Vol.2019-SE-201 No.16
2019/3/8

ソースコード修正履歴が自動バグ修正の結果に与える影響の分析

首藤 巧[1,a)]　亀井 靖高[1,b)]　鵜林 尚靖[1,c)]　佐藤 亮介[1,d)]

概要：自動バグ修正において，版管理システムのソースコード修正履歴を機械学習手法によって学習し修正に利用することで，修正結果を開発者の行うような修正に近付けるために手法がある．同一の修正対象のバグであっても学習したソースコード修正履歴によって修正結果は異なってくることが予想されるが，どの程度影響があるのか実証的な実験はなされていない．本研究では，学習に用いるソースコード修正履歴がバグの修正結果にもたらす影響の調査を次の二つの Research Questions (RQ) を基に調査を行った．修正対象のバグと学習データの OSS の関係が修正結果にどのような影響を与えるのかという RQ について調査した結果，学習データセットに利用する修正履歴を修正対象と同一の OSS から収集することによって，テストケースに通過するだけの無意味な修正パッチの生成数を減らし，正しい修正パッチに割り当てる優先順位を向上させた．学習データセットの修正パターンが修正結果に影響を及ぼすのかという RQ について調査を行った結果，Prophet により抽出される学習データセットの特徴を本調査における修正パターンの空間の違いと，テストケースの品質による修正結果への影響のため，学習データセットの修正パターンは修正結果に影響を及ぼさなかった．

キーワード：自動バグ修正

1. はじめに

ソフトウェア開発においてバグ修正は重要であり，多くのコストが費やされる．Gazzola らによると，デバッグ作業がソフトウェア製品の開発コスト全体の約 50 %を占めることが多いと報告されている [1]．通常バグの修正は一部ツールを利用しながらも開発者自身が手動で行っており，このバグ修正の作業を自動化することによって開発者の負担を大きく軽減することができる．

現在提案されている自動バグ修正手法は，バグを含むソースコードとテストスイートを元にバグの修正パッチ（修正パッチはプログラムのソースコードとそれに対する編集操作）を生成する．これらの手法ではバグの修正結果はテストケースに依存する可能性が高く，テストケースには全て通るものの，開発者に受け入れられるような修正とはかけ離れた無意味な修正パッチを生成する可能性がある．

自動バグ修正のテストケースへの依存による無意味な修正パッチを生成する問題を解決するため，Long らは版管理システムにある過去のバグ修正履歴から機械学習によって確率モデルを生成し，パッチ生成に利用する自動バグ修正手法である Prophet を提案した [2]．この手法では開発者が行うようなバグの修正パッチを生成することによって正しいパッチが生成できるという考えのもと，過去の開発者のパッチと類似度の高いパッチを優先的に生成する設計になっている．

このように開発者の行った修正履歴を自動バグ修正に利用する手法では，学習に利用したデータセットによって修正結果は異なってくるものと考えられる．よって本研究では版管理システムのバグ修正履歴を利用した自動バグ修正手法の修正結果への学習データセットの影響を明らかにするために，以下の 2 つ Research Question (RQ) を基に調査を行った．

RQ1：修正対象のバグと学習データの OSS の関係が修正結果にどのような影響を与えるのか

RQ2：学習データセットの修正パターンが修正結果に影響を及ぼすのか

以降，2 章では本研究の背景について説明する．3 章では調査において使用するデータセットについて説明をする．

[1] 九州大学
[a)] shuto@posl.ait.kyushu-u.ac.jp
[b)] kamei@ait.kyushu-u.ac.jp
[c)] ubayashi@ait.kyushu-u.ac.jp
[d)] sato@ait.kyushu-u.ac.jp

