

ソースコードメトリクスが自動バグ修正に与える影響の分析

中村 司^{1,a)} 池田 翔^{1,b)} 亀井 靖高^{1,c)} 佐藤 亮介^{1,d)} 鵜林 尚靖^{1,e)}

概要: 自動バグ修正手法の1つに DeepFix がある。これは、C 言語プログラムをトークン列として Seq2Seq モデルで学習し、文法エラーに特化した修正を行う手法である。しかし、他の自動バグ修正手法と同様に、これですべてのバグが解決されるわけではなく、さらにソースコードのどのような性質が修正を妨げているのかも明らかになっていない。そこで本研究ではソースコードの特徴である、(RQ1) 行数、(RQ2) トークン数、(RQ3) 複雑度の3つが自動バグ修正手法の修正結果に影響を与えるのか、調査する。そのために、DeepFix によって修正できたソースコードと、そうでないソースコードとをそれぞれの観点から比較し、修正に影響を与える特徴について考察する。C 言語プログラムの入門用講義で学生が書いたソースコードに対して実験を行ったところ、修正できたソースコードの方が、トークン数や複雑度が小さいことが明らかになった。これにより、DeepFix にとって修正しやすいソースコードの特徴が明らかになり、自動バグ修正の支援ができると考える。

キーワード: 自動バグ修正, DeepFix, ソースコードメトリクス, C 言語

1. はじめに

ソフトウェア開発において、もっともコストのかかる作業の1つがデバッグである。デバッグには開発コスト全体の50%以上が費やされることもあるが、これは主に、バグを特定し除去するために人の手による尽力が必要となるからである[6]。そのため、デバッグにかかるコストを少しでも軽減するための取り組みとして、自動バグ修正の研究が現在盛んに行われている。自動バグ修正とは、人の手ではなくコンピュータによってプログラムのエラーを解消する技術の総称である。

自動バグ修正のツールの1つとして、DeepFix[8]がある。これは、C 言語プログラムを学習し、C 言語の文法的なエラーに特化した修正を行う手法である。最初に提案された論文の中で行われた実験では、27%のプログラムに対してすべてのエラーを修正し、19%のプログラムに対して一部のエラーを修正することができた、という結果が得られている。また、DeepFix はコンパイラによってエラーの有無を判定するため、GenProg[14] や Prophet[11] では必要なテストケースを必要としなかったり、初学者による文法エ

ラーに対応できたりするという利点もある。

しかし、DeepFix による修正の成功率を向上させる試みはあまり行われていない。提案論文[8]では、修正箇所が長くなるほど修正が難しくなると述べられているが、修正の対象となるソースコードの特徴に着目した考察はされていない。そのため、DeepFix による修正に影響を与える要因について、深く調査されていないのが現状である。

本論文では、修正するソースコードのいくつかの特徴について、修正の成否との関係を調査した。インド工科大学および九州大学で収集されたソースコードのデータを DeepFix に適用し、文法エラーを修正できたソースコードとできなかったソースコードの特徴量に有意な差があるのかを検定した。これによって DeepFix で修正しやすいソースコードの性質を明らかにし、自動バグ修正の支援をすることが、本研究の目的である。

2. 背景

2.1 自動バグ修正

デバッグにかかるコストを軽減するため、自動バグ修正の研究が近年盛んに行われている。自動バグ修正の中には、論理エラーを修正するための試みと、文法エラーを修正するための試みがある。論理エラーとは、間違った分岐条件などにより想定と異なる動作をしてしまうエラーである。また、文法エラーとは、変数宣言の欠如のような、プログラミング言語の持つ文法規則に従っていないことによ

¹ 九州大学
Kyushu University

a) t.nakamura@posl.ait.kyushu-u.ac.jp

b) ikeda@posl.ait.kyushu-u.ac.jp

c) kamei@ait.kyushu-u.ac.jp

d) sato@ait.kyushu-u.ac.jp

e) ubayashi@ait.kyushu-u.ac.jp

ソースコード 1 トークン化される前のソースコードの一部

```
2 printf("hello, \uworld! \u n");
```

ソースコード 2 トークン化されたソースコードの一部

```
2 ~ _<APICall>_printf _<op>_( _<string>_ -  
_<op>_) _<op>_;
```

るエラーである。

本研究では、文法エラーの修正に着目する。文法規則のようなプログラミング言語特有の性質は、特にプログラミング初学者にとっては、誤りやすく訂正しにくいものである [5]。文法エラーを自動的に修正することにより、デバッグにかかるコストをより大きく軽減することができると考えられる。

2.2 DeepFix

DeepFix は、2017 年に Gupta らによって提案された、C 言語の文法エラーを修正することに特化したツールである [8]。これは、C 言語で書かれたソースコードをトークン列として Seq2Seq モデル [13] に学習させ、エラーを含むソースコードに対して正しいトークン列を推論することで、文法エラーの修正を行う。モデルの学習に用いられるエラーのないソースコードは、修正の対象となるソースコードとともに入力データとして与えられる。

