特別研究報告

題目

プログラミングコンテストのデータを用いたファインチューニングによるバグ検出器の性能改善

指導教員

肥後 芳樹 教授

報告者

高田 智生

令和６年２月７日

大阪大学 基礎工学部 情報科学科

令和 5 年度 特別研究報告

プログラミングコンテストのデータを用いた
ファインチューニングによるバグ検出器の性能改善

高田 智生

内容梗概

ソフトウェア開発におけるバグの検出・修正は重要な課題であり，効率的で適切なデバッグ手法の開発が望まれる．変数の誤用や演算子の誤りなどのバグを対象とする機械学習によるバグ検出・修正は有効な手法であるが，十分な性能を実現するために大規模なバグ修正のデータセットを作成する必要がある．本研究ではプログラミングコンテストに着目し，ユーザによる提出コードからバグ修正のデータセットを作成した．プログラミングコンテストを用いた実際のバグ修正データセットをバグ検出器のファインチューニングに用いて性能改善を試みた．また，プログラミングコンテストを用いたバグ修正データセットと一般的なバグ修正との違いやそれぞれの特徴について考察を行い，バグ検出器のさらなる性能改善に向けた方針について考える．

主な用語

ニューラルバグ検出器
トランスフォーマモデル
プログラミングコンテスト

目 次

1 はじめに **4**

2 背景 **5**

2.1 ソフトウェアのバグ検出・修正 5
2.2 単一トークンバグ 5
2.3 機械学習による単一トークンバグの検出・修正 6
2.4 バグ修正のデータセットに関する性能比較 8
2.5 競技プログラミング 11
2.6 Project CodeNet 11

3 提案手法 **12**

4 プログラミングコンテストのデータを用いた実バグ修正データセットの作成 **14**

4.1 データセットの作成手順 14
4.2 作成したデータセットの概要 14

5 作成したデータセットを用いたモデルの性能比較 **17**

5.1 人工バグ修正データセットによる学習済みモデル 17
5.2 評価手法 17
5.3 評価用テストデータ 17
5.4 評価基準 18

6 評価 **19**

7 考察 **20**

7.1 競プロ実バグ修正データと GitHub 実バグ修正データの比較 20
7.2 競プロ実バグ修正データと GitHub 実バグ修正データを混合したデータによるファインチューニング 21

8 妥当性の脅威 **22**

8.1 内的妥当性への脅威 22
8.2 外的妥当性への脅威 23

9 おわりに **24**

謝辞 **25**

参考文献 **26**

1　はじめに

ソフトウェア開発におけるバグの検出と修正は多くのコストを必要とする工程であるとともに，バグは現代において社会に与える影響が非常に大きい課題でもあるため，効率的なデバッグ手法の研究が望まれる [6][7][9][12].

デバッグ支援を目的とする研究の 1 つとして，機械学習を用いたバグの検出・修正の手法が研究されている．機械学習を用いたバグ検出・修正は，変数の誤用や演算子の誤りなどのバグを対象としており，静的チェックのみによりバグの原因となる個所を特定・修正するものである．しかしバグ検出・修正を機械学習によって行うためには多くのバグ修正過程のデータが必要であり，実際のバグ修正のデータを十分な規模に集めることは難しい．そのため人工的にバグ修正データを生成し，機械学習に用いる研究も行われている [2].

しかし実際のバグ修正のデータセットと人工的に生成されたバグ修正のデータセットを比較する Richter の研究 [11] において，人工的に生成されたバグ修正は検出器の性能の面で実際のバグ修正に劣り，実際のバグ修正は十分な規模のデータセットの作成が難しいことが示されている．また，人工的に生成されたバグのデータを用いた事前学習の後に，実際のバグ修正データを用いたファインチューニングを行うことで性能を改善させられることが示されるとともに，大規模な実際のバグ修正のデータセットの作成が実現することでさらなる性能改善を試みることが提案されている．

本研究ではプログラミングコンテストに注目し，大規模な実際のバグ修正データセットを作成することで，バグ検出器の更なる性能改善を試みた．

プログラミングコンテストの特徴として，ユーザにより大量のソースコードが提出されることや，用いられるプログラミング言語の多様性が挙げられる．また提出コードはオンラインジャッジシステム [14] のテストにより評価が行われており，バグを含むコードであるかどうかについての厳密なタグ付けが行われている．提出されたデータからユーザによる修正過程を取り出すことで，大規模で正確な実際のバグ修正データからなるデータセットの作成が可能であると考えた．プログラミングコンテストの提出コードを多く含むデータセット Project CodeNet[10] を採用し，データセットを作成した．