ⓒ 2019 Information Processing Society of Japan

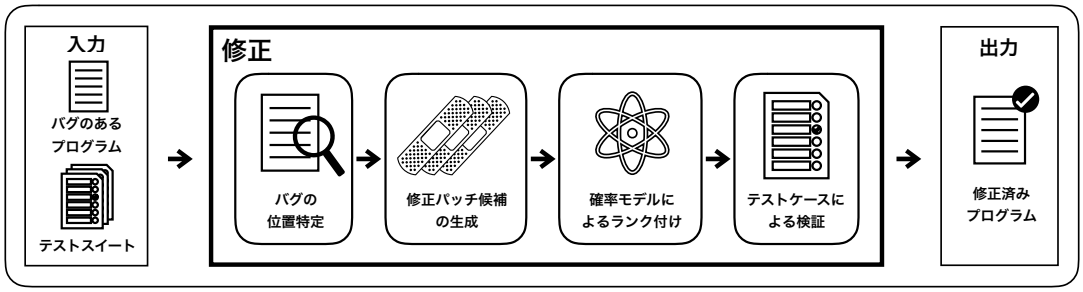


図 1　Prophet の修正過程

4 章では RQ についてのアプローチと実験結果について説明をする．5 章では妥当性への脅威について述べ，6 章で結論と今後の課題を述べる．

2. 研究背景

2.1 自動バグ修正

近年ソフトウェア工学において自動バグ修正の分野は盛んに研究が行われている．自動バグ修正の実現によってソフトウェア開発におけるデバッグ作業が大幅に減ることが期待できるため様々な手法が研究者によって提案されている．

自動バグ修正手法における代表的な手法として GenProg[3] がある. この手法はバグを含むプログラムとそのバグを検出することを目的としたテストケースの集合体であるテストスイートを入力とし，バグを修正したプログラムを出力とする．GenProg は欠陥位置特定アルゴリズムによって推定された欠陥位置に対して遺伝的プログラミングに基づいてコードの変更を行なっていく．最終的に入力として与えられたテストケースに全て通過したプログラムを修正済みのプログラムとして GenProg は出力する．しかし多くの開発者はテストケースには記載されていないプログラムの動作も想定した上でバグの修正を行う [4]．したがって開発者の行うような修正プログラムが生成されない場合が多い．このような問題を解決するために実際に開発者が行った修正を自動バグ修正に利用する Prophet と呼ばれる手法を Long らは提案した．

2.2 Prophet

Prophet は過去の開発者のバグ修正履歴から機械学習により確率モデルを生成してプログラムの修復に利用する自動バグ修正手法である．Prophet はバグを含む実行可能な C 言語で記述されたプログラム及びバグの検出を目的としたテストケースの集合であるテストスイートを入力として受け取り，プログラムの自動修正を行う．Prophet の修正過程について図 1 に示した．出力はテストスイートに含まれるテストスイートを全て通過したプログラムである．Prophet が修正を行う際に必要とする確率モデルを生成するための機械学習と実際の修正を行う段階について次節から詳しく説明をする．

2.2.1 機械学習

学習データセット：Prophet は Git や Subversion といったソースコードの版権管理システムから開発者が OSS ソフトウェアに行った修正を学習データセットとして利用する．この際，学習する修正パッチは Prophet の修正空間に含まれている必要がある．

特徴抽出：収集したそれぞれの修正パッチについて Prophet は特徴の抽出を行う．特徴は大きく分けて二つあり，プログラムの値の特徴と変更の特徴である．プログラムの値の特徴とはプログラムの修正前と修正後で変数や定数がどのように使用されているかをまとめたものであり，変更の特徴とは修正の際に行われた変更の種類とその修正が行われた箇所の付近のステートメントの関係性をまとめたものである．Prophet はこれらの特徴を約 3500 次元の一つの特徴ベクトルに変換し学習に利用する．

機械学習：Prophet は特徴抽出により得られた特徴ベクトルから機械学習によって確率モデルを生成する．確率モデルは Prophet の修正空間に含まれる修正パッチ候補に対し確率を割り当てる．この確率はその修正パッチ候補が正しい可能性，つまり開発者が行うような修正である可能性を示す．

