類似した開発者の分類と不具合予測におけるその効果

北村 紗也加[1,a)]　近藤 将成[2,,b)]　水野 修[3,c)]

概要：ソフトウェアの不具合予測の 1 つとして，開発者個人に対する予測モデルを構築する研究が行われている．開発者個人に対して予測モデルを構築するため，予測の根拠を示しやすいなど利点がある．しかし，既存の研究では変更数の多い開発者のみにしか予測モデルを構築しておらず，変更数の少ない開発者の多いオープンソースソフトウェア（OSS）への応用には課題が残っている．本研究では，類似した開発者をまとめることで変更数が少ない開発者に対しても予測モデルを構築し，OSS へも応用が可能な手法を検討する．評価実験によって，類似した開発者をまとめることで，既存の開発者個人に対する予測モデルよりも高い予測精度が出せること，および，全ての開発者に対して 1 つの予測モデルを構築するモデルと比較して，2 つのプロジェクトで高い予測精度が出せることがわかった．また，変更数が少ない開発者は，変更が多いファイルを変更している開発者のクラスタに分類されていることがわかったため，変更しているファイルに着目することで，より良い分類が行える可能性を示した．

1. はじめに

ソフトウェアの品質保証活動の 1 つとして，不具合予測手法の研究が広く行われている [1, 6, 10, 13, 15, 22, 23, 25]．既存の多くの研究では，機械学習などの手法を用いて，ソフトウェアの開発情報から不具合予測モデルを構築し，将来的に不具合になるであろうファイルや変更などを予測している [1, 6, 10, 22, 23, 25]．この時，多くの研究では，取得可能な全ての情報から 1 つの予測モデルを構築することが行われている．一方で，コード補完の研究においては，実際のソースコードや抽象構文木のノードなどの異なる情報に対しては別のモデルを構築し，最後にその結果を統合することで予測の精度を上げる試みがされている [14]．また，あるファイルを編集したのがどの開発者であるのかという情報を用いることで，不具合予測の精度を向上させようとする研究もある [15]．これらより，開発者の情報を不具合予測モデルを構築する時に考慮し，個別のモデルを構築することで，不具合の予測精度を向上させる研究が行われている [6]．こうした手法は，学習データなどから予測の根拠を示しやすいという利点も期待できる．

Jiang ら [6] は，開発者個人に対して不具合予測モデルを構築し，既存の予測モデルよりも予測の精度が向上することを示している．しかし，この研究ではコミット数が多い上位 10 人の開発者に実験対象を絞っており，コミット数が少ない開発者に対する不具合予測は研究対象としていない．一方で，オープンソースソフトウェア（OSS）では，プルリクエストなどを利用することで気軽に開発に参加をすることができるため，コミット数が少ない開発者が多くおり，コミット数が少ない開発者に対しても予測モデルを構築することが求められる．しかし，我々が知る限り，コミット数が少ない開発者の情報を考慮した個別の不具合予測モデルに関する研究は未だ行われていない．機械学習の教師あり学習を利用した不具合予測モデルでは，モデルを構築する上で十分な量の訓練データが必要であり，コミット数が少ない場合，予測モデルを構築することが難しくなるためである．

そこで，本研究では，類似した開発者をまとめることで，コミット数が少ない開発者に対しても予測モデルを構築し，予測精度を向上させることを目指す．具体的には，教師なし学習のクラスタリング手法であるスペクトラルクラスタリングを用いて開発者を分類しておき，それぞれのクラスタで不具合予測モデルを構築する．分類するための特徴として，それぞれの開発者が行なったコミットの変更メトリクス [8] を利用した．以上の特徴を備えた不具合予測手法，すなわち本研究での提案手法を LPDP（Light-Personalized Defect Prediction），既存の開発者個人に対する不具合予測モデルを構築する手法 [6] を PDP（Personalized Defect Prediction），そして，全ての開発者データを利用して 1 つの予測モデルを構築する手法を GDP（Global Defect Prediction）

[1] 京都工芸繊維大学 大学院工芸科学研究科 情報工学専攻
[2] 京都工芸繊維大学 大学院工芸科学研究科 設計工学専攻
[3] 京都工芸繊維大学 情報工学・人間科学系
[a)] s-kitamura@se.is.kit.ac.jp
[b)] m-kondo@se.is.kit.ac.jp
[c)] o-mizuno@kit.ac.jp

と呼ぶこととする．

本研究では，LPDP を評価するために以下の 3 つの研究設問を設定する．

RQ1: 既存の開発者個人に対する不具合予測の精度は開発者全てに対する不具合予測よりも優れているか？

RQ2: 類似した開発者の分類は不具合予測の精度に寄与するか？

RQ3: 分類された開発者のクラスタにはどのような特徴があるか？

これらの研究設問に答えるため，我々は，LPDP，PDP，および GDP 間の比較評価実験，および，LPDP で作成された各クラスタにおける分類された開発者の特徴の分析を行う．比較評価実験では 6 つの Java で書かれたオープンソースソフトウェアプロジェクトを用いて，予測精度を調査する．その結果，(1) PDP は全ての開発者に対して適用すると，GDP よりも予測精度が悪くなる (2) LPDP は既存の PDP よりも不具合予測精度が高い (3) LPDP は 2 つのプロジェクトで通常の GDP よりも予測精度が良くなる (4) LPDP においては，コミット数が少ない開発者は，変更が頻繁に行われているファイルを編集しているような開発者が多いクラスタに分類される，ということが結果として得られた．