DeepFix による学習および修正を行うためにトークン化される前のソースコードを**ソースコード 1**に、トークン化されたソースコードを**ソースコード 2**にそれぞれ示す。各トークンはその型とともに行番号の後ろに並べられるが、学習や修正の性能を高めるため、文字列や数値などの、文法的に意味のないデータはその型のみが保持される。学習のために用意された、文法エラーを含まないソースコードは、このようにしてトークン化されたのちに文法エラーを埋め込まれることで学習用データとなる。

DeepFix は、「多くの場合、プログラムのエラーは、エンジニアの不注意や言語に対する知識不足などによって生じる文法エラーである。」という構想のもとで開発されたものであるため、論理エラーは修正の対象としていない。提案論文の実験では全体の 27% のプログラムを完全に、19% のプログラムを部分的に修正することに成功している。

2.3 DeepFix による修正

DeepFix によって行われた修正の実例を**ソースコード 3**に示す。**ソースコード 3**では、12 行目の行末にセミコロンが欠けており、エラーメッセージにもそのような旨の記述がされていた。これに対し DeepFix による修正を行ったものが**ソースコード 4**である。12 行目を見ると修正前に欠けていたセミコロンが付与されていることがわかる。ただ

し、トークン化の段階で捨象された数値および文字列、インデント、空行のデータは手作業による復元を行なった。

一方、**ソースコード 5**も、同様に 9 行目の行末にセミコロンが欠けていることによるエラーである。エラーの数や種類、修正方法は**ソースコード 3**と変わりがないように予想されるが、DeepFix はこれに対して正しい修正を行うことができなかった。

これらのことから、エラーの数や種類以外にも、DeepFix による自動バグ修正に影響を与える要因が存在することが予想される。

2.4 研究の動機

DeepFix の提案論文では、修正の成否に関する考察は少なかった。エラーを解消するために必要な修正パッチのトークン数が多くなるほど修正が難しくなる、と述べられていたが、修正の対象となるソースコードについては分析がされていなかった。

本研究では DeepFix による修正が難しいソースコードの特徴を明らかにする。これにより、その特徴に対するリファクタリングを修正前にあらかじめ行うことで、DeepFix による修正を支援することができると考えられる。

DeepFix による修正に影響を与えるソースコードの性質を調べるため、ソースコードの規模および複雑さと言う 2 つの側面から、本研究では次の 3 つの RQ について調査した。

- (RQ1) 修正ソースコードの行数は修正に影響を与えるのか
- (RQ2) 修正ソースコードのトークン数は修正に影響を与えるのか
- (RQ3) 修正ソースコードの複雑度は修正に影響を与えるのか

3. データセット

本研究では、次に示す 4 つのデータセットを使用した。各データセットに含まれるソースコードの件数など、数値データについては表 1 に示す。

3.1 IITK(Indian Institute of Technology Kanpur)

DeepFix を提案した論文で用いられたデータセットである。インド工科大学カンプル校において Prutor[3] と呼ばれる個別指導システムを用いて収集されたものであり、C 言語プログラミングの入門用講義で課された 93 の課題と、それに対する学生の解答をまとめたものとなっている。

3.2 ED(Education), PS(Pharmaceutical Science), LA(Law)

九州大学で開講された基幹教育科目の講義によるもので

ソースコード 3 修正できたプログラムの例 (修正前)

```

1 #include <stdio.h>
2
3 int main( void )
4 {
5     float a, b;
6     double c;
7     int d;
8
9     a=62.5;
10    b=23.3;
11    c=a*b;
12    d=9
13
14    d*=5;
15    printf("%d\n", d);
16    d/=2;
17    printf("%d\n", d);
18    return 0;
19 }
```

ソースコード 4 修正できたプログラムの例 (修正後)

```

1 #include <stdio.h>
2
3 int main ( void )
4 {
5     float a , b ;
6     double c ;
7     int d ;
8
9     a = 62.5 ;
10    b = 23.3 ;
11    c = a * b ;
12    d = 9 ;
13
14    d *= 5 ;
15    printf ( "%d\n" , d );
16    d /= 2 ;
17    printf ( "%d\n" , d );
18    return 0 ;
19 }
```