2.2.2 修正パッチ生成

欠陥位置特定：Prophet の欠陥位置特定アルゴリズムはテストケースを入力したプログラムの実行トレースの分析を行い，欠陥位置を優先度付きのリストとして出力する．バグの検出を目的としたネガティブテストケースを入力として与えた場合の実行頻度が高く，要件を満たしているかどうかをチェックするポジティブテストケースを入力として与えた場合の実行頻度が低いようなプログラムの文の欠陥位置としての優先度を高く設定する．

修正パッチ候補生成と特徴抽出：欠陥位置特定アルゴリズムによって算出された位置に対して Prophet は修正空

間内の修正操作を全て適用してパッチ候補の生成を行う．Prophet の修正空間は Long らの先行研究である SPR[5] と呼ばれる自動バグ修正手法と同様のものである，Prophet は各修正パッチ候補に対して，学習対象の修正パッチに行ったものと同様の特徴抽出を行う．

修正パッチ候補の順位付けと検証：Prophet は学習済みの確率モデルと各修正パッチ候補の特徴ベクトルから確率スコアを算出する．Prophe はそれらのスコアに基づいて修正パッチ候補をソートし，スコアの高い修正パッチ候補からテストケースを実行する．全てのテストケースを通過した修正パッチ候補を修正パッチとして出力する．

2.3 本研究の着眼点

Long らが行った Prophet の評価実験では SPR や GenProg といった他の自動バグ修正手法の評価にも用いられたベンチマークに対して修正を行った．SPR は Prophet のベースとなった自動バグ修正手法であるが，Prophet がバグ修正履歴を学習することによって SPR よりも良い修正結果を残した．

しかしこの評価実験で収集されたバグ修正履歴は，リポジトリの総コミット数や Prophet の探索空間に含まれているか等といった条件に適合しているものを集めたに過ぎず，収集元であるアプリケーションや各バグ修正の特徴などは考慮されていない．そこで本研究では学習データセットと修正結果に着目した研究を行う．

2.4 関連研究

Yang らは自動バグ修正においてテストケースにオーバーフィットした修正パッチが生成される問題を解決するためにそのような修正パッチを検出し除外するフレームワークを提案した [6]．結果として GenProg や SPR が使用したベンチマークの 45 個のバグにおいて 75.2%のオーバーフィットパッチを除外した．

Kong らは様々な自動バグ修正手法に関してテストスイート，修正対象のプログラムが自動バグ修正の効果や効率にどのように影響するのかについて，異なるテストスイートとバグのあるプログラムにそれぞれの自動バグ修正手法を 3000 回以上適用することによって調査した [7]．結果として小規模のプログラムで有効に働く自動バグ修正手法は中規模のプログラムに適用すると多大なコストがかかることや修正が有効に働かなくなることが判明した．また，テストの通過率と，修正の成功率や時間効率の関係についても明らかにした．

本研究では，自動バグ修正手法の性能を調査するために，データセットとしてテストケースではなく学習データについての調査を行ったという点で新規性がある．

表 1 学習データセット全体

プロジェクト	学習に用いられた修正履歴
libtiff	11
apr	12
curl	53
httpd	75
wireshark	85
python	114
php	187
subversion	240

3. データセット

本章では修正対象であるベンチマークのバグデータセットと，学習に利用した修正パッチのデータセットについて述べる．

3.1 学習データ

Long らの Prophet の提案論文での評価実験に用いられた学習データセットを利用する．表 1 に示す 8 つの OSS プロジェクトから構成されており，GitHub 上に存在する C 言語のプロジェクトである．OSS に着目した実験を行うにあたって十分な種類の OSS プロジェクトから構成されているため本研究ではこの学習データセットを利用した．

3.2 ベンチマーク

Long らの Prophet の評価実験や SPR, Kali[8], GenProg, AE[9] の評価にも利用された 8 つの OSS プロジェクトのバグから構成されるベンチマークのデータセットの内，本研究で利用する学習データである表 1 に含まれている OSS プロジェクトのバグをベンチマークとした．合計 85 件のソースコード修正に関するデータセットによって構成されており，ベンチマーク中のいずれのデータセットもネガティブテストケースとポジティブテストケースを含んでいる．