本研究の主な貢献を以下にまとめる．

(1) 開発者個人に着目した不具合予測モデルを，実際の開発に近い環境で再評価した．

(2) 類似した開発者をまとめることで，既存の開発者個人に着目した不具合予測モデルよりも予測精度を向上させ，また通常の予測モデルに近い精度が出せることを示した．

(3) 類似した開発者をまとめ，不具合予測モデルを構築し評価するための手順をまとめ，また，その結果どのような分類が行われているかを初めて分析した．

以降の本稿の構成を紹介する．第 2 節では，類似した開発者を分類して不具合予測モデルを構築した動機について述べる．第 3 節では，不具合予測手法および，開発者個人に着目した予測モデルについて関連研究を紹介する．第 4 節では，提案手法である LPDP について説明する．第 5 節では，評価実験について説明する．第 6 節では，それぞれの研究設問に対する結果をまとめる．第 7 節では，分類されたクラスタ数について考察を行う．第 8 節では，本研究の妥当性の検証を行う．第 9 節では，本研究の結論と今後の拡張について述べる．

2. 研究動機

表 1 は本研究で分析した OSS の詳細のデータである．人気があり規模が大きく，時系列を考慮した分析が行い易い OSS が選択されている．最後の 2 つの列の N は変更（コミット）数を表している．つまり，コミット数が 100 未満の開発者の割合およびその開発者のコミット数の割合をそれぞれ表している．先行研究 [6] ではコミット数が多い上位 10 人のコミットから 100 個のコミットを実験に使用していた．しかし，この表より，多くの OSS では，100 回未満のコミット数である開発者が 90%前後を占めることがわかる．また，その開発者のコミット数も 10%を超えており，場合によっては 20%を上回ることがわかる．この結果より，類似した開発者をまとめることで，コミット数が少ない開発者に対しても個別の予測モデルを構築する必要があるとして，本研究を開始した．なお，開発者の同定方法については，第 8 節にて詳しく述べる．

3. 関連研究

3.1 不具合予測手法

現在までに，様々な不具合予測手法が提案されている [1, 6, 8, 10, 11, 13, 22, 23, 25]．予測対象はファイルやパッケージ，各リリース，もしくは，各変更（例えば，コミット）など様々な粒度で存在している．例えば，Kamei らの研究 [8] では，ソフトウェアの変更をメトリクス化して不具合予測を行なっている．一方で，Zimmermann ら [25] はファイルもしくはパッケージのレベルで不具合予測を行なっている．本研究では，コミットレベルでの不具合予測を行なった．これは，コミットレベルでの不具合予測は (1) 変更を行なったすぐ後に不具合が含まれていそうかがわかるため，素早いフィードバックが行える (2) 変更を行なった直後であるため，どの開発者に修正を依頼すれば良いかがわかりやすい (3) 変更を行なった開発者の記憶が新しく，書かれているコードを再度読む手間が最小限で済む，などの利点があるためである．

予測を行う上でコミットを特徴量ベクトルとして表現しなければならない．コミットを特徴量ベクトルに変換するために，追加された行数などをメトリクスとして用いる手法と，ソースコードを用いる手法が存在する．ソースコードを用いる手法として，例えば，Mizuno らの研究 [13] では，ソースコードを学習データとし，スパムフィルタを用いて不具合予測を行なっている．本研究は，初めて開発者を分類して不具合予測を行うため，より多くの研究で議論され信頼性の高い，変更メトリクス [8] を用いた手法を採用する．

3.2 開発者個人に着目した不具合予測モデル

今までにも開発者個人に着目することで，不具合予測モデルの精度を向上させようとする研究が行われている．例えば，Ostrand ら [15] はファイルとそのファイルを変更している開発者に着目をすることで，不具合予測の精度を向上させようとしている．そして，何人の異なる開発者が，ある 1 つのファイルを変更したかの情報が不具合予測に有用であることを発見している．Jiang ら [6] は，開発者個人

表 1 対象プロジェクトの詳細.

プロジェクト	言語	変更数	不具合率	開発者数	$N < 100$ の開発者率	$N < 100$ の変更率
Bazel	Java	21,217	27.2%	567	89.6%	21.9%
Camel	Java	33,985	27.4%	622	95.7%	10.9%
Geode	Java	7,656	8.0%	158	87.3%	27.2%
Gerrit	Java	36,861	19.1%	399	92.2%	7.9%
Hadoop	Java	21,683	24.9%	299	78.3%	17.8%
OsmAnd	Java	54,386	12.9%	934	91.5%	14.3%

に対する不具合予測モデルを構築することで不具合予測の精度を向上させることができることを示している.

Jiang らの研究はコミット数が多い上位 10 人の開発者に実験対象を絞っている．一方で，第 2 節で示しているように，OSS の開発においてはコミット数が少ない開発者が多く，これらの開発者に対しても不具合予測モデルを構築できるようになることが求められている．そこで，本研究では，類似した開発者を分類することで，コミット数が少ない開発者に対しても個別の不具合予測モデルを構築し，予測精度を向上させることを目指す.

3.3 スペクトルクラスタリング

スペクトラルクラスタリングは，節点と枝を持つグラフ上で行われるクラスタリング手法である．各節点は実体（分類したい要素），各枝は実体間での類似度を表し，実体間の類似度に基づいて枝を削除しデータを分割することで，指定した個数の非連結なグラフのクラスタを生成する．この手法は，不具合予測においても使用された実績がある [24].

4. 提案手法

類似した開発者をまとめ，それぞれのクラスタに対して予測モデルを構築する LPDP の手順について説明する．LPDP は以下に示す 3 つの手順に沿って進められる.

(1) 変更メトリクスの前処理
(2) 変更メトリクスを用いた開発者の分類
(3) 予測モデルの生成

なお，ここで示す手順は，訓練，検証，テストデータの 1 つの組に対して適用されるものである．訓練，検証，テストデータの作成に関しては第 5.3 節にて紹介する.

4.1 変更メトリクスの前処理

コミットに対する不具合予測モデルで頻繁に利用される変更メトリクス [8] を利用している（**表 2**）．変更メトリクスは亀井ら [8] が行なった前処理を行なっている．具体的には，高い相関のあるメトリクスを除外，もしくは結合させることにより取り除く．ただし，得られたデータに負の数が含まれていたため，我々は対数変換ではなく，z-score を用いて，メトリクスの標準化を行なった.