また，作成したデータセットを学習に用いた際の性能について，実際にバグを含むコードに対するバグ検出・修正により評価する．バグの検出性能，修正性能それぞれの変化を比較することにより，作成したプログラミングコンテストを用いたバグ修正データセットと，一般的な実際のバグ修正データの違いについて考察を行った．バグ検出器のさらなる性能改善に向け，プログラミングコンテストを用いたバグ修正データセットの特徴について考える．

2 背景

2.1 ソフトウェアのバグ検出・修正

ソフトウェア開発においてバグの検出と修正は手作業や時間を必要とする工程であり，ソフトウェア開発に必要なコストの多くがデバッグに割り当てられている [6][7]．またソフトウェアが広く浸透している現在の社会において，バグが社会に与える影響は大きく，経済的損失から人命に関る重大な事故まで多岐にわたる [9][12]．

ソフトウェア開発の支援やバグによる損失の防止のために，デバッグを支援するためのツールの開発や研究が望まれている．近年では，デバッグ支援のための研究として，テストケース実行時の結果と実行経路の情報に基づいてバグの位置を推定するスペクトラムベースフォルトローカライゼーション [13] など，発生したバグの原因位置を推定するフォルトローカライゼーションの手法が提案されている [5]．

このようにテストケースの実行によるデバッグの研究が行われる一方，静的チェックによるデバッグ支援ツールの研究も行われている [5]．静的チェックによるデバッグの例として，身近にはエディタの構文チェック機能などが挙げられる．静的チェックによるデバッグ支援は実行を伴わないソースコードに対する解析のみで行われるため，テストケースの作成を伴うデバッグと比較して低コストである．

2.2 単一トークンバグ

単一トークンバグはプログラム中のトークン 1 つを置き換えるのみで修正が可能なバグである [8]．サイズの小さな単純なバグであるため，ソフトウェア開発中に単一トークンバグが生じる頻度は高く，また見逃されることも多いバグである [1]．

単一トークンバグの例を図 1 に示す．このコードは Project CodeNet[10] に含まれている，入力された非負整数の和の桁数を出力するプログラムの修正過程である．修正前のコードが誤った解答を出力するプログラムであるのに対し，8 行目の二項演算子を修正するのみで適切なプログラムに修正される．

また，単一トークンバグの特徴として，検出・修正が次の 3 つのタスクをまとめて 1 つのトークン置換の操作として表現することが可能である点が挙げられる．

- プログラムがバグを含むものであるかどうかの分類
- バグの検出
- 適切な修正後のトークンを特定

修正前

```
1 import sys.math
2
3 inputs = list()
4
5 for n in sys.stdin:
6         inputs.append(list(map(int,n.split())))
7
8 for n in inputs:
9         print(math.floor(math.log10(n[0]+n[1]))+1)
```

修正後

```
1 import sys.math
2
3 inputs = list()
4
5 for n in sys.stdin:
6         inputs.append(list(map(int,n.split())))
7
8 for n in inputs:
9         print(math.floor(math.log10(n[0]*n[1]))+1)
```

図 1: 単一トークンバグの例

例えば図 1 において，修正前のコードに対するバグの修正はトークン “+” をトークン “*” に置換する操作として表現される．ここで単一トークンバグにおいてバグの検出は，このトークン置換により書き換えられるべきトークンをバグの出現箇所として特定することを意味する．また，修正はこのトークン置換内容を特定してバグの修正内容とすることを意味する．

2.3 機械学習による単一トークンバグの検出・修正

単一トークンバグの検出・修正のトークン置換を機械学習によって推測する研究が行われている [2][11]．これらの研究においてバグの出現箇所・修正候補の推測は，プログラムのトークン列を読み込み，適切なトークン置換を推測する機械学習モデルにより行われる．このような機械学習モデルの学習には次のような要素からなるバグ修正のデータセットが必要である．

- バグを含むコードスニペット
- バグを修正するためのトークン置換

しかし機械学習によるバグ検出・修正の分野には，十分に大規模なバグ修正のデータセッ

トを準備する必要があるなど多くの課題がある [2].

2.3.1 データセットのための人工バグの生成

バグ修正のデータセットを生成し機械学習に用いるバグ検出器 BUGLAB の研究が行われている [2]. バグ修正のデータセットを与える教師あり学習による検出器と異なり，BUGLAB はコードの書き換えにより一般的なバグに近いバグを含んだコード（人工バグ）の修正データを生成して学習する自己教師型の検出器である.

