (c) 外れ値除去を行った p 値の集合の平均を算出．

(4) 得られた p 値の集合の平均が帰無仮説 H_0 を有意水準 5%で棄却する場合，該当のコーディング規約を不具合

表 1 調査対象プロジェクトのサマリ（n=30）
Table 1 Target project summary for pre-evaluation (n=30).

メトリクス種類		平均	標準偏差	最小値	中央値	最大値
コミット数		7,675	15,278	50	2,496	76,495
コミッタ数		148	211	1	54	819
実活動日数		1,040	1,051	14	783	4,074
Java ファイル数		1,659	3,638	2	462	17,661
Java ファイル割合 (%)		51.7	31.1	3.8	66.0	96.1
Java コード行数		245,015	550,537	123	76,400	2,630,102
Java コード行数割合 (%)		58.7	27.2	5.0	63.6	99.0
文献 [25]	コミット数	249	2,105	10	77	150,380
	コミッタ数	9	33	2	5	3,384

[*2] http://commit.guru/（2019/7/31 閲覧）

[*3] 予備実験と提案手法の実装で利用した Checkstyle ver.8.11 の実装で検査可能なすべてのコーディング規約数から，正規表現でルールを記述しないと Checkstyle の実行ができない 2 規約を除いた規約数である．

表 2 予備実験の結果（p 値平均の昇順に記載）
Table 2 Pre-evaluation results (in ascending order of p-value averages).

#	規約	規約違反の内容	p 値平均	有意差有 Proj. 数
1	EmptyLineSeparator	空行による意味的なまとまりの形成についての違反	0.000135	27
2	MagicNumber	-1，0，1，2 以外のマジックナンバーの使用	0.001867	26
3	OuterTypeFilename	クラス名とファイル名の不一致	0.001968	28
4	DeclarationOrder	変数定義のスコープ順序違反	0.002266	27
5	MultipleStringLiterals	同じ文字列リテラルの再出現	0.004098	25
6	NestedIfDepth	if 文のネストの深さの超過	0.004445	24
7	CyclomaticComplexity	サイクロマチック複雑度の超過	0.005516	24
8	ImportOrder	import 順序，区切りを目的とした空行に対する違反	0.005643	27
9	AvoidInlineConditionals	インライン条件演算子の使用	0.007427	25
10	JavadocVariable	フィールド変数への Javadoc コメント欠落	0.007811	27
11	HiddenField	フィールド変数名と同名のローカル変数の定義	0.008073	25
12	ReturnCount	メソッド内の複数箇所での return 文の記述	0.008799	24
13	SingleSpaceSeparator	空白 1 文字以外での文字列の区切りの出現	0.008848	23
14	FinalLocalVariable	代入されないローカル変数の final キーワード欠落	0.009200	26
15	ExplicitInitialization	フィールド変数へのデフォルト値による無意味な初期化	0.009275	25
16	FinalParameters	代入されないメソッドパラメータ変数の final キーワード欠落	0.009966	27

有無によって正規化コーディング規約違反メトリクスに有意差を生じる規約として記録した．

3.2.3 予備実験の結果

3.2.2 項の手順 (4) により記録された規約は，全 152 種類中 57 種類であった[*4]．このように予備実験で対象とした 30 プロジェクトでは，Checkstyle 検査対象コーディング規約のうちの約 1/3 に，不具合混入と有意な関連が見られた．特に有意である傾向の強い p 値が 0.01 未満（有意水準 1%以内で帰無仮説を棄却）のコーディング規約が 16 種類存在し，それらは 23 プロジェクト以上という多くのプロジェクトで有意差があることを結果として得た．紙面の都合上，それらの規約のみを**表 2** に示す．これらに限らず，Checkstyle が検査対象にしているコーディング規約に対する違反は当然プログラム理解や保守を困難にする．ここでは特にこれらの 16 種類への違反が，プロジェクトによらず不具合混入に関連することが観測できた．

4. 不具合予測手法の提案

4.1 概要

本研究では，3.2.3 項で明らかにした不具合混入に関連するコーディング規約群を利用し，正規化コーディング規約違反メトリクスに基づく不具合予測手法を提案する．予備実験において，不具合を含む変更では特定のコーディング規約群に対する正規化コーディング規約違反メトリクスの値が，有意に異なることを明らかにした．したがって，変更前後のソフトウェアのソースコードから正規化コーディング規約違反メトリクスの値を算出し，その値を 2 群に分類した際に不具合を含む変更に分類できる場合，その変更に不具合が含まれる可能性が高い．このような考えに基づき提案手法では変更に対する不具合混入を予測する．

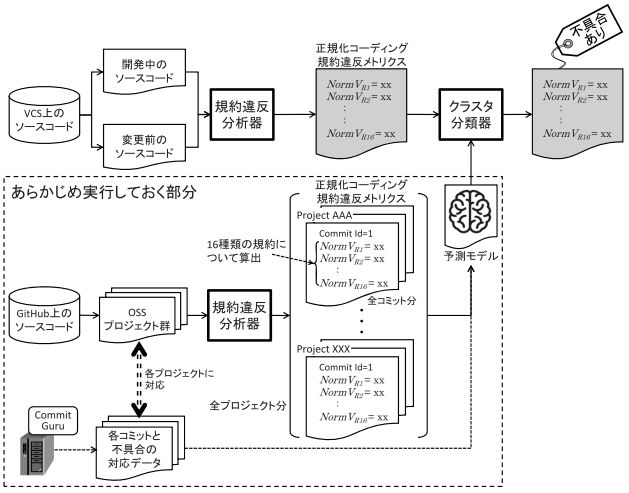

図 1 提案手法の概要
Fig. 1 Overview of proposed method.

提案手法の概要を**図 1** に示す．提案手法では，GitHub などの OSS リポジトリに存在する複数のプロジェクトに対してあらかじめ正規化コーディング規約違反メトリクスを算出し，予備実験の手順と同様に不具合混入とのラベル付けを行い，そのラベルに従い予測モデルを導出しておく．そのうえで，予測対象のプロジェクトの変更前後のソースコードから正規化コーディング規約違反メトリクスを計算し，導出した予測モデルによって分類する．その分類結果により対象の変更に不具合を含むかどうかを予測する．

本章の残りでは，提案手法の処理のうち，規約違反分析器，クラスタ分類器の実施する処理の詳細について述べる．

4.2 規約違反分析処理

まず変更前後のソースコードに対して Checkstyle によりコーディング規約違反の検査を実施する．この際，3.2.3 項で示した 16 種類のコーディング規約を利用する．コーディング規約が必要とする属性値には，予備実験と同様にデフォルト値を利用する．そしてコーディング規約違反の変化量を基に，16 種類のコーディング規約に対する正規化コーディング規約違反メトリクス $NormV_R$ を算出する．

OSS リポジトリに対する分析の際は，指定したプロジェクトの各コミットによる変更前後のソースファイルを取得し，ソースファイルが含む変更前後のソースコードに対してコーディング規約違反を検査し，正規化コーディング規