表 2 変更メトリクスの説明.

分類	名称	定義
Diffusion	NS	修正されたサブシステムの数
	ND	修正されたディレクトリの数
	NF	修正されたファイルの数
	Entropy	修正されたコードの各ファイルごとの分布
Size	LA	追加されたコード行数
	LD	削除されたコード行数
	LT	変更される前のファイルのコード行数
Purpose	FIX	バグ修正の変更か否か
History	NDEV	修正に関わった開発者の人数
	AGE	最後の変更から最新の変更までの平均時間
	NUC	ユニークな変更の数
Experience	EXP	開発者の経験
	REXP	最近の開発者の経験
	SEXP	サブシステムについての開発者の経験

4.2 変更メトリクスを用いた開発者の分類

前処理を行なった変更メトリクスを用いて開発者の分類を行う．分類のために，本研究ではスペクトラルクラスタリング [21] を用いた．それぞれの開発者が各節点の実体に対応しており，その開発者間の類似度が各枝の類似度となる．全ての開発者を含んだ類似度グラフを変更メトリクスより作成し，疎な枝を削除することで，指定された個数の非連結なグラフのクラスタを作成する．そして，それぞれの連結しているグラフが類似した開発者のクラスタとなる.

スペクトラルクラスタリングを適用するために，開発者毎に 1 つの特徴量ベクトルを作成する必要がある．特徴量ベクトルとしては変更メトリクスを用いるが，その具体的な値は，開発者ごとに複数のコミットを持っているため，複数存在する．今回は，訓練データにおいて，それぞれの開発者ごとに全てのコミットをまとめ，その変更メトリクスそれぞれの中央値をその開発者の特徴量ベクトルとした．例えば，開発者 A が 10 回のコミットを行なっていた場合，前処理を行なった変更メトリクスがそれぞれのコミットに

おいて計算される．この変更メトリクスそれぞれに対し 10 個全てのコミット間における中央値を計算し，それをこの開発者の特徴量ベクトルとする．つまり，各メトリックで，10 個のコミット間の中央値を取る．

スペクトラルクラスタリングは，生成するクラスタ数がパラメータとして必要となる．今回は適切なクラスタ数を発見するために，クラスタ数を 1 から 10 まで用意して，それぞれで予測モデルを構築し，検証データを用いて評価指標を計算して，最も精度が高かったものを選択した．具体的な値については，第 7 節で考察する．

4.3 予測モデルの生成

分類されたそれぞれの開発者クラスタに対して不具合予測モデルを作成する．ただし，予測モデルを学習するために，訓練データおよび検証データに以下の制約をつける：

- 訓練データには 2 つ以上の不具合および正常なコミットのデータがそれぞれあること．
- 検証データには 1 つ以上の不具合および正常なコミットのデータがそれぞれあること．

もし，不具合もしくは正常なコミットのどちらかのコミットのデータがない場合に，モデルを評価するための指標を計算できなくなるためである．訓練データにおいては，使用した予測モデルを構築するライブラリの制約で 2 つ以上としている．[*1]

訓練ができなかった開発者，および，テストデータで初めて出てきた開発者に対しては，通常の不具合予測モデルである GDP を適用する．

予測モデルにはロジスティック回帰モデル [12] を利用した．しかし，Tantithamthavorn ら [20] によれば，ロジスティック回帰モデルなどのモデルを実装するために使用可能な Python や R などのライブラリのデフォルトのパラメータを利用すると，最適なモデルを生成できず誤った結論を導く可能性が指摘されている．そこで，我々の研究ではロジスティック回帰モデルのパラメータである正規化項 C を検証データを用いて最適化した．方法はスペクトラルクラスタリングのクラスタ数を選択する方法と同じである．$C = \{0.001, 0.01, 0.1, 1, 10, 100\}$ を調査する．また，この値はクラスタ数と共に調整されるため，クラスタ数と正規化項の値の全ての組み合わせが調査される．なお，クラスタ数の考察（第 7 節）と異なり，正規化項の最適化は今回の研究の本筋ではない予測モデルに関するパラメータであり，本論文の範囲を超えるため，具体的な数値に関する考察は行わない．

最後に作成された予測モデルをテストデータに対して適用し，評価値を計算して，予測モデルの精度を確かめる．

[*1] 使用したライブラリは `glmnet` [4] である．

5. 評価実験

提案手法の評価実験では，既存の PDP および GDP との比較を行う．また，分類されたクラスタの分析も行う．実験の主要なステップを以下にまとめる．

(1) リポジトリからコミットを取得し，Commit Guru [16] を用いて，コミットが不具合を混入したかどうかのラベル情報を基にコミットにラベル付けをする．
(2) Commit Guru から 14 個の変更メトリクスを取得する．
(3) 検証手法を用いてデータセットを複数の訓練，検証，およびテストデータに分割し，GDP モデルを作成する．
(4) それぞれの訓練，検証，およびテストデータにおいて，開発者ごとにデータを分割し，PDP モデルを作成する．
(5) 開発者をスペクトラルクラスタリングによって分類し，提案手法である LPDP モデルを作成する．
(6) 評価指標に基づいてそれぞれのモデルの予測精度を評価する．
(7) スペクトラルクラスタリングによって分類されたクラスタの特徴を分析する．

以降，それぞれの方法について詳細を説明する．

5.1 Commit Guru による分析データの準備

不具合予測手法の評価において実験データの公開性と透明性は重要である [7]．そのため，我々は Rosen らが公開している Commit Guru [8, 16] から得られるデータを利用する．Commit Guru は Git のリポジトリを指定すると変更メトリクス [8]，及び，不具合混入コミットの情報を自動で計算可能な Web アプリケーションである．