ソースコード 5 修正できなかったプログラムの例

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main(){
5     int a,b;
6     printf("整数値を入力してください:");
7     scanf("%d",&a);
8
9     b = a % 2
10    if(b==0){
11        printf("偶数です");}
12    if(b==1){
13        printf("奇数です");}
14    return 0;
15 }
```

ある。C 言語プログラム初級者向けの課題に対する解答として学生が書いたソースコードから構成されている。また、学生の所属は、ED データセットでは教育学部、PS データセットでは薬学部、LA データセットでは法学部が主となつ

表 1 各データセットに含まれるソースコードの件数

データセット	問題数	エラーあり	エラーなし	合計
IITK	93	6,978	46,500	53,478
ED	47	3,157	7,339	10,496
PS	57	1,551	3,834	5,385
LA	21	1,861	3,936	5,797

表 2 主要なエラーメッセージとその主な原因

エラーメッセージ	主な原因
e ₁ expected declaration or statement	閉じ括弧の不足
e ₂ expected identifier	閉じ括弧の過多
e ₃ undeclared	変数や関数の宣言がされていない
e ₄ expected (misc.) tokens	その他のトークンや宣言の不足

表 3 各データセットに含まれるエラーの件数

データセット	e ₁	e ₂	e ₃	e ₄	合計
IITK	1,061	1,659	5,468	6,236	16,556
ED	676	485	887	2,182	8,623
PS	391	176	784	947	4,684
LA	308	257	937	1,013	3,923

ている。九州大学では、TERA-TERM を通じて接続できる作業サーバーが学生に提供されている。そのサーバーには GCC がインストールされているため、学生はコンパイラをインストールすることなくコンパイル作業に取り組める。本データセットは、そのサーバーから SFTP (SSH File Transfer Protocol) を通じてコンパイル時のログを収集したものである [4]。

DeepFix を提案した論文では、データセット中に多く見られたエラーとその原因がまとめられていた。それを日本語に訳したものを表 2 に示す。九州大学で収集されたデータセットについてもここに記されたエラーの件数を調査したところ、表 3 のようになった。ただし、これら 4 種類のいずれにも属さないエラーも存在していたため、エラー e₁~e₄ の件数の合計は全体の合計と一致しない。これにより、九州大学で収集されたデータセットについても、IITK で収集されたデータセットと同様に、変数宣言の欠如や括弧の閉じ忘れが多いことがわかる。また、これらの他に、本来あるはずのない記号が含まれているというエラーも多く見られた。

4. 実験と結果

実験手順の流れを図 1 に示す。

手順 1. 各データセットに対して、DeepFix の提案論文 [8] で用いられていたものと同様のミュレーションによって学習データを生成し、同じデータセットのソースコードに修正を適用した。この際、IITK データセットにおいてはトークン数が 100 以上 400 以下のもの、九州大学で収集さ

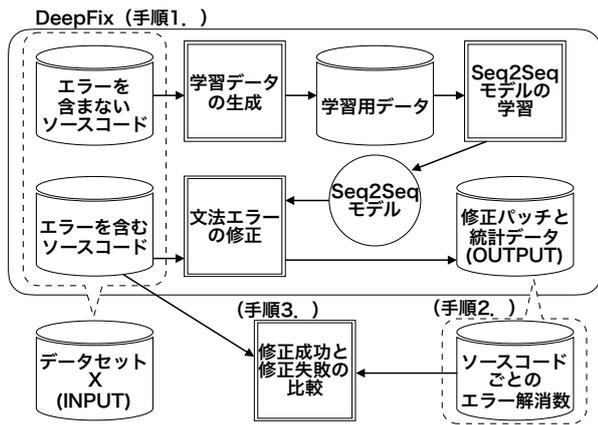


図 1 DeepFix の修正過程と本研究のアプローチ

れた 3 つのデータセットにおいてはトークン数が 25 以上 400 以下のソースコードに限定して、学習データの生成に利用した。この選別は、DeepFix の提案論文で述べられていた実験の設定を踏襲したものであるが、九州大学で収集された 3 つのデータセットについてはデータの大きさを確保するために下限を小さく設定した。

手順 2. DeepFix は、生成した修正パッチだけでなく、それぞれの修正ソースコードに対して解消できた文法エラーの数も出力する。これをもとに、修正を施す前のソースコードを、修正に成功したソースコードと失敗したソースコードに分類した。ただし、ここでは 1 つ以上エラーを解消できたものを修正成功、1 つもエラーを解消できなかったものを修正失敗と定義する。