4. (RQ1) 修正対象のバグと学習データの OSS の関係が修正結果にどのような影響を与えるのか

4.1 動機

ベンチマークのデータセットにはプログラミング言語や画像操作ライブラリなど様々な OSS プロジェクトが含まれている．これらのベンチマークの OSS プロジェクトと学習データセットに利用する OSS プロジェクトの関係性がバグの修正結果に与える影響が判明すれば学習データセットを収集する際の指針として役立つと考えられる．例として，修正対象と同一のドメインの OSS による学習が有効的であると判明した場合，プログラミング言語の OSS プロジェクトのバグを修正する際にはプログラミング言語の OSS プロジェクトから収集した修正履歴によって学習デー

```
    le->refcount++;
+   if (!(1))
        (*stream)->rsrc_id =
            zend_register_resource();
```

図 2　Prophet が生成した修正パッチ

```
+   if (index == -1) {
        le->refcount++;
        (*stream)->rsrc_id =
            zend_register_resouce();
+   } else {
+       regentry->refcount++;
+       (*stream)->rsrc_id = index;
+   }
```

図 3　開発者が実際に行った修正

タセットを構築する方が良い，といった場合が挙げられる．

本研究ではバグを修正する際に利用する確率モデルを生成するための学習データセットを，修正対象と同一の OSS から収集した修正履歴と異なる OSS から収集した修正履歴の二種類を用意し実験結果を比較する．

4.2　アプローチ

本調査では libtiff，wireshark，python，php の四つの OSS プロジェクトの修正を行う際にそれぞれ二種類ずつの学習データセットを用意し実験を行う．

一種類目は表 1 の OSS の内，修正対象の OSS と同一の OSS の修正パッチで構成されたデータセット（以後 Within データセットと呼称する）である．もう一方のデータセットは学習データセット全体の内，Within データセットの OSS 以外の OSS 全てから構成されるデータセット（以後 Cross データセットと呼称する）である．例として libtiff のバグを修正する際に用意する学習データセットは libtiff から収集した修正パッチのみから構成される Within データセットと，apr, curl, httpd, wireshark, python, php, subversion の 7 つの OSS から収集された修正パッチから構成される Cross データセットの二つの学習データセットである．これらの二種類のデータセットによって修正パッチを生成できたベンチマークのバグの件数について比較する．

4.3　実験結果

表 2 には各 OSS ごとに修正パッチを生成できた件数を示した．python 及び wireshark については両データセットも修正パッチ生成数は変わらず, php と libtiff については Within データセットよりも Cross データセットの方が修正パッチ生成数が多いという結果になった．Cross データセットで修正パッチを生成し，Within データセットでは修正パッチを生成できなかった 4 件について，Cross データセットが生成した修正パッチについて詳しく調査を行った．それら 4 件の内，php の 2 件と libtiff の 1 件については Cross データセットによって生成された修正パッチは文の条件式を恒偽にするなど，開発者の修正履歴と比較しても明らかに無意味な修正を行っていることが判明した．今

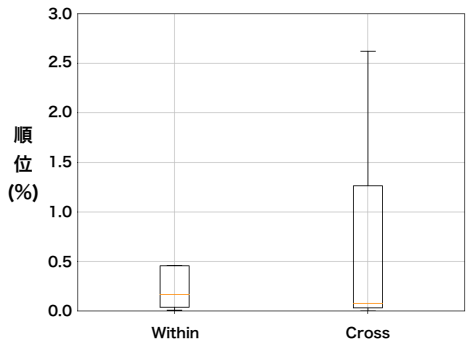


図 4　正しいパッチに割り当てられた優先順位の分布

回確認された無意味な修正の一つを図 2 に示す．また，図 3 は同じバグに対して開発者が行った修正である．

また，2.2.2 項で示した，確率モデルが修正パッチ候補に割り当てる優先順位に着目した実験も行った．表 3 には本研究のベンチマークのうち Long らによって Prophet が正しい修正パッチを生成できることが確認されているバグ 16 件について，全ての修正パッチ候補の中での正しい修正パッチの順位を各データセットごとに示したものである．また，順位百分率を図 4.3 に箱ひげ図としてプロットしたところ，Within データセットにより割り当てられた順位の方が全体として高い順位に分布していた．