本研究では，不具合混入コミットの情報より，不具合コミットと正常なコミットのラベル付けを行う．また，計算された変更メトリクスを利用してコミットを特徴量ベクトル化する．Commit Guru が行う（1）不具合混入コミットのラベル付けと（2）メトリクスの収集方法について以下に述べる．

（1）不具合混入コミットのラベル付け：Commit Guru はコミットに対し，不具合が混入したコミットとそれ以外のコミットにラベル付けをする．不具合混入コミットは不具合を修正したコミットから推定する．Commit Guru は，まずコミットメッセージを解析し，Hindle らの研究 [5] におけるコミットを分類するためのキーワード（例えば bug や fix など）があれば，そのコミットが不具合を修正しているコミットであると決める．次に，その修正を行ったコミットの修正行を特定し，その行が追加されたコミットを不具合が混入したコミットとしてラベル付けする．他の不具合コミットを推定する一般的な手法として SZZ アルゴリズム [17] があるが，公開されている信頼性の高いライブラリなどが無く，それぞれの研究で独自の実装が試みられてい

るため，公開性という観点で SZZ アルゴリズムではなく Commit Guru を利用する．

（2）メトリクスの収集：収集されるメトリクスは 14 個の変更メトリクス（change metrics）[8] である（表 2）．これらのメトリクスを，コミットの特徴量ベクトルとして利用する．

5.2 対象とするプロジェクトデータ

本研究では，6 つのオープンソースソフトウェア（OSS）である Bazel, Camel, Geode, Gerrit, Hadoop, および OsmAnd のリポジトリを利用する．プロジェクトの詳細を表 1 に示す．

5.3 予測精度検証手法

予測精度検証手法として，訓練データとテストデータの選択によるバイアスがかからないように，交差検証やブートストラップ法などが適用されることが多い．一方で，コミットに対する不具合予測モデルを評価する際にこれらを利用すると，未来のコミットを訓練データ，過去のコミットをテストデータとして使うなど現実的ではない組み合わせが発生し，適当ではない不自然に高い予測精度を記録する可能性がある．また，不具合の発見・修正に時間がかかるため，プロジェクトの後半のデータは不具合になりにくいなど不具合予測問題固有の問題がある．そのため，時間を考慮しつつ，不具合予測固有の問題を考慮した検証手法を使用する必要がある [18].

Tan ら [18] は，時系列を考慮した稼働検証法（online chnage classification）を提案している．これは，（1）不具合混入コミットは通常発見され修正されるために 100-300 日程度の時間がかかる [9] ため，不具合混入コミットであるが正常と判断されるコミットがデータに混入する（2）単純に過去のデータを訓練および検証データに使用し，その後をテストデータとして 1 回分割するのみであると，分割の仕方によってデータ選択のバイアスがかかる恐れがある（3）訓練および検証データとテストデータ間に大きな時間差があると，プログラマが変わるなどして，同じ分布から得られたデータであると仮定することが難しくなるという 3 点の問題を解決する検証方法である．本研究ではこの検証方法に我々の改良を加えたものを利用した．

（1）の問題に対しては，**間** g を訓練および検証データとテストデータの間に取る．間は訓練および検証データの不具合を発見するための余分の時間を与え，より正確なラベルづけを可能にする．間には不具合混入コミットを修正するために要した時間の平均もしくは中央値を利用する．準備実験より，**表 3** に示す値とした．

（2）および（3）の問題に対しては，訓練・検証データおよびテストデータを複数回更新しながらサンプリングすることで解決する．具体的には訓練・検証データに用いるコミットの時間間隔（**訓練間隔** Tr）およびテストデータに用いるコミットの時間間隔（**テスト間隔** Te），そして，先に説明した間を過去から未来にずらしながらデータをサンプリングする．これにより，特定の訓練・検証データやテストデータから来るバイアスを回避する．また，複数回のサンプリングを行うことで，テスト間隔を短く取ることができ，訓練・検証データとテストデータの母集団の差異を出来うる限り小さくする．それぞれの間隔はある一定のオフセットの日数分未来に動く事になる．このオフセットを**単位** u と呼ぶ．今回は単位として 30 日を利用した．テスト間隔もこの単位と同じ 30 日とした．この単位はパラメータであるが，準備実験より，この単位の変化は不具合予測に大きな影響を与えないことを確認した．

さらに，我々は**始端間** g_s および**終端間** g_e を導入した．これはプロジェクトの始端と終端のコミットからどれだけのコミットを訓練・検証データおよびテストデータに使用しないかを表している．これらを導入した理由としては，プロジェクトの始端はプロジェクト初期固有の変更が多くなると考えられ，また，プロジェクトの終端は不具合混入コミットとして同定されていないコミットが多いためである．

表 3 に具体的なパラメータの値を示している．始端間は，そのプロジェクトにおけるコミット数が増加し減少した後の日までとした．つまり，最初のコミットが多い時期が終わった後である．ここまででプロジェクトのソースコードが安定すると仮定している．終端間を決めるために，我々はまず繰り返し回数 t および訓練間隔を計算した．これらは以下の式によって求められる：

$$p = (d_l - d_s) - m,$$

$$t = (p/2 - g)/u,$$

$$Tr = t \cdot u.$$

ここで，p は分析対象となるコミットデータの期間，d_l は最も新しいコミット日，d_s は始端間を決めるために定めた分析を開始するコミットの日，m は最低限利用する終端間の期間である “安全間隔” である．本実験では 365 を安全間隔として採用した．また，この式からわかるように，単位はテストデータの長さと等しいので，繰り返し回数はテストデータの数と等しい．最後に終端間 g_e は以下の式で決定される：

$$d_e = d_s + (2 \cdot Tr + g),$$

$$g_e = d_l - d_e.$$

ここでの訓練期間に入ったコミットが訓練・検証データとして利用される．これらのコミットは，訓練期間および間に含まれるデータを用いてラベル付けがされる．テスト

表 3　実験に使用した稼働検証法におけるパラメータ値（日数）.