人工バグを生成する書き換えの操作は変異と呼ばれ [11]，書き換え規則に従ってトークン置換を行いバグを発生させることで作られる．人工バグの例を図 2 に示す．このコードは Project CodeNet[10] に含まれる 1 から 9 までの数の積を出力するプログラムのコードである．この例において，コードに含まれるトークンからランダムに選ばれた 6 行目の数値リテラルを他の数値リテラルに置き換える変異の操作を行っており，変異後のプログラムは意図した動作を行わないため，変異後のコードはバグを含むと言える.

変異前

```
1 M = 9
2 N = 9
3
4 def main():
5     for i in range(1,M+1,1):
6         for j in range(1,N+1,1):
7             mult = i * j
8             print(str(i) + "x" + str(j) + "=" + str(i * j))
9 main()
```

変異後

```
1 M = 9
2 N = 9
3
4 def main():
5     for i in range(1,M+1,1):
6         for j in range(0,N+1,1):
7             mult = i * j
8             print(str(i) + "x" + str(j) + "=" + str(i * j))
9 main()
```

図 2: 人工バグの例

1 つの正しいコードに対して想定される変異のためのトークン置換は多くあるため，候補となるトークン置換からランダムに選択されたトークン置換を行い変異を起こす．置換後の

候補となるトークンは一般的に多くみられる単一トークンバグである次の 4 つのバグを生成するためのトークンが含まれる．

- 変数の誤用
- 二項演算子の誤用
- 単項演算子の誤用
- リテラルの誤用

また学習の際，人工バグの変異を行った部分を修正箇所として扱うことでバグ修正のデータとして学習を行う．この時，変異前の正しいコードはデータセットにおいてバグ修正後のコードとして扱われ，変異後のコードはバグ修正前のコードとして扱われる．またバグ修正に用いられるトークンとして，変異させたトークンの変異前のものを記述する．

人工バグ修正のデータセットの特徴として規模の拡大が容易であることが挙げられる．変異を発生させる元となる正しいコードの数を増やす方法に加え，１つの正しいコードに対して変異を行う回数を増やすことによりさらに多くの人工バグ修正のデータを作成することが可能である．この時，１つのコードに対して複数の人工バグを生成する場合には同じ変異を発生させないように注意を行う必要があり，さらに元のコードに対して変異を行う回数は重複を避けるために適切な範囲に設定される必要がある．

2.4 バグ修正のデータセットに関する性能比較

Richter らは，人工バグによるデータセットと実バグによるデータセットのそれぞれの学習がバグ検出・修正の性能に対して与える影響についての研究を行った [11].

2.4.1 人工バグにより作成されるデータセット

人工バグによるデータセットの作成は BUGLAB[2] と同様の手法により行われる．GitHub に公開されているソースコードから関数を切り出したコードスニペットを用い，変異の操作により人工バグのデータセットを作成する．

2.4.2 実バグにより作成されるデータセット

実バグによるデータセットの作成は GitHub で公開されているオープンソースプロジェクトのコミット履歴に対するスクレイピングにより得られる．トークンが置き換えられることに加え，コミット時のメッセージにバグ修正を報告するキーワードが含まれているコードをバグの修正過程として検出する．検出されたコミット履歴の変更前，変更後，変更内容をそ

れぞれ修正前のコード，修正後のコード，修正内容とする実バグ修正のデータとしてデータセットに加える．スクレイピングによって得られたコードは手作業により確認されておらず，実際にはバグが適切に修正されていない場合を含む可能性がある．

2.4.3 単一トークンバグの検出器

単一トークンバグの検出はトークンが前後の文脈に適合しない可能性を計算することによって行われる．機械学習を行ったモデルを用いて，各トークンがバグである確率をソフトマックス分布として計算し，最もバグである可能性の高いトークンを推測する．

2.4.4 単一トークンバグの修正器

単一トークンバグの修正も同様に，各トークンが修正に用いられるトークンとして適切である確率をソフトマックス分布として求め，修正に用いるトークンを特定する．この時，バグの修正に用いられるトークンが同一プログラム中で用いられているとは限らない．そのため修正に用いる候補となるトークンの集合には，同一プログラム中の他のトークンに加え，一般的に用いられることの多い次のようなトークンを候補として与える．