手順 3. DeepFix による自動バグ修正とその対象となるソースコードの特徴との関係性を調べるため、ソースコードの規模と複雑さという 2 つの方面から、4.1 節から 4.3 節に示す分析を行なった。ただし、本研究では関数やセミコロンのような言語によって扱いの異なるメトリクスは避け、C 言語以外の言語によるものと比較できるような実験結果を得ることを目指した。

手順 4. 以上の実験により、いくつかのソースコードメトリクスについて、修正に成功したソースコードとそうでないソースコードとの間に有意差があるのかを確かめられた。しかし、複数のメトリクスについて有意差が認められた場合、あるメトリクスによる有意差が他のメトリクスによる擬似相関である可能性も考えられる。そこで、修正の成否を目的変数、修正ソースコードの各メトリクスを説明変数とするロジスティック回帰分析を行った。こうして得られた回帰係数から、各メトリクスが修正の成否に影響を与えていることを確認できる。

4.1 (RQ1) 規模-行数

4.1.1 動機

DeepFix による自動バグ修正が難しいソースコードの特徴として、ソースコードの規模が大きいということが考え

られる。そこで、DeepFix が修正に成功したソースコードの集団と、修正に失敗したソースコードの集団に対し、行数の比較を行った。

4.1.2 アプローチ

DeepFix による修正を施す前のソースコードに対し、構造解析ツール Understand^{*1} を用いていくつかのメトリクスを測定し、そのうちの 1 つである行数^{*2}を比較した。また、帰無仮説および有意水準を

帰無仮説 このデータセットにおいて、修正に成功したソースコードと失敗したソースコードは、同じ母集団から抽出されたものである。

有意水準 $\alpha = 0.05$

と設定してマン・ホイットニーの U 検定を行い、比較によって見られた差が有意なものであるのかをデータセットごとに評価した。ただし、このとき検定の多重性に関する問題を解消する必要がある。本研究では、検出力を落とさないためにホルム法^{*3}を採用し、 p 値を有意水準と比較する前に、有意水準にはホルム法による補正をかけた。

4.1.3 結果

データセットごとに修正に成功したソースコードと失敗したソースコードに対して行数を測定した結果の比較を図 2 に示す。さらに、それぞれのデータセットに対してマンホイットニーの U 検定を行ったところ、 p 値はそれぞれ

$$\text{IITK } pvalue = 2.08 \times 10^{-6}$$

$$\text{ED } pvalue = 8.11 \times 10^{-4}$$

$$\text{PS } pvalue = 2.05 \times 10^{-6}$$

$$\text{LA } pvalue = 1.07 \times 10^{-4}$$

となった。これらをホルム法による補正をかけた有意水準と比較すると、

$$\text{PS } pvalue = 2.05 \times 10^{-6} < \alpha/4$$

$$\text{IITK } pvalue = 2.08 \times 10^{-6} < \alpha/3$$

$$\text{LA } pvalue = 1.07 \times 10^{-4} < \alpha/2$$

$$\text{ED } pvalue = 8.11 \times 10^{-4} < \alpha$$

となり、4 つすべてのデータセットにおいて帰無仮説が棄却された。これにより、4 つのデータセットすべてにおいて、修正されるソースコードの行数について有意差が認められた。

4.1.4 考察

4 つすべてのデータセットにおいて有意差が認められたため、修正に失敗したソースコードの方が行数が大きいと言える。しかし、例えば次節で考察するトークン数のような、行数と相関のある他のデータから影響を受けているこ

*1 <http://understand.techmatrix.jp/>

*2 Understand における表記は CountLine。コメントのみの行や空白の行も行数に含む。

*3 複数の帰無仮説を検定する際に、正しい帰無仮説が誤って棄却される可能性を抑えるための方法。N 個の帰無仮説を p 値の小さい順に並べ、 k 番目の帰無仮説に対しては有意水準を $\alpha/(N+1-k)$ とした検定を行う。棄却されない帰無仮説が見つければ、それ以降の帰無仮説をすべて保留とする。

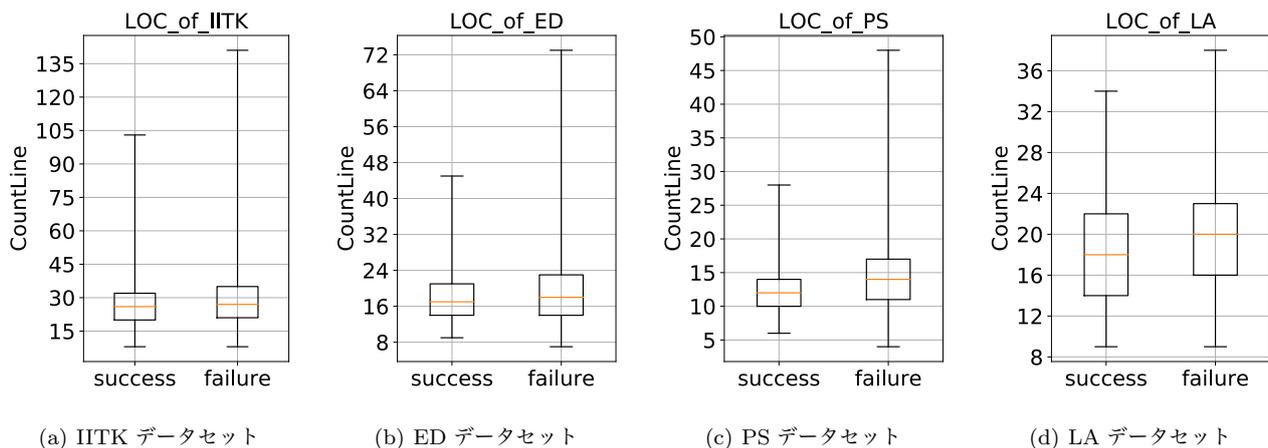


図 2 修正に成功/失敗したソースコードにおける行数の比較

とも考えられる。

4.2 (RQ2) 規模-トークン数

4.2.1 動機