プロジェクト	始端間	終端間	間	単位・テスト間隔	訓練間隔	繰り返し回数	安全間隔
Bazel	328	506	95	30	300	10	365
Camel	743	466	45	30	1,500	50	365
Geode	216	431	50	30	360	12	365
Gerrit	375	499	129	30	1,410	47	365
Hadoop	925	524	125	30	1,020	34	365
OsmAnd	1,011	395	5	30	930	31	365

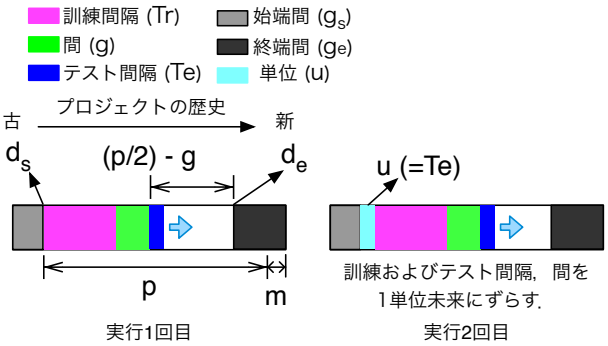

図 1　稼働検証法.

期間に入っているデータを含めてより未来にあるコミットはラベル付けには利用されない．**図 1** に稼働検証法を図示した．

5.4　予測モデルとパラメータ

第 4.2 節で示した通り，LPDP にはロジスティック回帰モデルを利用する．また，正規化項 C のパラメータを調整する．PDP および GDP においても同様の方法を取る．ただし，これらの場合には，スペクトラルクラスタリングを適用する必要がないため，最適化するパラメータはロジスティック回帰モデルの正規化項 C のみである．また，ここで作成された GDP は，提案手法の LPDP において，訓練できなかった開発者のグループおよびテストデータで初めて出てきた開発者に対して使用する GDP としても利用される．

PDP の手順は提案手法である LPDP のクラスタ数が開発者数と同じである場合と等価である．そのため，コミット数が上位の 10 人に対して適用していた Jiang ら [6] の方法とは異なり，全ての開発者に対してモデルを作成することを試みる．LPDP と同様に，モデルを構築できなかった開発者に対しては GDP を適用する．

GDP の手順は，提案手法の手順（1）の変更メトリクスの前処理までは同じである．その後，訓練データ全てを使ってロジスティック回帰モデルを作成し，検証データを用いて正規化項 C を評価する．最後に，最適な正規化項 C を使用して訓練したロジスティック回帰モデルでテストデータ上でその精度を評価する．

表 4　AUC の中央値.

	Bazel	Camel	Geode	Gerrit	Hadoop	OsmAnd	最高精度合計
LPDP	0.646	**0.664**	0.604	0.696	0.716	**0.699**	2
PDP	0.660	0.660	**0.638**	0.628	0.708	0.660	1
GDP	**0.675**	0.662	**0.638**	**0.701**	**0.719**	0.691	4

5.5　評価指標

評価指標には閾値に依存しない ROC（Receiver Operating Characteristic）曲線から得られる AUC（Area Under the Curve）を利用した．このメトリクスを全てのテストデータに対して計算し，それぞれのプロジェクトで中央値を取ったものを最終的な予測精度とする．

6.　結果

6.1　RQ1: 既存の開発者個人に対する不具合予測の精度は開発者全てに対する不具合予測よりも優れているか？

6.1.1　動機

開発者個人に対する不具合予測モデルを評価した既存研究は存在するが [6]，ここまでで述べてきたようにコミット数が上位の開発者のみに絞るなど制約が多い．これでは第 2 節で述べたように，OSS においては 90%前後の開発者に対して適用することができないことを意味する．そこで，この開発者個人に対する不具合予測モデルを全てのデータに適用し，開発者個人に対する不具合予測モデルが本当に優れているのかについて分析する．

6.1.2　アプローチ

第 5.4 節で説明した PDP および GDP を比較することによって，開発者個人に対する不具合予測モデルが通常の不具合予測モデルよりも優れているのか分析する．

6.1.3　結果

表 4 に AUC の中央値を結果として示す．太文字は 3 つの予測モデル（LPDP，PDP，GDP）の中で最も精度の高いものを指す．太文字合計はそれぞれの予測モデルでの太文字の個数である．この表から，PDP は GDP と比較して，1 つのプロジェクト以外では精度が悪くなることがわかった．また，その 1 つのプロジェクトでも値は良くなってはいない．よって，コミット数が少ない開発者に対する予測モデルの構築を考慮することが必要となる．

6.2 RQ2: 類似した開発者の分類は不具合予測の精度に寄与するか？

6.2.1 動機

RQ1 より，開発者個人に着目した不具合予測では，コミット数が少ない開発者に対する不具合予測精度の悪化により，予測精度が悪くなることがわかった．そこで，類似した開発者を分類することにより，不具合予測の精度を高めることができるのかを実験した．

6.2.2 アプローチ

第 4 節で説明した方法を我々の提案手法 LPDP として，PDP および GDP と比較し，類似した開発者をまとめることで，不具合予測精度を向上させることができるかを確かめる．

6.2.3 結果

表 4 より，類似した開発者をまとめることで，2 つのプロジェクトで，LPDP が 3 つの予測モデルの中で最も良い予測精度を出すことがわかった．また，PDP とのみ比較すると，4 つのプロジェクトで LPDP の方が予測精度が高かった．以上より，類似した開発者をまとめることは，開発者個人に着目した不具合予測モデルを構築する上で，予測精度に寄与する可能性があることがわかった．分類手法などにはまだ改善の余地があり，それらをより詳細に詰めていくことで，より良い LPDP を構築できる可能性がある．詳しくは第 9 節で述べている．