[*4] 実験結果から次の 2 種類（各 1 規約）については除外した：
- ファイル単位では検査を実施できない規約（ImportControl：外部 XML ファイルを利用する）
- デフォルト値での検査に意味がない規約（WriteTag：属性値で指定されたタグを検出する）

約違反メトリクスを算出する．算出したメトリクス値はコミット単位でまとめて保持する．この処理を，指定された全プロジェクトの全コミットに対して実施する．

4.3 クラスタ分類処理

前述のように，不具合予測を行う先行研究のうちの一部では予測対象プロジェクト固有の特徴を利用する．したがって，予測対象プロジェクトが十分に変更された後でないと適用が難しい．本研究ではより実用的な方法にすることを目的に，予測対象プロジェクトのある特定の変更のみを利用した場合に，不具合混入を予測できる手法を提案する．予備実験の結果，3.2.3 項で示した 16 種類のコーディング規約についてはプロジェクトの相違にかかわらず，規約違反と不具合混入が関係していることを明らかにした．この結果を利用し提案手法では，予測対象とは異なるプロジェクトを利用した教師あり学習によって不具合混入の予測モデルをあらかじめ導出する．そして導出した予測モデルに従って変更に対する不具合混入を予測する．予測モデルの導出には，複数の OSS プロジェクトを対象にあらかじめ分析を行った結果を利用する．これにより予測対象プロジェクトに対する学習がいっさい不要になり，ある特定の変更に対する正規化コーディング規約違反メトリクスだけを算出できれば予測が可能になる．

クラスタ分類に利用する手法としては，データ数が多い場合でも計算コストをかけず実用的な精度で予測できる手法として，ランダムフォレストを利用した．ランダムフォレストによる学習の際の説明変数としては，4.2 節でコミット単位で算出した，コーディング規約ごとの正規化コーディング規約違反メトリクスを利用する．そして目的変数として，正規化コーディング規約違反メトリクスに対応する各コミットについて，Commit Guru のデータセットを利用し不具合混入に関連するかどうかをラベル付けした結果を利用することで，予測モデルを導出する．

変更に対する不具合混入の予測の際は，まず予測対象プロジェクトの該当の変更前後のコードに対して 4.2 節の手順で正規化コーディング規約違反メトリクスを算出する．そして予測モデルに基づき，算出したメトリクス値が不具合混入に関連するか分類する．分類の際の閾値には，デフォルト値である 0.5 を利用した．

5. 実装・評価

5.1 実装

4 章で示した提案手法を，Python 言語を用い実装した．

クラスタ分類器の実装の際は，機械学習ライブラリに scikit-learn 0.20.3 を利用した．また図 1 で入力として与える OSS プロジェクトには，3.2.3 項の予備実験で使用した 30 プロジェクトのうち，16 規約すべてに対して有意水準 1%以内で帰無仮説を棄却した 14 プロジェクト（それらが含むコミットのうち，Commit Guru のデータセットにより不具合混入に関連するかどうかのラベル付けを実施できた合計 54,161 コミット）を利用した．また，ランダムフォレストを実施する際に指定するハイパーパラメータについては，教師データを利用しチューニングを行った結果，決定木の個数 300 個，決定木の深さの最大値 15 を指定した．

表 3 評価実験の対象プロジェクトのサマリ
Table 3 Target project summary for evaluation.

メトリクス種類	平均	標準偏差	最小値	中央値	最大値
コミット数	3,037	2,787	650	2,099	8,172
コミッタ数	75	32	9	87	107
実活動日数	725	573	247	472	1,969
Java ファイル数	253	236	22	158	652
Java ファイル割合 (%)	50.5	26.8	9.7	54.2	92.1
Java コード行数	42,832	42,311	2,826	22,516	142,233
Java コード行数割合 (%)	68.7	18.6	30.2	72.1	92.3

5.2 評価実験

5.2.1 概要

評価実験を行い，提案方式の有効性を評価する．実験にあたって，次のリサーチクエスチョンを設定することで，実験目的を明らかにした．

【RQ1】 利用するコーディング規約の種類や数が不具合予測性能に影響を与えるか

提案手法は，予備実験で明らかにした 16 種類のコーディング規約を利用する．これらは，p 値が 0.01 未満という十分に有意と考えられる規約だが，どのコーディング規約が不具合予測性能にどれだけの影響を与えるかが明らかでない．

【RQ2】 既存手法に比べてどの程度の性能で不具合予測を実施可能か

提案手法は，予測対象プロジェクトと異なるプロジェクトのプロジェクトデータを教師データとして利用する．既存の一部の教師あり不具合予測手法のように，対象プロジェクトについての教師データを必要としない．同様に対象プロジェクトについての教師データを必要としない教師なし学習による不具合予測手法 [7], [8], [9], [10] や，他のプロジェクトのデータを教師データとして利用する不具合予測手法 [12] に対して，どの程度の性能で不具合予測が可能であるか明らかでない．

OSS プロジェクトを対象に，提案手法を適用した評価実験を行うことによって，設定したリサーチクエスチョンを解決する．予測結果が正しいかどうかを判断できるように，対象としたプロジェクトには，Commit Guru データセットが存在するプロジェクトを利用した．そして，2019 年 5 月に分析結果が存在していた，予備実験で利用したものとは異なる 8 プロジェクトを利用した．対象としたプロジェクトのプロファイルを，**表 3** に示す．

各コーディング規約の利用による影響を明らかにする目的で，次の 2 つの実験を行った．

実験 1：各規約の利用が不具合予測に与える影響の調査

実験 2：複数規約の利用が不具合予測に与える影響の調査

実験 1 では各コーディング規約を利用した場合の，実験 2 では複数のコーディング規約違反メトリクスを同時に利用した場合の不具合予測の評価指標を測定した．これらの実験により RQ1 を解決する．そして実験 2 において 16 規約を利用した場合の不具合予測の評価指標を先行研究と比較することで，RQ2 を解決する．

5.2.2 実験手順と結果

実験 1 各規約の利用が不具合予測に与える影響の調査

調査対象の 8 プロジェクトに対して，16 種類の規約をそれぞれ利用して，不具合予測を実施した（図 1 で 16 規約をすべて利用せず，1 つずつ利用して 16 回実験を行った）．各規約利用時の評価指標（精度，適合率，再現率，F1 値，AUC の値）を全プロジェクトに対して平均したものを**表 4** に示す．なお，各行はそれぞれの規約を表し，先頭の番号は，表 2 の各規約の先頭の番号に対応する．また，各値には標準偏差を括弧内に示した．

表 4 実験 1 の結果（単位：%）
Table 4 Result of experiment 1 (unit: %).