よって二つのデータセットによって修正パッチが生成されたバグの数は実質的に同等だが，正しい修正パッチに割り当てた優先順位は Within データセットの方が高かった為，Within データセットにより生成された確率モデルの方が良い性能を示したと言える．

> 学習データセットに利用する修正履歴を修正対象と同一の OSS から収集することによって，テストケースに通過するだけの無意味な修正パッチの生成数を減らし，正しい修正パッチに割り当てる優先順位を向上させた．

表 2　RQ1 実験結果

プロジェクト	LOC	テスト数	件数	概要	修正パッチを生成できたバグの件数	
					Within データセット	Cross データセット
php	1,046,000	8,471	44	プログラム言語 (Web)	14	17
libtiff	77,000	78	23	画像操作ライブラリ	10	11
python	407,000	355	11	汎用プログラム言語	6	6
wireshark	2,814,000	63	7	ネットワークパケットアナライザ	4	4

表 3　学習データセットによる正解パッチの順位比較

バグ	全パッチ候補数	正解パッチ順位/順位百分率 (%)	
		Within データセット	Cross データセット
php-307562-307561	22,881	104/0.455	600/2.622
php-307846-307853	17,610	918/5.213	6,936/39.387
php-307914-307915	30,838	8/0.026	1/0.003
php-308262-308315	70,879	135/0.190	55/0.078
php-308734-308761	10,645	1,221/11.470	1,415/13.293
php-309111-309159	41,554	59/0.142	338/0.813
php-309516-309535	21,016	1,446/6.880	6,050/28.788
php-309579-309580	40,152	3/0.007	11/0.027
php-309688-309716	45,647	54/0.118	44/0.096
php-309892-309910	27,437	54/0.197	14/0.051
php-310011-310050	52,498	242/0.461	18/0.034
php-310991-310999	69,387	5/0.007	26/0.037
php-311346-311348	5,099	14/0.275	14/0.275
libtiff-ee2ce5b7-b5691a5a	67,2413	29/0.04	13/0.019
libtiff-d13be72c-ccadf48a	71,876	75/0.104	54/0.075
libtiff-5b02179-3dfb33b	157,153	13/0.008	40/0.025

5. (RQ2) 学習データセットの修正パターンが修正結果に影響を及ぼすのか

5.1 動機

学習データセットにおける修正パターンの特徴と修正結果の間に因果関係があれば，学習データセット作成の為の指針になることが考えられる．本研究では修正対象のバグと学習データセットの修正パターンの関係が自動バグ修正に及ぼす影響を明らかにするために，データセットの修正操作のパターンを分類し，データセットの特徴の傾向を分析する．

5.2 アプローチ

Sobreira らは Java の OSS 修正パッチとテストスイートを収集したデータセットである Defects4j[10] に含まれる修正操作を，修正前と修正後の差分を元に頻出する修正パターンに基づいて分類を行った [11]．その修正パターンによる分類を本研究のデータセットにも適用する．

表 4 はその分類の一覧である．表 4 の一列目は修正パターンを大きく五つに分類したものである．二列目は一列目に示した修正パターンをより詳細に分類したものである．これらの修正パターンに基づいて本研究で用いるデータセットを目視により分類する．

Sobreira の分類した詳細修正パターンの内，本研究のデータセットで分類されなかった項目については表 4 から除外している．また，本研究のデータセットが C 言語プログラムであり例外処理に関する構文が言語レベルでは存在しないため，目視により例外送出に関する処理かどうかを判断した．学習データセットについては，各データセットごとに無作為に 30 個ずつの修正パッチを選択し目視分類を行った．30 に満たない OSS については全ての修正パッチについて目視調査を行った．