LPDP では，訓練ができなかった開発者，および，テストデータで初めて出てきた開発者に対しては GDP を適用すると述べていたが，これらのデータを除いた時の LPDP の値も 1 つ気になる点である．実際にそれらのデータに対して LPDP 適用し評価した．また，比較対象として GDP も同様のデータに対して適用した．結果は，細かい数値の違いを除けば，表 4 と変わらないものであった．つまり，Camel プロジェクトと，OsmAnd プロジェクトで LPDP が GDP より優れていた．これは予測可能である．何故ならば，LPDP では対応できないデータに対しては GDP が使われており，表 4 の差は，LPDP が適用可能なデータで発生したものであるためである．

6.3 RQ3: 分類された開発者のクラスタにはどのような特徴があるか？

6.3.1 動機

RQ2 より，開発者を分類することで，開発者に着目した不具合予測の予測精度を向上させることができる可能性を示した．一方で，どのような分類が行われているのかについては分析をしていない．同じクラスタに分類される開発者の特徴を分析することで，どの特徴が分類する際に注目されるのかがわかる．この情報より，なぜ，分類すれば予測精度が向上するのかについて調査を行う．

表 5 一時開発者クラスタの特徴.

プロジェクト	一時開発者 (%)	繰り返し (%)
Bazel	85.7	10.0
Camel	100.0	44.0
Geode	100.0	25.0
Gerrit	100.0	63.8
Hadoop	100.0	44.1
OsmAnd	100.0	51.6

6.3.2 アプローチ

ここでは 2 つの点を調査する：(1) コミット数が少ない開発者（以降，一時開発者，コミット数が 3 以下が条件）は同じクラスタに分類されるのか (2) 一時開発者が多いクラスタの特徴は何か.

(1) を調査するために，稼働検証法のそれぞれの繰り返しにおいて，一時開発者が最も分類されたクラスタ（一時開発者クラスタ）を分析する．それぞれの繰り返しにおいてこの割合（一時開発者クラスタの一時開発者数/全ての一時開発者数）をまとめ，一時開発者が同じクラスタに分類されることが多いのか，それとも，別々のクラスタに分類されることが多いのかを調べる．一時開発者のコミット数の閾値は，今回の実験では決め打ちとした．3 以下のコミットとした理由は，6 つ中 4 つのプロジェクトで半分以上の開発者を含むことができ，また，プロジェクトに定常的に寄与しているとは言い難い数字であるためである．

(2) を調査するために，一時開発者クラスタについてさらに詳しく分析する．具体的には，一時開発者クラスタと，そうではないクラスタ間で，変更メトリクスのどのメトリクスに差があるのかを調査する．一時開発者クラスタとそれ以外のクラスタの全ての変更メトリクスをそれぞれまとめて，箱ひげ図で表示し，その差を調べる．

Commit Guru はそれぞれのコミットが，ソースコードの修正コミットであるのか，特徴追加のコミットであるのかを分類することができる．そこで，どちらのコミットが多いのかを用いて，それぞれのクラスタを修正クラスタ，特徴追加クラスタ，もしくは，どちらでもない中性クラスタとして分類し，箱ひげ図と同様に比較を行なった．ここで，修正コミットおよび特徴追加コミットの個数の割合の差が 0.2 を超える場合は，このうちのどちらかに分類し，そうでない場合は中性クラスタとして分類した．これは，修正コミット数と特徴追加コミット数の差が小さい場合は，どちらのコミットも含まれているクラスタとする方が直感に合うためである．

6.3.3 結果

表 5 に示すのは，それぞれのプロジェクトでまとめた一時開発者クラスタの情報である．一時開発者クラスタにおける一時開発者数を，全ての一時開発者数に対する割合で表したもの，および，この割合の一時開発者クラスタが生成された稼働検証法における繰り返しの割合である．2 つ目の列（一時開発者）の値は，3 つ目の列（繰り返し）の割合

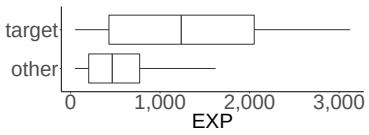

図 2 一時開発者クラスタ（target）とそれ以外のクラスタ（other）の EXP の箱ひげ図.

が最も高かった値を選択している．Bazel および Geode プロジェクトを除くと，全ての一時開発者が同じクラスタに分類された繰り返しが 50%前後であった．例えば，Camel プロジェクトでは，一時開発者が 100%同じクラスタに分類された繰り返しが 44%であった．この結果より，多くの場合は，一時開発者は同じクラスタに分類されていることがわかった．

図 2 は，Bazel プロジェクトにおいて，一時開発者と他の開発者で差が存在した変更メトリクスの 1 つである EXP（開発者の経験，表 2）の箱ひげ図である．“target” とされている箱ひげ図が，一時開発者クラスタの EXP をまとめたものであり，“other” とされている箱ひげ図がそれ以外のクラスタの EXP をまとめたものである．一時開発者のクラスタの方が高い値を示しており，他の全てのプロジェクトでも同様の結果が得られた．これは SEXP（サブシステムについての開発者の経験，表 2）でも同様の結果であった．その他のメトリクスでは，少ないプロジェクトで多少の差異が出ることはあったが，ほとんどの場合でほぼ同じ箱ひげ図となった．EXP と SEXP はそれぞれ，そのコミットでその開発者が変更をしたファイルを，以前にその開発者が変更した回数もしくは，そのサブシステムを変更した回数より計算される．このことより，一時開発者は，頻繁に修正が行われているファイルを変更している開発者が多いクラスタに分類されていると考えられる．これは OSS においては，変更数が少なく情報が少ない箇所よりも，頻繁に更新され情報が手に入りやすいファイルの方が情報が多く，プルリクエストを作成しやすいためであると考えられる．

ページ数の都合上，クラスタを修正クラスタ，特徴追加クラスタ，および，中性クラスタに分類する実験は文章のみで結果を示す．結果より，Geode プロジェクトの 1 つの繰り返しで修正クラスタに分類されたケースを除いて，一時開発者クラスタは全ての繰り返しで中性クラスタに分類された．その他の全ての修正クラスタおよび特徴追加クラスタは，一時開発者クラスタ以外のクラスタに分類された．これより，一時開発者クラスタは，多くの場合，修正も特徴追加もどちらも行う開発者が分類されるクラスタであることがわかる．

まとめると，一時開発者の分類では，どの開発者がどのファイルをどれだけ変更してきたかの情報が影響を与えていることがわかった．変更の種類は分類に大きな影響を与えていないことがわかった．

表 6 最適化されたクラスタ数.