#	精度	適合率	再現率	F1 値	AUC
1	81.76 (4.75)	71.73 (13.54)	3.64 (2.77)	6.78 (4.8)	51.67 (1.35)
2	81.83 (4.82)	64.82 (27.42)	4.61 (3.81)	8.32 (6.65)	52.07 (1.72)
3	81.49 (4.86)	57.71 (28.54)	1.32 (1.45)	2.54 (2.75)	50.60 (0.72)
4	82.07 (4.51)	68.78 (12.21)	5.88 (3.69)	10.62 (6.24)	52.68 (1.73)
5	81.54 (4.91)	63.87 (31.47)	1.75 (2.21)	3.31 (4.05)	50.81 (1.08)
6	81.55 (4.64)	46.48 (37.60)	1.32 (1.15)	2.56 (2.23)	50.60 (0.56)
7	81.42 (4.76)	52.68 (43.88)	0.24 (0.21)	0.47 (0.43)	50.11 (0.10)
8	81.59 (5.03)	63.88 (19.06)	2.95 (4.86)	5.21 (8.07)	51.36 (2.40)
9	81.53 (4.75)	51.55 (41.54)	1.00 (0.96)	1.96 (1.86)	50.48 (0.47)
10	82.07 (4.72)	72.89 (14.56)	5.94 (4.91)	10.60 (7.86)	52.78 (2.43)
11	81.62 (4.72)	58.97 (31.38)	2.37 (1.91)	4.50 (3.53)	51.08 (0.92)
12	81.44 (4.75)	46.23 (40.96)	0.41 (0.45)	0.81 (0.89)	50.19 (0.22)
13	81.42 (4.79)	37.34 (41.59)	0.60 (0.63)	1.17 (1.22)	50.25 (0.25)
14	82.42 (5.02)	71.60 (13.03)	11.73 (10.66)	18.35 (13.41)	55.34 (5.09)
15	81.56 (4.59)	58.79 (37.64)	1.57 (1.23)	3.04 (2.36)	50.70 (0.61)
16	83.88 (5.13)	65.33 (6.95)	30.99 (13.98)	40.54 (12.77)	63.81 (6.97)

表 5 実験 2 の結果（単位：%）
Table 5 Result of experiment 2 (unit: %).

規約数	精度	適合率	再現率	F1 値	AUC
(少) 1	81.76 (4.75)	71.73 (13.54)	3.64 (2.77)	6.78 (4.88)	51.67 (1.35)
2	82.19 (4.79)	70.56 (10.61)	7.78 (5.69)	13.46 (8.96)	53.53 (2.67)
3	82.21 (4.71)	70.45 (14.57)	8.00 (6.21)	13.69 (9.59)	53.64 (2.92)
4	83.09 (4.65)	68.13 (5.77)	19.35 (8.82)	28.95 (9.96)	58.62 (4.03)
5	83.55 (4.71)	66.34 (5.73)	27.30 (10.76)	37.27 (9.53)	62.01 (4.99)
6	83.79 (4.58)	66.71 (5.82)	28.23 (9.62)	38.68 (8.81)	62.51 (4.62)
7	83.77 (4.50)	65.22 (4.90)	29.17 (9.38)	39.48 (8.46)	62.82 (4.57)
8	83.97 (4.61)	64.41 (5.90)	34.77 (10.75)	44.07 (7.72)	65.17 (5.24)
9	83.91 (4.61)	60.84 (5.31)	42.60 (10.69)	49.17 (6.28)	68.15 (5.30)
10	83.37 (4.84)	57.24 (5.88)	49.25 (9.45)	52.20 (4.62)	70.38 (5.04)
11	83.77 (4.77)	58.86 (6.33)	48.73 (9.77)	52.53 (5.21)	70.43 (5.21)
12	83.88 (4.60)	59.78 (6.28)	46.83 (9.80)	51.64 (4.94)	69.77 (5.02)
13	84.12 (4.56)	61.00 (6.25)	45.88 (10.57)	51.41 (5.65)	69.54 (5.33)
14	83.71 (4.91)	57.86 (6.23)	53.71 (10.32)	54.87 (5.26)	72.34 (5.49)
15	83.78 (4.91)	58.30 (5.89)	52.39 (11.02)	54.30 (5.61)	71.88 (5.74)
(多) 16	83.52 (4.93)	55.89 (5.68)	62.06 (10.47)	58.18 (5.31)	75.42 (5.92)

実験 2 複数規約の利用が不具合予測に与える影響の調査

調査対象の 8 プロジェクトに対して，利用する規約数を p 値平均の小さい順に 1〜16 規約に変化させ，不具合予測を実施した（図 1 で 16 規約をすべて利用せず，1 つずつ増やしながら 16 回実験を行った）．利用する規約数を変化させた場合の評価指標（精度，適合率，再現率，F1 値，AUC の値）を全プロジェクトに対して平均したものを**表 5** に示す．なお，各行の先頭の数字は，表 2 に記載した規約のうち p 値平均の昇順に何個の規約を利用したかを表している．また，各値には標準偏差を括弧内に示した．

6. 考察

6.1 RQ1（利用するコーディング規約の種類や数が不具合予測性能に与える影響）について

実験 1 では，各コーディング規約に基づくコーディング規約違反メトリクスに，どの程度の不具合予測性能があるかを調査した．表 4 に示したように，どのコーディング規約にも大きな差がみられなかった．さらに値の大小に予備実験の結果である p 値平均の順位との関連がみられず，また値のばらつきも大きい．特に再現率に関しては，ほとんどの規約で非常に低い．ただし #14 と #16 については他と比べて高くなっており，それらは F1 値や AUC で表される不具合予測性能も他の規約より高い値を示している．

一方で，実験 2 では複数のコーディング規約に対するコーディング規約違反メトリクスを利用した場合の不具合予測性能を調査した．**図 2** に，利用した規約数と不具合予測性能の関係を記載する．このグラフや表 5 に示したように，今回対象にした 16 個の規約に対しては，利用規約数が多い場合に，評価指標のうち F1 値や AUC といった予測性能を示す指標が向上している．これは，再現率が向上していることに起因する．すなわち，実験 1 で示したように個々のコーディング規約違反メトリクスの利用だと False Negative の予測結果が多く含まれるが，複数のコーディング規約違反の変化を同時に利用することで False Negative の予測結果を減らすことができたことが分かる．

なお実験 1 と実験 2 では p 値が 0.01 未満の 16 規約を利用して不具合予測性能の評価を行った．3.2.3 項で示した

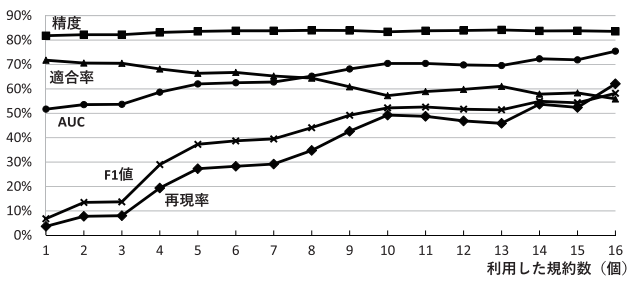

図 2 利用した規約数と不具合予測性能
Fig. 2 Relation of defect prediction performance with used coding standards.