RQ1 では、修正するソースコードの規模を表す指標としてソースコードの行数を採用した。しかし、実際のソースコードには空行やコメントなどプログラム実行上では意味を持たない行も含まれる。そこで、同様に規模が反映される指標のうち、よりソースコードの意味に着目したものとして、本節ではトークン数を採用した。

4.2.2 アプローチ

DeepFix では、学習・修正のためのソースコードはトークン化された状態で扱われる。このとき、トークン化されたソースコードとそのトークン数は出力データベースに保存されるため、これらを抽出して統計的な処理を行なった。また、それぞれのデータセットに対し、ホルム法による補正をした U 検定を RQ1 と同様に行なった。

4.2.3 結果

修正に成功したソースコードと失敗したソースコードに対してデータセットごとにトークン数を測定した結果の比較を図 3 に示す。さらに、それぞれのデータセットに対して RQ1 と同様の U 検定を行なったところ、 p 値はそれぞれ

$$\text{IITK } p\text{value} = 2.11 \times 10^{-25}$$

$$\text{ED } p\text{value} = 2.07 \times 10^{-4}$$

$$\text{PS } p\text{value} = 3.60 \times 10^{-8}$$

$$\text{LA } p\text{value} = 1.21 \times 10^{-6}$$

となった。これらをホルム法による補正をかけた有意水準と比較したところ、4つのデータセットすべてにおいて帰無仮説が棄却された。これにより、4つのデータセットすべてにおいて、修正されるソースコードのトークン数について有意差が認められた。

4.2.4 考察

4つ全てのデータセットで有意差が認められており、修

正に失敗したソースコードの方がトークン数が大きいと言える。DeepFix ではソースコードはトークン化された形で扱われるため、行数による修正への影響は、トークン数を介した間接的なものであったことが考えられる。

ソースコード 6 は、行数および文法エラーの内容に関してはソースコード 5 と概ね一致するが、DeepFix によって文法エラーを修正することができた。要因の1つとして、変数の使い方や分岐条件の違いにより、トークン数が小さくなっていることが考えられる。

4.3 (RQ3) 複雑度

4.3.1 動機

RQ1 および 2 では、修正するソースコードの規模に着目した分析を行った。しかし、自動バグ修正の障害となりうるものとして、規模以外にもソースコードの複雑さが予想される。そこで、本節ではソースコードの複雑さに着目した分析を行なった。

ここでは、ソースコードの複雑さを表す指標として循環的複雑度、Halstead のソフトウェアサイエンス [9] の 2つを採用した。これにより、動作をフローチャートで表した際の分岐の多さ、ソースコードに含まれるオペランドやオペレータの多様さから、それぞれソースコードを評価する。

4.3.2 アプローチ

循環的複雑度については RQ1 と同様に Understand による測定^{*4}を行い、Halstead のソフトウェアサイエンスについては commentedCodeDetector^{*5} による測定を行った。それぞれの指標に対して、RQ1 および RQ2 と同様の U 検定を行った。

4.3.3 結果

実験結果を図 4 および図 5 にそれぞれ示す。また U 検定における p 値は、循環的複雑度については

^{*4} Understand における表記は AvgCyclomatic であり、各関数の循環的複雑度をソースコードごとに平均したメトリクスである。

^{*5} <https://github.com/dborowiec/commentedCodeDetector>

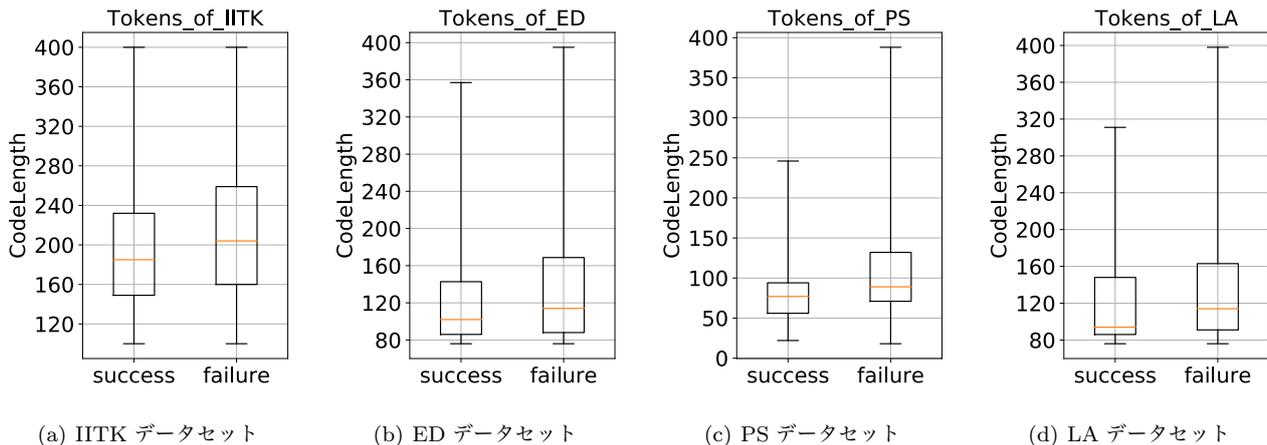


図 3 修正に成功/失敗したソースコードにおけるトークン数の比較

ソースコード 6 トークン数の小さいプログラムの例

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main (){
5      int a;
6
7      printf("整数を入力してください=");
8      scanf("%d",&a);
9
10     if(a%2==0){
11         printf("%d は偶数です \n",a);
12     } else {
13         printf("%d は奇数です \n",a);
14     }
15     return 0;
16 }
```