プロジェクト	1	2	3	4	5	6	7	8	9	10	合計
Bazel	1	4	1	0	1	0	1	0	1	1	10
Camel	22	4	3	5	2	4	2	3	4	1	50
Geode	3	0	4	1	1	0	1	1	0	1	12
Gerrit	30	2	2	5	3	1	3	0	1	0	47
Hadoop	15	2	3	2	3	1	1	4	2	1	34
OsmAnd	16	4	2	1	3	1	2	0	1	1	31
合計	87	16	15	14	13	7	10	8	9	5	184

7. 考察（クラスタ数）

本論文において，スペクトラルクラスタリングによって開発者を分類することは重要な要素である．今回の実験では，開発者を最大で 10 の異なるクラスタに分類するが，このクラスタ数は検証データを用いてそれぞれの繰り返し実行で最適化している．この節では，具体的にいくつのクラスタが生成されるのかを分析する．

分析をすると，半分程度の繰り返し実行では全ての開発者が同じクラスタに分類される結果となったが，その他半分の繰り返し実行では 2 つ以上のクラスタに分類されることがわかった．**表 6** に示すのは，実際に最適化されたクラスタ数の分布である．それぞれの列が，1 から 10 までのクラスタ数に対応しており，それぞれの行が，各プロジェクトで，それぞれのクラスタ数を最適値として実行した繰り返し実行の回数を示している．

全てのプロジェクトの繰り返し実行を合計した値を見てみると，実際に半分程度の繰り返し実行がクラスタ数 1 となっていることがわかる．その後，クラスタ数が増えていくごとに，少しづつ値が減少していることも観察できる．

それぞれのプロジェクトを見ていくと，1 つのクラスタに分類されたケースが最も多いプロジェクトが多いが，その割合にばらつきがあることがわかる．例えば，Gerrit プロジェクトでは，47 回の繰り返し実行中，30 回の実行で，全ての開発者を同一のクラスタに分類している．一方で，例えば Camel プロジェクトでは，50 回の繰り返し実行中，22 回の実行で，全ての開発者を同一のクラスに分類し，それ以外のケースでは 2 つ以上のクラスタに分類している．

以上の結果より，開発者は実際に複数のクラスタに分類されるケースが半分程度存在することがわかる．また，クラスタ数は概ね小さい値に収まりそうであることがわかる．

8. 妥当性の検証

8.1 構成概念妥当性

予測精度の評価指標として，AUC を用いている．他に不具合予測でよく使用される評価指標である適合率や再現率などが存在するが， これらの指標は結果にバイアスを混入させる恐れがあると報告されている [2, 3, 19]．Tantithamthavorn ら [19] によると閾値を使用しない評価指標を使うべきであり，AUC はこの基準を満たしている．

本実験では，時系列データであるコミットを扱うため，Tan ら [18] によって提案されている手法を改良して，提案手法を評価している．交差検証などが一般的に使用される手法であるが，時系列データに対して使用することは難しく，また，ソフトウェアプロジェクト特有の性質を考慮する必要があったためである．

開発者の同定に利用したのは，Commit Guru が取得した各コミットにおける開発者名である．これは，Git におけるコマンドである `git log --pretty=format:"%an"` によって得られる名前であるため，例えば，同じ開発者が異なるメールアドレス（アカウント）でコミットをした場合に，同一の開発者であると同定することに失敗する恐れがある．今後，追加の分析などを行う必要が存在する．

8.2 外的妥当性

本実験では 6 つの OSS を選択し，それらから収集したコミットを実験データとして利用している．選択肢したプロジェクトは複数の分野を考慮しているが完全ではないため，今後，対象のプロジェクトを増やすことが必要である．また，選択されたプロジェクトは Java の OSS である．これは，今後クラス間の依存性解析などを行う際に，ツールが豊富であり分析が容易である，という点から Java のみに絞っている．しかし，これは 1 つの制約であるため，今後，多言語へも拡張していく必要がある．

8.3 内的妥当性

本研究の実験データへのラベル付けおよび変更メトリクスの計算には，Commit Guru が利用されている [16]．一方で，Commit Guru のラベル付けの精度は完全ではない．例えば，Commit Guru の論文で挙げられている特定のキーワードを含まない不具合修正コミットは見逃してしまう．一方で，ソースコードが公開されており，また，Web アプリケーションが利用可能であるという点から実験の再現性が高く，今後の応用も容易であると考える．

実験に利用したスクリプトに対しては検証を行なっているが，我々が気づいていない不具合が含まれている可能性は存在する．そのため，再現スクリプトとして実装を公開し[*2]，容易に実験の再現およびその正しさの検証を行えるようにしている．

9. 結論

類似する開発者を分類することによって，開発者個人に対する不具合予測の精度を高めることができるかを実験した．結果として，既存の開発者個人に対する不具合予測モデルより，予測精度を向上させることができた．また，通常の全てのデータを使った予測モデルと比較すると，2 つのプロジェクトで予測精度がよくなった．