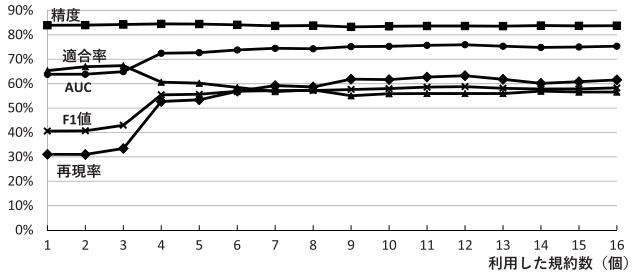

図 3 利用した規約数と不具合予測性能（利用するコーディング規約を予測性能順に変化）

Fig. 3 Relation of defect prediction performance with used coding standards (where used coding standards are ordered according to prediction performance).

ように，一般的に有意とされる p 値が 0.05 未満（有意水準 5%以内で帰無仮説を棄却）の規約が，57 種類存在した．そこで，16 規約に加え，残りの 41 規約を含む全 57 規約を対象に，実験 2 と同様に p 値の昇順に追加し不具合予測を実施した．その結果，AUC は 18 個目の規約を適用した場合に最大値（78.62%）を示し，F1 値は 19 個目の規約を適用した場合に最大値（60.04%）を示した．そしてそれ以上規約を追加しても，予測性能が高くなることはなかった．したがって，不具合予測に関して今回実験を行ったプロジェクト群では，説明変数となったコーディング規約違反メトリクスの種類はたかだか 19 個であったと考える．

また実験 2 では，最終的に 4 章で示したように 16 規約すべてを利用して不具合予測を実施することを前提に，利用する規約数を p 値平均の小さい順に増やした場合の不具合予測性能の変化を調査した．この際に p 値平均の値の大小を，利用する規約の順序付けに利用した．p 値は帰無仮説が棄却できる確率であり，要因の影響の強さを表現するものではない．そしてここでは，帰無仮説が棄却できる（＝不具合有無での分類とメトリクス値の差に関連がある）と期待できる順での順序付けに利用した．その一方で，実験 1 では前述のように一部の規約で不具合予測性能が他の規約よりも高い値になり，それらの要因の影響が強いことを示した．そこで参考として，実験 1 での予測性能（AUC）の高い順に規約数を変化させて，実験 2 と同様の調査を実施した．図 2 と同様に，利用した規約数と各評価指標に基づく不具合予測性能の関係を，**図 3** に示す．このグラフに示すように AUC の高い順に規約数を変化させた場合，少ない規約数でも図 2 と比較して再現率は低くならず，評価指標が高い値を示す．そして，実験対象のプロジェクト群に対してはたかだか 9 規約の利用で，F1 値や AUC が 16 規約すべてを利用した場合と同程度の値を示し，予測には十分であったことが読み取れる．

このことは，予測対象プロジェクトのコミットを利用して予測を実施した結果から得られる．したがって，予測対象プロジェクトに依存する．提案手法は予測対象プロジェクト自体のデータの分析を必要とせずに不具合混入予測を実施する手法であり，このような予測対象プロジェクト自体のデータ分析によって得られた評価指標を規約の選択基準として利用できない．しかし，もし仮に予測対象プロジェクトのプロジェクトデータの一部をパイロット的に利用できる状況ならば，予測対象プロジェクトに対して効果の高い規約に絞り込んで予測を実施できる可能性がある．

表 6 16 規約を利用した場合の予測性能評価指標（単位：%）

Table 6 Prediction performance results when using 16 coding standards (unit: %).

GitHub プロジェクト名	精度	適合率	再現率	F1 値	AUC
owncloud/android	84.02	62.20	62.60	62.40	76.19
ctripcorp/apollo	88.50	48.09	80.43	60.19	84.95
AppIntro/AppIntro	90.77	62.50	75.58	68.42	84.33
LMAX-Exchange/disruptor	75.68	45.91	62.02	52.76	70.77
lingochamp/FileDownloader	77.06	57.51	52.11	54.68	69.10
brettwooldridge/HikariCP	82.00	54.84	47.00	50.62	68.78
mikepenz/MaterialDrawer	86.86	57.32	59.87	58.57	75.85
realm/realm-java	83.31	58.74	56.88	57.80	73.42
平均値	83.52	55.89	62.06	58.18	75.42
中央値	83.66	57.42	60.95	58.18	74.63

6.2 RQ2（先行研究との比較）について

6.2.1 教師なし学習により不具合を予測する手法との比較

実験 2 において 16 規約を利用して予測を行った場合の評価指標を，対象プロジェクトごとに**表 6** に示す．先行研究 [7] では AUC の中央値が 71%，先行研究 [8] では AUC の平均値が 73%，F1 値の平均値が 43%，先行研究 [9] では F1 値の平均値が 45%，先行研究 [10] では AUC の平均値が 80〜82%と，それぞれ報告している．これらと比較し，提案手法による不具合予測は同等の性能であった．ただし先行研究 [10] より AUC の平均値が若干劣る．

先行研究 [7] では，特定の開発コミュニティの公開データセットに含まれるプロジェクトを，学習用プロジェクトと評価用プロジェクトに分け，学習用プロジェクトを学習させたうえで実験を行っている．この公開データセットは提案手法で利用したプロジェクトデータのようにコミット単位で用意されているものではない．したがって，コミット単位で分析した提案手法での評価実験の結果との比較は適切ではない可能性がある．しかし提案手法では公開データセットのようなあらかじめ分析を目的として用意されたデータセットではなく，コミットという開発プロセスにおいて通常行われる活動の単位で得られたデータの分析によって，先行研究 [7] で実施された予測と同等の不具合予測を実施できることを，評価実験により示すことができた．

なお先行研究 [7] では比較を目的に，提案手法と同様に教師あり学習（ランダムフォレスト）で分類する方法も実装しており，AUC の中央値が 70%と報告している．この実装は，提案手法とは利用するメトリクスのみが異なることになる．前述のように先行研究 [7] では，特定の開発コ

ミュニティのプロジェクトを対象にしている．一方で提案手法は，開発コミュニティを特定せず，利用ドメインを含め予測対象プロジェクトと異なる開発コミュニティのプロジェクトを学習することで，同等の予測性能を示している．このことから正規化コーディング規約違反メトリクスが対象プロジェクトに依存しないメトリクスであるといえる．

このように，実験結果からは先行研究に近い予測性能で不具合の予測を実施できることが示されたが，先行研究の一部（先行研究 [10]）に対しては予測性能が若干劣っていた．しかし提案手法は，予測対象プロジェクトのプロジェクトデータに依存せずに，評価実験の結果にあるような予測性能で不具合予測を実施できる．先行研究の多くは予測対象のプロジェクト自体のプロジェクトデータを分析することで予測モデルを作成しており，プロジェクトに対し実施された変更データが十分に存在するかどうかに予測性能が影響する．提案手法はそのような影響を受けずに，先行研究に近い予測性能で不具合予測を実施できる．