IITK $pvalue = 1.93 \times 10^{-2}$

ED $pvalue = 3.90 \times 10^{-1}$

PS $pvalue = 5.98 \times 10^{-8}$

LA $pvalue = 5.89 \times 10^{-6}$

となり, Halstead のソフトウェアサイエンスについては

IITK $pvalue = 1.60 \times 10^{-16}$

ED $pvalue = 8.83 \times 10^{-3}$

PS $pvalue = 6.24 \times 10^{-7}$

LA $pvalue = 1.07 \times 10^{-4}$

となった。これらをホルム法による補正をかけた有意水準と比較したところ, 循環的複雑度については IITK, PS および LA データセットにおいて, Halstead のソフトウェアサイエンスについては 4 つのデータセットすべてにおいて, それぞれ帰無仮説が棄却された。これにより, 循環的複雑度については IITK, PS および LA データセットにおいて有意差が認められ, Halstead のソフトウェアサイエンスについては 4 つのデータセットすべてにおいて有意差が認められた。

4.3.4 考察

循環的複雑度について, IITK, PS および LA データセットの 3 つにおいては有意差が認められた。また, ED データセットにおいては有意差こそ認められなかったが, 図 4 によるとデータの範囲に差があり, 循環的複雑度が 8 より大きいものは修正できていないことがわかる。これらのことから, 循環的複雑度も修正に失敗したソースコードの方が大きいと言える。

また, Halstead のソフトウェアサイエンスについては, 4 つすべてのデータセットにおいて有意差が認められた。すなわち, Halstead のソフトウェアサイエンスも修正に失敗したソースコードの方が大きいと言える。これは, ソースコードに含まれるトークンの種類が多くなることにより, 正しいソースコードの推論が難しくなるためであると考えられる。

4.4 ロジスティック回帰分析

以上の実験で得られた結果の正当性を確かめるため, 修正が成功したかどうかを目的変数, 修正ソースコードの各メトリクスを説明変数とするロジスティック回帰分析を行った。ただし, データの範囲が大きい行数, トークン数, Halstead のソフトウェアサイエンスの 3 つは, 外れ値の影響を抑えるためにそれぞれの常用対数をデータとして使用した。また, 説明変数となる各メトリクスの分散拡大係数 (VIF) を確認し, その値が 10 を超えるものが無くなるよう, 他の変数との間に強い相関を持つ変数を除外した。これにより, メトリクスどうしの間に相関がある場合はそれを軽減し, あるメトリクスと修正結果との相関が他のメトリクスによる擬似相関であった可能性について検証できる。そして, 修正率を P , 各メトリクスを X_i , ロジットを Y とすると, 回帰式は式 (1) および式 (2) のようになる。