5.3 実験結果

図 5，図 6 と図 7 は表 4 の一列目の修正パターンに基づいた分類結果のレーダーチャートである．

また，各データセット間の修正パターンの特徴の隔たりを示すためにデータセット間でユークリッド距離を算出した．表 5 はベンチマークのバグの修正パターンの分類結果から算出された特徴ベクトルと，学習データセットの修正パターンから算出された特徴ベクトルの間のユークリッド距離を示したものである．特徴ベクトルは各データセットにおける修正パターンの割合を五次元の単位ベクトルとしたものである．

修正パターンの分布が RQ1 の修正結果に影響すると仮定した場合，直観的にはベンチマークと修正パターンに隔たりがないデータセットの方が修正パッチの生成件数は多く

表 4　修正パターン

修正パターン名	詳細修正パターン名	説明
Conditional Block	Conditional block addition (CB-A)	if ブロックの追加
	Addition with return statement (CB-AR)	return 文を含んだ if ブロックの追加
	Addition with exception throwing (CB-AE)	例外送出を含んだ if ブロックの追加
	Conditional block removal (CB-R)	if ブロックの削除
Expression Fix	Logic expression expansion (EF-LE)	論理式の拡張
	Logic expression reduction (EF-LR)	論理式の削減
	Logic expression modification (EF-LM)	論理式の変更
	Arithmetic expression modification (EF-AM)	算術式の変更
Wraps-with/Unwraps-from	Wraps-with if statement (WW-I)	if 文により囲む
	Wraps-with if-else statement (WW-IE)	if-else 文により囲む
	Wraps-with else statement (WW-E)	else 文により囲む
	Wraps-with loop (WW-L)	for 文や while 文などのループ分により囲む
	Unwraps-from if-else statement (WW-IE)	if-else 文による囲みの削除
Wrong Reference	Variable (WR-V)	変数への参照の変更
	Method (WR-M)	関数の呼び出しの変更
Other	Constant Change (O-CC)	定数の変更
	Not classfied (O-N)	未分類

表 5　ベンチマークと学習データセットの修正パターンのユークリッド距離

OSS プロジェクト	ベンチマークとの間の修正パターンのユークリッド距離	
	Within データセット	Cross データセット
php	0.730	0.758
libtiff	0.489	0.437
python	0.451	0.655
wireshark	0.386	0.667

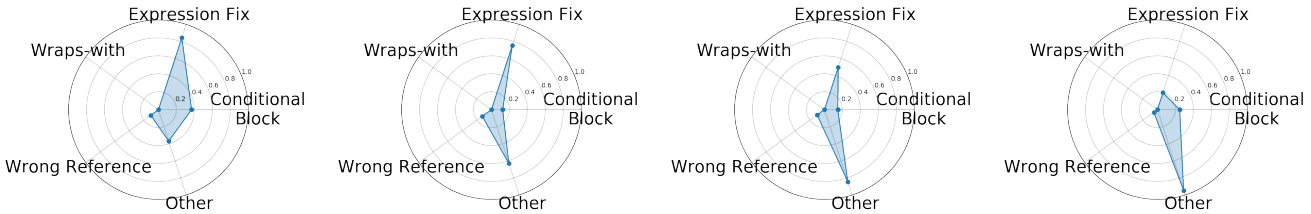
なると考えられる．しかしながら，表 5 より，Within データセットの方がベンチマークとの隔たりが少ない python と wireshark ついては表 2 の示すように修正件数に差はなかった．また，各データセットで同程度ベンチマークの修正パターンと隔たりがある php と libtiff については修正パッチの生成件数に差が発生している．よって本調査の結果と RQ1 の結果の間に関連が見られなかった要因について追加で調査を行った．