6.2.2 他プロジェクトのデータを利用する教師あり学習により不具合を予測する手法との比較

提案手法と同様に，他プロジェクトのデータを教師データとして利用する不具合予測手法 [12] で提案されているメトリクスと，予測性能を比較する．まずそれらのメトリクスによって提案手法と同様の手順で不具合予測を実施する追加実験を行い，提案手法での評価実験の結果と比較した．

先行研究 [12] では，次の 6 つのメトリクスを定義し，ソフトウェア変更による不具合予測を実施している．

(1) 変更による追加行数 ÷ 全体のコード行数
(2) 変更による削除行数 ÷ 全体のコード行数
(3) 変更による修正行数 ÷ 全体のコード行数
(4) リリース前不具合件数 ÷ 全体のコード行数
(5) (追加+修正+削除行数) ÷ (変更期間コミット数+1)
(6) 合計サイクロマチック数 ÷ 全体のコード行数

このうち (4)，(5) については，リリースなどの一定期間での変更におけるコミットに対する不具合予測を想定している．提案手法はコミット単位で不具合混入の予測を実施しており，評価実験で利用したプロジェクトはリリースの概念があるものだけではなかった．したがって，(4) のメトリクスを算出できない場合があり，ここでは除外した．さらにコミット単位での予測の実施であり，変更期間にコミットが存在しない．そこで，(5) の除数をつねに 1 にした．

追加実験では，まず 3.2 節の手順と同様に予備実験を行った．はじめに，提案手法の予備実験で対象にした 30 プロジェクトを対象に，全コミットによる変更に対して (1)，(2)，(3)，(5)，(6) の 5 種類のメトリクスを算出した．次に，Commit Guru 分析結果を用い，各コミットに対して不具合混入のラベル付けを行った．そして 5 種類のメトリクス値が不具合混入と関連するかどうかを，マン・ホイットニーの U 検定により検定した．全プロジェクトについて算出した p 値から，予備実験と同様に 1 サンプルの T 検定（有意水準 5%）によって外れ値除去を実施し平均した結果を**表 7** に示す．なお，提案手法での予備実験の結果（表 2）では，p 値が 0.01 未満の規約のみを特に有意である傾向が強い規約と見なして載せたが，ここではそのようなメトリクスが存在しなかった．そこで，表 7 には，5 種類のメトリクスすべてを p 値の昇順に記載している．この表にあるように (6) 以外のメトリクスが一般的に有意である傾向があると見なされる p 値が 0.05 未満を示した．

表 7 先行研究 [12] のメトリクスを利用した予備実験の結果（p 値平均の昇順に記載）
Table 7 Pre-evaluation results using metrics in Ref. [12] (in ascending order of p-value averages).

#	メトリクス	p 値平均	有意差有 Proj. 数
1	(3) 変更による修正行数 ÷ 全体のコード行数	0.017335	26
2	(1) 変更による追加行数 ÷ 全体のコード行数	0.017868	26
3	(5) (追加+修正+削除行数) ÷ (変更期間コミット数+1)	0.020952	24
4	(2) 変更による削除行数 ÷ 全体のコード行数	0.033767	21
5	(6) 合計サイクロマチック数 ÷ 全体のコード行数	0.079680	19

表 8 先行研究 [12] の 4 メトリクスを利用した場合の予測性能評価指標（単位：%）
Table 8 Prediction performance results when using 4 metrics in Ref. [12] (unit: %).

GitHub プロジェクト名	精度	適合率	再現率	F1 値	AUC
owncloud/android	78.02	28.15	2.46	4.52	50.39
ctripcorp/apollo	88.35	9.09	0.85	1.56	49.91
AppIntro/AppIntro	83.67	14.29	4.65	7.02	50.19
LMAX-Exchange/disruptor	77.45	37.50	4.33	7.76	51.15
lingochamp/FileDownloader	72.66	33.33	2.82	5.19	50.39
brettwooldridge/HikariCP	79.28	21.43	2.07	3.78	50.11
mikepenz/MaterialDrawer	83.70	22.58	2.24	4.07	50.42
realm/realm-java	79.72	31.71	0.79	1.54	50.18
平均値	80.36	24.76	2.53	4.43	50.34
中央値	79.50	25.36	2.35	4.29	50.29

次に，5.2.2 項の実験 2 の手順と同様に，先行研究 [12] のメトリクスに対して評価実験を行った．5.2.2 項の提案手法に対する評価実験では p 値が 0.01 未満の規約を利用したが，ここでは表 7 において p 値が 0.05 未満であった上位 4 つのメトリクスを利用した．そして 4 メトリクスすべてについて，メトリクスの選択の基準である有意水準 5%以内（p 値が 0.05 未満）で帰無仮説を棄却した 20 プロジェクト（それらが含むコミットのうち，Commit Guru のデータセットにより不具合混入に関連するかどうかのラベル付けを実施できた合計 68,324 コミット）を教師データとして利用し，評価実験で利用した 8 プロジェクトの全コミットの不具合混入を予測した．その際のハイパーパラメータについては，教師データを利用しチューニングを行った結果，決定木の個数 300 個，決定木の深さの最大値 15 を指定した．全 4 種類のメトリクスを利用し 8 プロジェクトに対して不具合予測を実施した際の評価指標の値を**表 8** に示す．

表 6 と比較すると，評価実験に利用したデータでは，適合率，再現率ともに，先行研究 [12] が提案手法よりも悪い結果になった．先行研究 [12] のメトリクスを利用した予備実験では，提案手法で特に有意である傾向が強いと見なした p 値が 0.01 未満のメトリクスが存在しなかった．前述のように p 値が小さいほうがより帰無仮説を棄却でき，不具合有無での分類とメトリクス値の差に関連があることが期待できる．利用したメトリクスに対する p 値が提案手法の評価実験よりも非常に大きかったことを考えると，予備実験や評価実験の対象プロジェクトでは先行研究 [12] のメトリクス値の差と不具合有無に関連が弱かった可能性が高い．そしてその結果，提案手法の規約群に基づくメトリクス値を利用した場合よりも予測性能が悪くなったと考えることができる．

このことは外的妥当性にも関連し，実験に利用したプロジェクトが先行研究 [12] のメトリクスによる分類に対して困難な特性であり，教師データや評価対象のプロジェクトデータとして利用することが適切でなかった可能性があったとも考えられる．したがって今回の評価実験では提案手法の方が予測性能が良かったが，その結果は利用したプロジェクトに依存し，このことのみで先行研究に対する優位性を主張できるものではなく，双方のメトリクスを補完的に利用することで外的妥当性の問題を緩和できると考える．