$$P = \frac{1}{1 + \exp(-(C_0 + \sum(C_i * X_i)))} \quad (1)$$

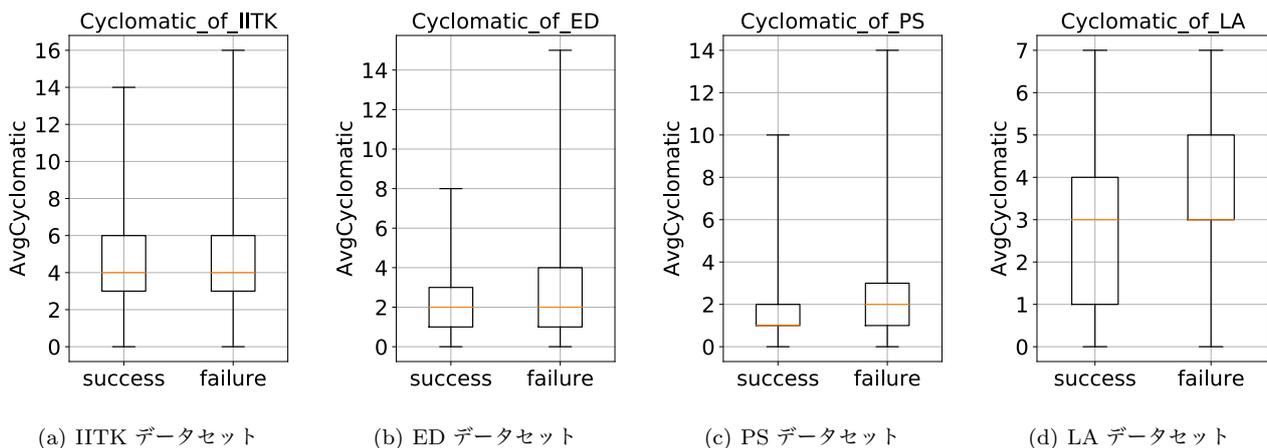


図 4 修正に成功/失敗したソースコードにおける循環的複雑度の比較

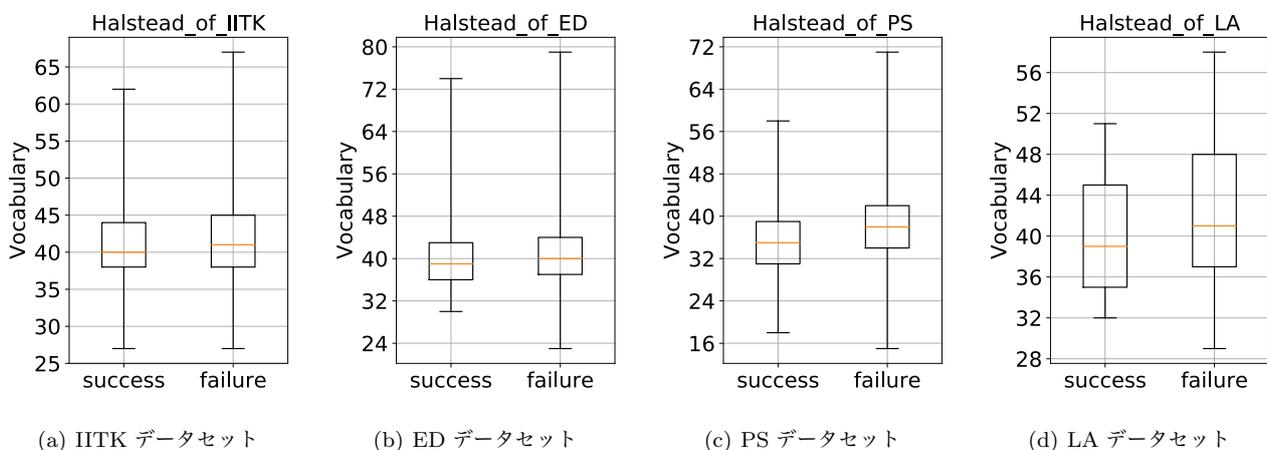


図 5 修正に成功/失敗したソースコードにおける Halstead のソフトウェアサイエンスの比較

$$Y = \ln \frac{P}{1-P} = C_0 + \sum (C_i * X_i) \quad (2)$$

各データセットのなかでエラー数の多かった課題を3つずつ選び、それぞれの課題に以上のような回帰分析を行った。これら12の課題のうち5つの課題で0.1以上の擬似決定係数が得られたが、その定数項 C_0 および各メトリクスの回帰係数 C_i は表4のようになった。どの課題も回帰係数は0に近い値ばかりではないため、修正の成否に影響を持つメトリクスが課題ごとに存在していると言える。

4.5 結果のまとめと考察

修正ソースコードの行数、トークン数、複雑度の3つすべてにおいて、修正に失敗したソースコードの方が大きいという結果が得られた。そのなかでも特にトークン数とHalsteadのソフトウェアサイエンスについては、DeepFixがソースコードをトークン列として扱うことから、修正の成否との間に相関があると考えられる。また、ロジスティック回帰分析の結果からも、修正ソースコードのメトリクスが修正結果に影響を与えていることが確かめられた。

5. 関連研究

自動バグ修正の研究は近年盛んに行われており、DeepFix以外にも様々な手法が提案されている。

DeepFixを提案したGuptaらは、その後も自動バグ修正の研究を続けており、2019年にはRLAssistを発表している[7]。RLAssistはプログラミング初心者のサポートを目的としており、DeepFixと同様にC言語の文法エラーを修正する。ソースコードの学習にはA3C (Asynchronous Advantage Actor-Critic) による強化学習を採用しており、DeepFixよりも柔軟に修正を行うことができる。さらに、人間や別のエージェントによるデモンストレーション[1]を取り込んだ場合には、より高い性能を発揮できることも示されている。

Longらの提案したProphetは、エラーのあるプログラムに対して修正パッチを生成する[11]。まず、テストスイートに含まれる入力でプログラムを実行し、エラーの原因となり得る命令を絞り込む。続いて、絞り込まれたそれぞれの命令に対応する修正パッチを生成し、それらを修正が

表 4 各データセットにおけるロジスティック回帰分析の結果

データセット	課題 ID	回帰係数					擬似決定係数 R^2
		定数項	行数	トークン数	循環的複雑度	Halstead	
PS	6_b-2	14.1897	27.3223	-	-0.3046	-31.1444	0.3063
LA	10-1	-5.0789	-3.0696	-33.3392	19.1411	7.8199	0.2469
LA	sample	2.4305	-0.7794	-5.5143	-0.0437	4.8469	0.1214
ED	8_a-2	0.2850	-5.9848	-2.4283	1.3017	4.1457	0.1159
PS	4_a-1	4.0876	-7.2477	-16.3479	11.6916	12.1166	0.1093

期待できる順に序列づける。この序列づけには、オープンソースソフトウェアの修正履歴を学習したモデルが利用されている。この修正パッチの序列から、テストスイートに対して正しい出力ができたものだけを選んで出力する。提案論文の実験では、実際のエラー 69 件に対して適用した結果、19 件に対して正しい修正パッチを出力することに成功している。

また、本研究のように、プログラムの分析や改善を行うツールの持つ、性質や特性を調査した研究も存在する。

Papadakis らは、ミューテーションスコアとバグ検出との間に本当に相関があるのかを調査した [12]。ミューテーションスコアとバグ検出の相関について調べた先行研究は数多く存在していたが、テストスイートの大きさによる影響を考慮に入れて調査した研究は少なかった。そこで、CoREBench[2] と Defects4J[10] の 2 つのデータセットを使用し、テストスイートの大きさを固定した場合としなかった場合で、それぞれ相関の分析を行なった。その結果、この相関は単にテストスイートの大きさによる効果だったと結論づけている。