まず Cross データセットによって生成されたが，Within データセットでは生成されなかった php と libtiff のバグの修正パッチ計四件について詳しく調査したところ，php の三件中二件と wireshark の一件が Conditional Block の修正パターンに当てはまる修正操作であった．図 5 と図 6 から分かるように php に関しては Cross データセットよりも Within データセットの方が Conditional Block に関する修正パターンの割合が多く，libtiff については Conditional Block について有意な差は見られない．

Cross データセットのみで生成された修正パッチと学習データセットの修正パターンの特徴が一致していないことについて考察を行う．本調査における修正パターンの分類は修正前後におけるプログラムの差分から読み取れる変更操作の分類であり，2.2.1 項で示した Prophet の特徴抽出において修正前後における変更操作以外で注目される周辺のプログラムとの関係や変数のプログラム中での扱われ方などについては分類の対象外である．よって学習データセットの修正履歴における変更操作以外の修正履歴の特徴がバグ修正履歴に大きく影響を与えたため，php と libtiff について RQ1 での修正パッチの生成件数と本調査の修正パターンの分類結果に関連が見られなかったと考える．

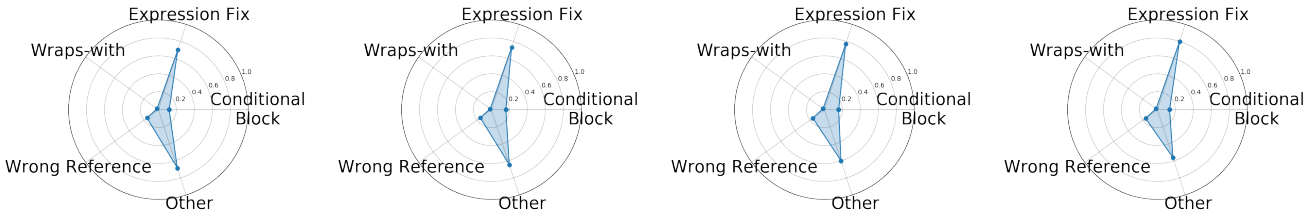
また，本研究の二つのデータセットによって生成された python と wireshark のバグの修正パッチについて詳しく調査したところ，python と wireshark 共に一つのバグにつき平均約 6 個の修正パッチが生成されていた．テストケースの品質の低さによって Prophet の修正空間内にテストケース全てに通過する修正パッチ候補が多数存在していたため，異なる学習データセットを用いた場合でも必ず修正パッチが生成されてしまい，結果に差が出なかったと考えられる．

学習データセットの修正パターンは修正結果に影響を及ぼさなかった．原因として，Prophet により抽出される学習データセットの特徴を本調査における修正パターンの分類では表すことが困難である点と，テストケースの品質による修正結果への影響が大きいという点が考えられる．



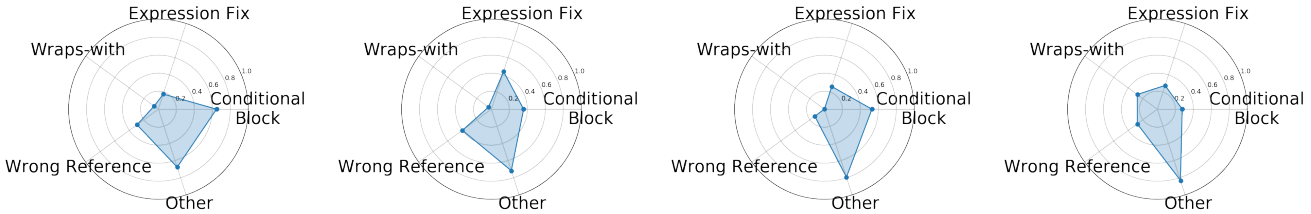
(a) php (b) libtiff (c) python (d) wireshark