- 二項演算子
- 単項演算子
- 頻繁に用いられる数値リテラル

2.4.5 モデルの比較

人工バグ修正データセット，実バグ修正データセットを用いた機械学習により次のようなモデルを作成し，その性能についての評価を行う．

- 人工バグ修正データセットのみを学習に用いたモデル
- 実バグ修正データセットのみを学習に用いたモデル
- 人工バグ修正データセットと実バグ修正データセットを混合して学習したモデル
- 人工バグ修正データセットのみを学習に用いたモデルに，ファインチューニングとして実バグ修正データセットをさらに学習したモデル
- 学習に用いる人工バグ修正データセットの規模（1 つの正しいコードあたりの変異回数）を拡大したモデル

表 1: Richter らの研究におけるモデルの性能

モデル	検出・修正率 (%)	誤検出率 (%)
人工バグのみ	21.4	25.2
実バグのみ	10.9	12.1
人工バグ・実バグの混合	24.5	27.0
人工バグのファインチューニング	32.2	26.7
変異回数の拡大	24.9	30.0
変異回数の拡大後にファインチューニング	36.7	22.0

- 学習に用いる人工バグ修正データセットの規模を拡大し，さらに実バグによりファインチューニングを行ったモデル

2.4.6 モデルの評価

モデルの評価は実際のソースコードに対するバグ検出・修正により行う．対象としているバグは 2.3.1 に挙げる一般的に多くみられる 4 種類の単一トークンバグであり，実際のバグ修正が行われたコードに対するバグの検出・修正の出力が，実際の修正と一致するかどうかによりモデルの性能を評価する．BUGLAB の評価 [2] に用いられたテストデータと同じものを用いる．

また評価のために，検出・修正が成功した割合に加え，正しいコードに対してバグ検出・修正を行った際の誤検出率についても計測を行う．

2.4.7 結果

それぞれのモデルにおけるテスト結果を表 1 に示す．

人工バグ修正データセットによる学習の後に実バグ修正データセットをファインチューニングに用いる検出器が高い性能を発揮する．実バグ修正データセットのみを学習に用いたモデルの性能は不十分であるが，ファインチューニングに実バグ修正データセットを用いることでモデルの検出・修正の能力が改善し，誤検出率も増加しない．

また，学習に用いる人工バグ修正データセットの規模を拡大することも検出・修正の能力の向上につながるが，誤検出率の観点から実バグ修正データセットを用いたほうが効果的である．

2.4.8 課題

Richter らの研究において，より大きな実バグ修正データセットの作成の必要性が述べられている．そのため，実バグデータセットの大規模化によりさらなる性能向上が考えられる．

2.5 競技プログラミング

競技プログラミングは参加者が与えられた問題に対するコーディングを行い，プログラミング能力を競うものである [4]．プログラミングコンテストとして数多く開催されており，例えば国内で実施されている主なコンテストとしては，AtCoder[1]や AIZU Online Judge[2]が挙げられる．これらのコンテストにおいて，参加者により提出されたプログラムはオンラインジャッジシステムによりコンパイル・実行される [14]．その際，提出されたプログラムはすべて出題者の用意したテストケースを用いて，正確さや性能についての自動評価が行われる．

近年では競技プログラミングの人気は高く，ユーザ数は増加し，開催回数も多い [3]．

2.6 Project CodeNet

Project CodeNet は機械学習の分野での利用を目的に提供される大規模で信頼性の高いソースコードのデータセットである [10]．このデータセットに含まれるコードは日本国内で開催された AIZU Online Judge，AtCoder の 2 つの大規模な競技プログラミングコンテストで出題された問題約 4000 問に対して提出されたものであり，それぞれのコードに対してオンラインジャッジシステムのテストケースによる評価がつけられている．データセットには次の内容が含まれる．

- 出題された問題の問題文
- その問題に提出されたコードのファイル名の一覧，及びそのコードのプログラミング言語，提出ユーザ，オンラインジャッジの結果ステータス（表 2）
- 提出されたコードのファイル本体

[1] https://atcoder.jp/
[2] https://judge.u-aizu.ac.jp/onlinejudge/

表 2: AIZU Online Judge におけるジャッジ結果の例

ステータス	略称	内容
Accepted	AC	受理
Compile Error	CE	コンパイルエラー
Wrong Answer	WA	出力結果が誤っている
Time Limit Exceeded	TLE	制限 CPU 時間をオーバーした
Memory Limit Exceeded	MLE	制限メモリーをオーバーした
Output Limit Exceeded	OLE	制限サイズを超えた出力を行った
Runtime Error	RE	実行中ランタイムエラーが発生した
WA: Presentation Error	PE	出力形式が誤っている