本研究は開発者を分類することで不具合予測の精度を向上させることができるかを試した初めての論文である．そのため，まだ多くの改善を行える余地があり，今後引き続き研究を行なっていく．具体的には（1）変更メトリクスのみならず，変更されたソースコードをベクトル化したものをコミットの特徴量ベクトルとして追加する（2）開発者の分類の際に，今までにその開発者がどのファイルを主に変更してきたのか，また，どれだけの期間このプロジェクトに関わっているのかの情報を追加する（3）中央値を使って開発者の代表値を計算しているが，代表値を使わない手法，制約付きボルツマンマシンなどの内部状態を学習するなど他の手法を使うとどうなるのか分析する（4）開発者の分類にスペクトラルクラスタリングのみならず，他の分類手法を用いるとどうなるのかを分析する，などがあげられる．

また，RQ3 より，ファイルがどの開発者によってどれだけ変更されたかという情報が，コミット数が少ない開発者を分類する 1 つの指標となることがわかった．この情報を元に，より適切なメトリクスの構築および選択などを行う必要がある．

謝辞 本研究は JSPS 科研費，JP19J23477 の助成を受けたものです．

[*2] https://github.com/MKmknd/SES2019_replication

参考文献

[1] Aversano, L., Cerulo, L. and Del Grosso, C.: Learning from bug-introducing changes to prevent fault prone code, *in proceedings of the 9th International Workshop on Principles of Software Evolution (IWPSE)*, ACM, pp. 19–26 (2007).

[2] Bowes, D., Hall, T. and Gray, D.: Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix, *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, ACM, pp. 109–118 (2012).

[3] Chicco, D.: Ten quick tips for machine learning in computational biology, *BioData mining*, Vol. 10, No. 1, p. 35 (2017).

[4] Friedman, J., Hastie, T., Tibshirani, R., Simon, N., Narasimhan, B. and Qian, J.: glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models (2019).

[5] Hindle, A., German, D. M. and Holt, R.: What do large commits tell us?: a taxonomical study of large commits, *in proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR)*, ACM, pp. 99–108 (2008).

[6] Jiang, T., Tan, L. and Kim, S.: Personalized defect prediction, *in proceeding of the 28th International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 279–289 (2013).

[7] Kamei, Y. and Shihab, E.: Defect prediction: Accomplishments and future challenges, *in proceeding of the 23th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5, IEEE, pp. 33–45 (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 Transactions on Software Engineering*, Vol. 39, No. 6, pp. 757–773 (2013).

[9] Kim, S. and Whitehead Jr, E. J.: How long did it take to fix bugs?, *Proceedings of the 2006 international workshop on Mining software repositories (MSR)*, ACM, pp. 173–174 (2006).

[10] Kim, S., Whitehead Jr, E. J. and Zhang, Y.: Classifying software changes: Clean or buggy?, *IEEE Transactions on Software Engineering*, Vol. 34, No. 2, pp. 181–196 (2008).

[11] Li, J., He, P., Zhu, J. and Lyu, M. R.: Software Defect Prediction via Convolutional Neural Network, *Proceedings of the 2017 Software Quality, Reliability and Security (QRS)*, IEEE, pp. 318–328 (2017).

[12] McDonald, J. H.: *Handbook of Biological Statistics (3rd ed.)*., Sparky House Publishing, Baltimore, Maryland. (2014).

[13] Mizuno, O. and Kikuno, T.: Training on errors experiment to detect fault-prone software modules by spam filter, *in proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ACM, pp. 405–414 (2007).

[14] Nguyen, A. T., Nguyen, T. D., Phan, H. D. and Nguyen, T. N.: A deep neural network language model with contexts for source code, *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, IEEE, pp. 323–334 (2018).

[15] Ostrand, T. J., Weyuker, E. J. and Bell, R. M.: Programmer-based fault prediction, *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ACM, p. 19 (2010).

[16] Rosen, C., Grawi, B. and Shihab, E.: Commit Guru: Analytics and Risk Prediction of Software Commits, *in proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, ACM, pp. 966–969 (2015).

[17] Śliwerski, J., Zimmermann, T. and Zeller, A.: When do changes induce fixes?, *Sigsoft Software Engineering Notes*, Vol. 30, No. 4, ACM, pp. 1–5 (2005).

[18] Tan, M., Tan, L., Dara, S. and Mayeux, C.: Online defect prediction for imbalanced data, *in proceedings of the 37th International Conference on Software Engineering*, IEEE, pp. 99–108 (2015).

[19] Tantithamthavorn, C. and Hassan, A. E.: An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges, Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP'18), p. To Appear (2018).

[20] Tantithamthavorn, C., McIntosh, S., Hassan, A. E. and Matsumoto, K.: Automated parameter optimization of classification techniques for defect prediction models, *Proceedings of the 38th International Conference on Software Engineering*, ACM, pp. 321–332 (2016).

[21] Von Luxburg, U.: A tutorial on spectral clustering, *Statistics and computing*, Vol. 17, No. 4, pp. 395–416 (2007).

[22] Wang, S., Liu, T. and Tan, L.: Automatically learning semantic features for defect prediction, *in proceedings of the 38th International Conference on Software Engineering*, ACM, pp. 297–308 (2016).

[23] Yang, X., Lo, D., Xia, X., Zhang, Y. and Sun, J.: Deep learning for just-in-time defect prediction, *in proceedings of the 2015 Software Quality, Reliability and Security (QRS), IEEE*, IEEE, pp. 17–26 (2015).

[24] Zhang, F., Zheng, Q., Zou, Y. and Hassan, A. E.: Cross-project defect prediction using a connectivity-based unsupervised classifier, *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, ACM, pp. 309–320 (2016).

[25] Zimmermann, T., Premraj, R. and Zeller, A.: Predicting defects for eclipse, *in proceedings of the 3th International Workshop on Predictor Models in Software Engineering*, IEEE, p. 9 (2007).