6.3 不具合関連傾向を示したコーディング規約について

3.2 節では予備実験により，16 種類の規約への違反がプロジェクトによらず不具合混入に関連することを示した．これらは 30 の OSS を対象に正規化コーディング規約違反メトリクスの値を算出した際に，不具合混入有無で有意差を生じる規約群であり，予備実験で対象にした 30 プロジェクトにおいて，違反した際に注目に値するコーディング規約群である．予備実験での対象ではないプロジェクトに対しても同様の性質を示すかどうかを確認した．

手順としては，5 章で述べた評価実験で対象にした 8 プロジェクトに対し，表 2 の 16 種類のコーディング規約を利用し，3.2.2 項の予備実験と同様の手順で正規化コーディング規約違反メトリクスの算出と検定を実施した．3.2.2 項の手順と同様に 8 プロジェクトに対して算出した p 値を平均した値を，**表 9** に示す．この表にあるように，評価実験の対象の 8 プロジェクトでも，予備実験と同様に 16 規約中 14 規約で不具合混入有無に対して規約違反が有意である傾向を示した．一方，#6 と #13 の規約については，必ずしも有意であるといえない結果になった．

それらの規約について，プロジェクトごとの p 値を**表 10** に示す．この表より，有意である傾向を示すプロジェクトとそうでないプロジェクトで，値に大きな差があることが読み取れる．有意である傾向を示したプロジェクトでは p 値が非常に小さい値を示しており，コーディング規約違反と不具合混入に強い関係があったことが分かる．

一般的に遵守が当然と考えられる規約（たとえば，LineLength：1 行の文字数超過）については，不具合予測への利用効果が小さい[*5]．同様に開発コミュニティで遵守が当然と考えられ，遵守を定められているような規約も不具合予測への利用効果が小さく，遵守と不具合混入の有無には関係性が低いはずである．表 10 で有意差を示さなかったプロジェクトでは，#6 や #13 の規約を遵守すべき規約として定めて運用していた可能性を推測できる．

このように考えると，今回の予備実験の結果として得ら

表 9 評価実験の対象プロジェクトにおける選択した規約と不具合混入有無との関連（表 2 の順に記載）

Table 9 Relation of defect existence with the selected coding standard in the target projects (in the order of Table 2).

#	規約	p 値		有意差有 Proj. 数
		順位	平均	
1	EmptyLineSeparator	4	1.5×10^{-7}	7
2	MagicNumber	11	0.003896	8
3	OuterTypeFilename	3	1.0×10^{-7}	7
4	DeclarationOrder	8	0.000033	7
5	MultipleStringLiterals	9	0.002517	8
6	NestedIfDepth	16	0.093648	5
7	CyclomaticComplexity	12	0.009382	7
8	ImportOrder	6	0.000002	8
9	AvoidInlineConditionals	10	0.003225	8
10	JavadocVariable	1	8.5×10^{-10}	7
11	HiddenField	7	0.000008	7
12	ReturnCount	14	0.020154	7
13	SingleSpaceSeparator	15	0.054445	4
14	FinalLocalVariable	5	5.9×10^{-7}	7
15	ExplicitInitialization	13	0.014819	7
16	FinalParameters	2	5.4×10^{-9}	7

※網掛けした行は，p 値平均が 0.05 を超えた規約．

表 10 表 9 で有意である傾向を示さなかった規約についてプロジェクトごとに得られた p 値

Table 10 Each project's P-values for the two coding standards which were found to be insignificant in Table 9.

GitHub プロジェクト名	#6 NestedIfDepth	#13 SingleSpaceSeparator
owncloud/android	4.7×10^{-25}	1.0×10^{-23}
ctripcorp/apollo	0.000788	2.5×10^{-5}
AppIntro/AppIntro	0.050673	0.067978
LMAX-Exchange/disruptor	0.260769	0.180516
lingochamp/FileDownloader	3.6×10^{-9}	0.001326
brettwooldridge/HikariCP	0.436906	0.092672
mikepenz/MaterialDrawer	5.0×10^{-5}	0.093038
realm/realm-java	1.9×10^{-9}	9.3×10^{-10}
平均値	0.093648	0.054445
中央値	0.000419	0.034652

※網掛けしたセルは，p 値平均が 0.05 を超えた値．

[*5] Checkstyle [23] では，3.1 節にあるガイドラインのうち 2) と 3) に対応したチェック用の設定ファイルを公開している．これらに共通で含まれる規約群を一般的に遵守が当然である規約と見なすと（LineLength もこれに含まれる），それらの規約群は今回明らかにした 16 種類の規約群には 1 つも含まれなかった．

れた 16 個の規約群から予測対象プロジェクトで遵守すべき規約として定められている規約を除外した規約群は，不具合と関連しやすいといえる．予測対象プロジェクトに合わせて規約を選択して利用することで，高い予測性能で不具合予測を実施できる可能性がある．

6.4 妥当性への脅威

6.4.1 外的妥当性

プロジェクトの選択には，コード行数や利用ドメインに特にバイアスをかけておらず，特に予備実験では調査対象のプロジェクトの偏りをなるべく減らすように，コミット数，コミッタ数，実活動日数，Java ファイル数について，様々な規模の OSS プロジェクトを 30 個選択した．その際に表 1 に示したように，文献 [25] に記載される GitHub に登録されている全プロジェクトデータを対象に分析した結果に対し，コミット数の最小値，最大値の偏りがなるべく大きく相違しないようにプロジェクトを選択した．しかし選択したプロジェクトが外的妥当性へ影響を与えた可能性を完全に否定できない．

同様に評価実験で選択したプロジェクトも偏りがないように選択したが，8 プロジェクトであり多いとはいえない．また 6.3 節で示したように，評価対象のプロジェクトによっては不具合予測への利用効果が小さいコーディング規約が存在した．このような特異なプロジェクトの存在が，実験結果に影響する．このように評価実験については，プロジェクトの選択に加え，対象プロジェクト数が多くないことも，外的妥当性に影響を与えた可能性がある．

6.2.2 項で述べたように，先行研究 [12] のメトリクスによる予測性能が良くなかった結果も，評価実験に利用したプロジェクトが外的妥当性に影響を与えた可能性がある．先行研究 [12] でもプロジェクト選択による妥当性への脅威が指摘されており，6.2.2 項で前述したように，提案手法を含め様々なメトリクスを補完的に利用することでプロジェクト選択による外的妥当性の問題を緩和できる．

6.4.2 内的妥当性

予備実験と提案手法の評価には Commit Guru に記録された不具合混入と修正の関連をラベル付けした結果を利用した．Commit Guru では，前述のようにコミットメッセージのキーワードを利用してラベル付けを行っている．したがって，たとえば不具合を修正したコミットのコミットメッセージがキーワードを含まない場合は，不具合を見落とす可能性がある．このように提案手法は，不具合修正コミットかどうかを判断する際に利用した手法の検出能力に，内的妥当性が影響を受ける．