これらに対し、提案されて以来あまり分析の進んでいなかった DeepFix について取り上げた点や、修正対象となるソースコードについて着目した点が、本研究の特徴である。

6. 妥当性への脅威

内的妥当性。 本研究で利用したデータセットのうち、ED、PS および LA データセットの 3 つについては、学生がコンパイルしたソースコードを記録し集めたものであった。そのため同じ内容のソースコードが複数存在している場合があり、それらが極端に修正しやすいものであったり極端に修正しにくいものであったりすると、実験結果に影響を与えている可能性がある。

また、本研究ではトークン数が極端に大きいソースコードや極端に小さいソースコードはデータセットから除外して実験を行ったため、これらを含めると結果が変わる可能性がある。

さらに、ロジスティック回帰分析においては分散拡大係数をもとに変数を選択したが、それでもなお係数が負となる変数が存在しており、係数が正のものと影響力を相殺し

あっている可能性がある。

外的妥当性。 本研究で利用したデータセットは、すべて講義の課題に対する学生の解答をまとめたものであった。そのため、学生の習熟度の影響を受けることが考えられ、どのような学生の集団に対しても必ず同様の結果が得られるとは言い切れない。

また、本研究の調査対象は DeepFix による実行結果のみとなっている。そのため、本研究の調査結果が他の自動バグ修正ツールの実行結果にも当てはまるとは言い切れない。

構成概念的妥当性。 本研究では、DeepFix によって修正できなかったソースコードの特徴を、ソースコードの規模と複雑さ(量と質)という 2 面から予測した。しかし、RQ として挙げた (RQ1) 行数、(RQ2) トークン数、(RQ3) 複雑度の 3 つで規模と複雑さのあらゆる側面を検証できているとは言い切れず、調査すべき性質が他にも存在する可能性がある。

7. まとめ

本研究では、DeepFix による自動バグ修正に対して、その対象となるソースコードの持つ性質がどのように影響しているのかを調査した。まずはソースコードの規模に着目し、ソースコードの行数とトークン数から分析を行った。どちらも修正を難しくしている要因ではあるものの、より直接的に作用しているのはトークン数であることが明らかになった。続いてソースコードの複雑さに着目し、循環的複雑度や Halstead のソフトウェアサイエンスについて分析を行なった。U 検定の結果やデータの分布から、ソースコードの複雑さも修正を難しくしていることが明らかになった。

今後の課題として、同様の調査を他のデータセットに対しても行うことで分析をより正確にすることや、本研究の結果を利用してソースコードを DeepFix にとって修正しやすい形式へと整形するツールを実装する試みが挙げられる。

謝辞 本研究の一部は JSPS 科研費 JP18H04097・JP18H03222・JP19H04086 の助成を受けた。

参考文献

- [1] Argall, B. D., Chernova, S., Veloso, M. M. and Browning, B.: A survey of robot learning from demonstra-

- tion, *Robotics Auton. Syst.*, Vol. 57, No. 5, pp. 469–483 (2009).
- [2] Böhme, M. and Roychoudhury, A.: CoREBench: studying complexity of regression errors, *International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - July 21 - 26, 2014*, pp. 105–115 (2014).
- [3] Das, R., Ahmed, U. Z., Karkare, A. and Gulwani, S.: Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis, *CoRR*, Vol. abs/1608.03828 (2016).
- [4] Fu, X., Shimada, A., Ogata, H., Taniguchi, Y. and Suehiro, D.: Real-time learning analytics for C programming language courses, *Proceedings of the Seventh International Learning Analytics & Knowledge Conference, Vancouver, BC, Canada, March 13-17, 2017*, pp. 280–288 (2017).
- [5] Fu, X., Yin, C., Shimada, A. and Ogata, H.: Error log analysis in c programming language courses, *Doctoral Student Consortium (DSC) - Proceedings of the 23rd International Conference on Computers in Education, ICCE 2015*, pp. 641–650 (2015).
- [6] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Trans. Software Eng.*, Vol. 45, No. 1, pp. 34–67 (2019).
- [7] Gupta, R., Kanade, A. and Shevade, S. K.: Deep Reinforcement Learning for Syntactic Error Repair in Student Programs, *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pp. 930–937 (2019).
- [8] Gupta, R., Pal, S., Kanade, A. and Shevade, S. K.: DeepFix: Fixing Common C Language Errors by Deep Learning, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pp. 1345–1351 (2017).
- [9] Halstead, M. H. M. H.: *Elements of software science*, New York : Elsevier (1977). "Elsevier computer science library."
- [10] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs, *International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - July 21 - 26, 2014*, pp. 437–440 (2014).
- [11] Long, F. and Rinard, M.: Automatic patch generation by learning correct code, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 298–312 (2016).
- [12] Papadakis, M., Shin, D., Yoo, S. and Bae, D.: Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 537–548 (2018).
- [13] Sutskever, I., Vinyals, O. and Le, Q. V.: Sequence to Sequence Learning with Neural Networks, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems, NIPS 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 3104–3112 (2014).
- [14] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S.: Automatically finding patches using genetic programming, *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 364–374 (2009).