図 5 Within データセットの修正パターン



(a) php (b) libtiff (c) python (d) wireshark

図 6 Cross データセットの修正パターン



(a) php (b) libtiff (c) python (d) wireshark

図 7 ベンチマークのバグの修正パターン

6. 妥当性への脅威

RQ1 の調査環境における脅威として修正結果がテストケースに品質の影響を大きく受けるという点があげられる．テストケースの品質が悪い場合，修正空間に多数のオーバーフィットパッチが存在する可能性があり，バグの修正可否に学習データセットが与える影響が低下すると考えられる．

また，RQ1 における他の脅威として，Prophet の修正空間が OSS ごとのバグの傾向を反映するほど大きくない可能性がある．Prophet は複数箇所への複雑な修正を行うことはできない．

RQ2 における脅威として学習データセットの修正パターン分類を行う際のサンプル数の少なさが挙げられる．各 OSS ごとに 30 個の修正履歴をサンプルとして抽出し分類を行ったが，各データセット全体の修正パターンの傾向を反映できていない可能性がある．

7. まとめと今後の課題

本研究では，版管理システムのバグ修正履歴を利用した自動バグ修正手法の修正結果への学習データセットの影響について二つの Research Questions (RQ) を基に調査を行った．修正対象のバグと学習データの OSS の関係が修正結果にどのような影響を与えるのかという RQ について調査した結果，学習データセットに利用する修正履歴を修正対象と同一の OSS から収集することによって，テストケースに通過するだけの無意味な修正パッチの生成数を減らし，正しい修正パッチに割り当てる優先順位を向上させた．学習データセットの修正パターンが修正結果に影響を及ぼすのかという RQ について調査を行った結果，Prophet により抽出される学習データセットの特徴を本調査における修正パターンの空間の違いと，テストケースの品質による修正結果への影響のため，学習データセットの修正パターンは修正結果に影響を及ぼさなかった．

今後の課題として品質の高いテストケースが用意されているベンチマークに対してのみ実験を行うことで学習データの影響をより正確に調査することや，修正パターンごとに学習データセットを構築し確率モデルを生成することでバグの種類ごとの修正結果がどのように変化するのかの調査を行うことが挙げられる．

謝辞 本研究は，JP15H05306, 18H04097 による助成を受けた．

ⓒ 2019 Information Processing Society of Japan

参考文献

[1] Gazzola, L., Micucci, D. and Mariani, L.: Automatic software repair: A survey, *IEEE Transactions on Software Engineering*, Vol. 45, pp. 34–67 (2017).

[2] Long, F. and Rinard, M.: Automatic patch generation by learning correct code, *ACM SIGPLAN Notices*, Vol. 51, No. 1, pp. 298–312 (2016).

[3] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S.: Automatically finding patches using genetic programming, *Proceedings of the 31st International Conference on Software Engineering*, IEEE, pp. 364–374 (2009).

[4] Ke, Y., Stolee, K. T., Le Goues, C. and Brun, Y.: Repairing programs with semantic code search (t), *Proceedings of the 2015 30th International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 295–306 (2015).

[5] Long, F. and Rinard, M.: Staged program repair with condition synthesis, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, pp. 166–178 (2015).

[6] Yang, J., Zhikhartsev, A., Liu, Y. and Tan, L.: Better test cases for better automated program repair, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, pp. 831–841 (2017).

[7] Kong, X., Zhang, L., Wong, W. E. and Li, B.: Experience report: how do techniques, programs, and tests impact automated program repair?, *Proceedings of the 2015 26th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, pp. 194–204 (2015).

[8] Qi, Z., Long, F., Achour, S. and Rinard, M.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, pp. 24–36 (2015).

[9] Weimer, W., Fry, Z. P. and Forrest, S.: Leveraging program equivalence for adaptive program repair: Models and first results, *Proceedings of the 2013 28th International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 356–366 (2013).

[10] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs, *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, pp. 437–440 (2014).

[11] Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M. and de Almeida Maia, M.: Dissection of a bug dataset: Anatomy of 395 patches from Defects4J, *Proceedings of the 2018 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 130–140 (2018).

ⓒ 2019 Information Processing Society of Japan