極端な場合を想定すると，異なる手法で不具合修正コミットと判断した結果が，Commit Guru でのラベル付け結果と異なってしまうことも考えられる．本研究では 30 個のプロジェクトを対象に，特定のプロジェクトによらずにコーディング規約違反と不具合混入の関係を導いた．仮に異なる手法による不具合検出結果が Commit Guru のラベル付け結果と異なっていた場合でも，対象プロジェクト数が少数でなければ，同様にコーディング規約違反と不具合混入にプロジェクトによらない関係性を導くことができる．すなわち内的妥当性は Commit Guru の不具合修正コミット検出能力に影響を受けるものの，仮に異なる手法で実施した不具合検出結果を利用したとしても同じような傾向を示し，正しい予測を実施できる見込みであると考える．

6.4.3 構成概念妥当性

3.1 節で述べた式 (1) では，あるコミットでのファイル F_i に対するコーディング規約違反の増加量 $V_R(F_{i_{aft}}) - V_R(F_{i_{bef}})$ が負の（減少している）場合に，値を 0 にしている．あるコミットに，特定のコーディング規約違反が増加したファイルと減少した別のファイルが関連している場合に式 (1) の分子で合計値を求めると，不具合を含む可能性を示唆している前者の増加量を，後者の減少量によって打ち消してしまう．コーディング規約違反増加が不具合に関連するという前提のもと，これを防ぐ目的で負の値による影響を無効化している．同様のことはファイル内変更にもあてはまる．あるコミットに関連したファイル内の一部でコーディング規約違反が増加し，別の一部で減少した場合である．しかし，規約違反の増加量がファイル単位の変化量であり，これについては表現できない．したがって，規約違反増加量に減少分の影響が現れ，本来観測されるべき規約違反増加量より少ない値が観測される可能性がある．このように，コーディング規約違反メトリクスによって表現しようと意図している特性を十分に表現できていない可能性があり，その場合は構成概念妥当性に影響を与える．

なお式 (1) では，コード編集量にコーディング規約違反の量が影響することを排除する目的で，編集量で正規化した．しかし本研究の提案はコードの編集量による不具合混入予測の手法を否定するものではない．そして，不具合混入予測に利用できる種々のメトリクスと同時にコーディング規約違反に基づくメトリクスを利用することで，より精度の高い不具合予測を実施できると考える．本研究では，コーディング規約違反に関する情報の利用により不具合混入を説明できる可能性があることを確認する目的で，コード編集量による影響を正規化により排除した．実際の不具合予測に本研究で提案したメトリクスを利用する際には，このことが評価実験に対する構成概念妥当性に影響を与えていた可能性があることを考慮する必要がある．

また，予備実験でも提案手法の実装でも Checkstyle の検査実施の際に，コーディング規約が属性値を必要とする場合は，Checkstyle で推奨されるデフォルト値を利用した．属性値によってはガイドラインで推奨値が異なる．ガイドラインの推奨値ではなく Checkstyle のデフォルト値を利用して検査を実施したことも，構成概念妥当性に影響を与

えた可能性がある．

さらに提案手法の実装の際は，予備実験で利用した 30 プロジェクトから 16 規約すべてに対して有意水準 1%以内で帰無仮説を棄却した 14 プロジェクトを選択し，それらによって学習に利用する教師データを作成した．この際，不具合の混入したコミットと不具合の混入していないコミットでデータ数が不均衡である場合は，多数派のクラスへのバイアスがかかり，予測モデルの学習精度を下げてしまう可能性がある．不具合予測に関するデータの不均衡性について調査した先行研究 [26] と同様に不均衡の程度（先行研究 [26] での Imbalance Ratio：IR，値が 1 のときに均衡）を算出すると，14 プロジェクトでの中央値は 1.35，最小値は 0.77，最大値は 2.80 であった．先行研究 [26] では IR が 10 未満であれば不均衡のレベルが低いとしており，学習に対して極端に不均衡なデータではなかった．しかし，不具合予測においてはデータのリサンプリングにより精度を上げることが可能であると述べた研究結果もある [27]．提案手法に対してもリサンプリング技術を用いることで，より不均衡性をなくすことができ予測精度が向上する可能性がある．換言すれば，データの不均衡性に厳密に対応していないことも，構成概念妥当性に影響を与えた可能性がある．

7. 結論

本研究では，対象プロジェクトの特徴に影響を受けにくく，不具合予測に利用できるソフトウェアメトリクスとして，正規化コーディング規約違反メトリクスを定義した．そして OSS を対象にした予備実験の結果，Checkstyle で検査可能なコーディング規約のうちの 57 種類の規約に対する規約違反に，不具合混入と有意な関連があることを明らかにした．この結果を受け，OSS プロジェクトの正規化コーディング規約違反メトリクスと不具合混入との関連を教師データとして利用する，不具合予測手法を提案した．

先行研究で提案される手法の一部ではプロジェクトに固有の特徴を用いて予測を実施する．したがって，教師データの利用有無にかかわらず学習データとして対象プロジェクトのプロジェクトデータに対する一定の履歴が必要である．提案手法では，OSS プロジェクトから事前に学習を行った分類器を利用する．対象プロジェクトのプロジェクトデータに対する履歴を必要としないものの，評価実験の結果，それらの先行研究に近い性能で不具合予測を実施できることを確認した．また，提案手法と同様に他のプロジェクトのデータを利用して不具合予測を行う先行研究に対しても，コーディング規約違反メトリクスによってそれらと同様に不具合予測を実施できることを確認できた．このことは，このメトリクスにより不具合混入を説明できることと，既存のコード行数の変化を利用した手法とあわせて利用することで不具合予測の精度を高める可能性があることを示している．そして提案手法を利用すれば，事前に学習を行った分類器をあらかじめ用意し配布することで，予測対象の変更前後のソースコード以外のいっさいの解析の必要なく，不具合予測を実施できる．

今後の課題としては，より多くのコーディング規約を利用し予測性能を高めることができるかどうかを確認することと，より多くのプロジェクトを対象に評価実験を行い，不具合予測性能を再確認することがあげられる．

謝辞 本研究の成果の一部は，科研費基盤研究（C）17K00110，2019 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] 畑 秀明，水野 修，菊野 亨：不具合予測に関するメトリクスについての研究論文の系統的レビュー，コンピュータソフトウェア，Vol.29, No.1, pp.106–117 (2012).
[2] Aversano, L., Cerulo, L. and Grosso, C.D.: Learning from bug-introducing changes to prevent fault prone code, *Proc. 9th Int'l Workshop on Principles of Software Evolution* (*IWPSE'07*), pp.19–26 (2007).
[3] Zimmermann, T., Premraj, R. and Zeller, A.: Predicting Defects for Eclipse, *Proc. 3rd Int'l Workshop on Predictor Models in Software Engineering* (*PROMISE'07*), p.9 (2007).
[4] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K., Adams, B. and Hassan, A.E.: Revisiting common bug prediction findings using effort-aware models, *Proc. 2010 IEEE Int'l Conf. Software Maintenance* (*ICSM'10*), pp.1–10 (2010).
[5] Jiang, T., Tan, L. and Kim, S.: Personalized defect prediction, *Proc. 28th IEEE/ACM Int'l Conf. Automated Software Engineering* (*ASE'13*), pp.279–289 (2013).
[6] Wang, S., Liu, T. and Tan, L.: Automatically learning semantic features for defect prediction, *Proc. 38th Int'l Conf. Software Engineering* (*ICSE'16*), pp.297–308 (2016).
[7] Zhang, F., Zheng, Q., Zou, Y. and Hassan, A.E.: Cross-Project Defect Prediction Using a Connectivity-Based Unsupervised Classifier, *Proc. 38th Int'l Conf. Software Engineering* (*ICSE'16*), pp.309–320 (2016).
[8] Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A. and Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance, *IEEE Trans. Softw. Eng.*, Vol.39, No.6, pp.757–773 (2013).
[9] Yang, X., Lo, D., Xia, X., Zhang, Y. and Sun, J.: Deep Learning for Just-in-Time Defect Prediction, *Proc. 2015 IEEE Int'l Conf. Software Quality, Reliability and Security* (*QRS '15*), pp.17–26 (2015).
[10] 近藤将成，森 啓太，水野 修，崔 銀惠：深層学習によるソースコードコミットからの不具合混入予測，情報処理学会論文誌，Vol.59, No.4, pp.1250–1261 (2018).
[11] Herbold, S., Trautsch, A. and Grabowski, J.: A Comparative Study to Benchmark Cross-Project Defect Prediction Approaches, *IEEE Trans. Softw. Eng.*, Vol.44, No.9, pp.811–833 (2018).
[12] Zimmermann, T., Nagappan, N., Gall, H.C., Giger, E. and Murphy, B.: Cross-project defect prediction: A large scale experiment on data vs. domain vs. process, *Proc. 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (*ESEC/FSE'09*), pp.91–100 (2009).

[13] 田口健介，名倉正剛，高田眞吾：ソフトウェア変更時のコーディング規約違反と不具合の共起傾向の調査，ソフトウェアエンジニアリングシンポジウム 2018 論文集，pp.200–207 (2018).
[14] Allamanis, M., Barr, E.T., Bird, C. and Sutton, C.: Learning Natural Coding Conventions, *Proc. 22nd ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering* (*FSE2014*), pp.281–293 (2014).
[15] 岩間 太，中村大賀：コーディングスタイルに基づくメトリクスを用いたソースコードからの属人性検出，日本ソフトウェア科学会 FOSE 2008 ソフトウェア工学の基礎 XV，pp.129–134 (2008).
[16] 独立行政法人情報処理推進機構：【改訂版】組込みソフトウェア開発向けコーディング作法ガイド［C 言語版］Ver.2.0，独立行政法人情報処理推進機構（IPA）技術本部ソフトウェア高信頼化センター（SEC）(2014).
[17] Boogerd, C. and Moonen, L.: Evaluating the relation between coding standard violations and faults within and across software versions, *Proc. 2009 6th IEEE Int'l Working Conf. Mining Software Repositories* (*MSR'09*), pp.41–50 (2009).
[18] Kawamoto, K. and Mizuno, O.: Predicting Fault-prone Modules Using the Length of Identifiers, *Proc. 2012 4th Int'l Workshop on Empirical Software Engineering in Practice*, pp.30–34 (2012).
[19] 阿萬裕久，天嵜聡介，佐々木隆志，川原 稔：変数名とスコープの長さ及びコメントに着目したフォールト潜在性に関する定量的調査，ソフトウェアエンジニアリングシンポジウム 2015 論文集，pp.69–76 (2015).
[20] Shen, H., Fang, J. and Zhao, J.: EFindBugs: Effective Error Ranking for FindBugs, *Proc. 2011 4th Int'l Conf. Software Testing, Verification and Validation* (*ICST 2011*), pp.299–308 (2011).
[21] Lee, S., Hong, S., Yi, J., Kim, T., Kim, C. and Yoo, S.: Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks, *Proc. 2019 12th Int'l Conf. Software Testing, Verification and Validation* (*ICST 2019*), pp.391–401 (2019).
[22] Sadowski, C., var Gogh, J., Jaspan, C., Soderberg, E. and Winter, C.: Tricorder: Building a Program Analysis Ecosystem, *Proc. 37th Int'l Conf. Software Engineering* (*ICSE'15*), pp.598–608 (2015).
[23] checkstyle: checkstyle–Checkstyle 8.22, checkstyle (online), available from ⟨https://checkstyle.sourceforge.io/⟩ (accessed 2019-07-31).
[24] Rosen, C., Grawi, B. and Shihab, E.: Commit Guru: Analytics and Risk Prediction of Software Commits, *Proc. 2015 10th Joint Meeting on Foundations of Software Engineering* (*ESEC/FSE 2015*), pp.966–969 (2015).
[25] Vasilescu, B., Serebrenik, A. and Filkov, V.: A Data Set for Social Diversity Studies of GitHub Teams, *Proc. 12th Working Conf. Mining Software Repositories* (*MSR '15*), pp.514–517 (2015).
[26] Song, Q., Guo, Y. and Shepperd, M.: A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction, *IEEE Trans. Softw. Eng.* (*Early Access*), p.16 (2018).
[27] Tan, M., Tan, L., Dara, S. and Mayeux, C.: Online Defect Prediction for Imbalanced Data, *Proc. 37th Int'l Conf. Software Engineering* (*ICSE'15*), pp.99–108 (2015).

名倉 正剛 （正会員）

1999 年慶應義塾大学理工学部卒業．2001 年同大学大学院理工学研究科修士課程修了．2006 年同博士課程単位取得退学．博士（工学）．2001～2003 年日本電気株式会社．2006～2007 年慶應義塾大学大学院理工学研究科特別研究助手．2007～2009 年奈良先端科学技術大学院大学情報科学研究科特任助教．2009～2015 年株式会社日立製作所．2015～2018 年日本大学工学部准教授．2018 年より南山大学理工学部准教授．ソフトウェア工学，システム運用管理に関する研究に従事．日本ソフトウェア科学会，電子情報通信学会各会員．

田口 健介

2017 年慶應義塾大学理工学部卒業．2019 年同大学大学院理工学研究科修士課程修了．修士（工学）．現在，ヤフー株式会社にて，ソフトウェア開発業務に従事．ソフトウェア品質に関する技術，研究に興味を持つ．

高田 眞吾 （正会員）

1990 年慶應義塾大学理工学部卒業．1992 年同大学大学院理工学研究科修士課程修了．1995 年同博士課程修了．博士（工学）．同年奈良先端科学技術大学院大学情報科学研究科助手．1999 年より慶應義塾大学理工学部情報工学科専任講師，現在，同大学教授．ソフトウェア工学，情報検索等の研究に従事．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEE CS 各会員．