3 提案手法

本研究ではプログラミングコンテストのデータを用いたファインチューニングにより，バグ検出・修正モデル [11] の性能改善を試みる．

プログラミングコンテストのデータを用いた大規模で信頼性の高いバグ修正データセットを作成する．プログラミングコンテストのデータの特徴として，まずユーザによる提出プログラムが多く保管されており，修正過程を抽出することで大規模な実バグ修正データセットの作成が可能であることが挙げられる．また全ての提出プログラムは出題者によって用意されたテストケースにより評価されているため，バグを含むコードと含まないコードが適切に分類されており，抽出した実バグ修正データセットも正しいバグ修正過程によって構成されている可能性が非常に高いと予想される．

また，プログラミングコンテストから作成した実バグ修正データセットをバグ検出・修正モデルの性能改善に用いることは可能であるかについて検証を行う．Richter らの研究 [11] と同様，人工バグ修正データセットを用いた機械学習によるバグ検出・修正モデルに対してファインチューニングを行い，モデルの性能を比較することで作成したデータセットによる機械学習への影響を評価する．本研究ではプログラミングコンテストのデータとして Project CodeNet[14] を採用する．

また機械学習・モデルの評価には次のように Richter らの研究 [11] と同様の手法・データセットを用いる．

- 人工バグ修正による学習を行ったモデルは Richter らの研究で用いられたものを利用する

- モデルの評価には BUGLAB[2] の評価に用いられたものと同じテストデータセットを用いる
- データセットによるモデルの学習は Richter らの研究で用いられたプログラムと同じものを利用する
- 学習に用いる一般的な実バグ修正データセットとして，Richter らの研究で用いられたデータセットを利用する．説明のため，このデータセットを GitHub 実バグ修正データセットと呼ぶ

4　プログラミングコンテストのデータを用いた実バグ修正データセットの作成

4.1　データセットの作成手順

Project CodeNet のデータを用い，次の手順でデータセットを作成した．ただし，本研究は Python で記述されたソースコードのバグ検出・修正を対象としているため，データセットを作成する際は Project CodeNet に含まれる拡張子が “.py” であるコードのみを取り扱う．

1. 同一ユーザによる同一問題に対する提出コードの組のうち，ステータスが “Accepted” 以外から “Accepted” に変化している組を全て検出する．
2. 各コードをトークン分離し，トークン列の長さがモデルの学習対象の長さを超えるものを除外する．
3. コードの組のトークン列の差分を求める．
4. トークン列中の差分が，１トークンの入れ替えのみである，またはトークン “not” が追加・削除されているのみであるコードの組を単一トークンバグ修正が行われる前後のコードとしてデータセットに追加する．
5. バグの種類は Richter らの研究に従い以下の 6 種類に分類する．いずれにも属さないバグは除外する．
    - 変数の誤用
    - バイナリ演算子の誤用
    - 比較演算子の誤用
    - 論理演算子の誤用
    - 代入演算子の誤用
    - リテラルの誤用

4.2　作成したデータセットの概要

Project CodeNet に含まれる 4,053 問に対する提出コードから，実バグ修正からなるデータセットを作成した．バグの種類別のデータ数を表 3 に示す．

Richter らの研究 [11] と比較してデータ数は増加している．

また，データセット作成時に計測した Project CodeNet に含まれるデータの特徴を下記に示す．

表 3: バグの種類

バグの種類	バグの内容	競プロバグ	GitHub バグ
変数の誤用	変数	14,874 個	11,228 個
バイナリ演算子の誤用	“+”，“<<”	1,961 個	2,236 個
比較演算子の誤用	“==”，“<=”，“in”	8,640 個	8,555 個
論理演算子の誤用	“and”，“or”，“not”	556 個	3,137 個
代入演算子の誤用	“=”，“+=”，“|=”	999 個	1,395 個
リテラルの誤用	リテラル	10,789 個	8,052 個
合計		37,819 個	34,603 個

- 読み込んだ Python ファイル数（拡張子が “. py” であるファイル数）は 3,286,314 ファイルであり，1 問あたり平均 810 個の Python コードが提出されている．
- Python コードのうち，同一問題に対する同一ユーザの提出回数を表 4 に示す．約 35 % のユーザが複数のコードを提出している．
- Python コードのうち同一問題に対する同一ユーザの提出から，ステータスが “Accepted” 以外から “Accepted” に変化している組が 1,269,813 組検出された．1 問あたり平均 313 組の再提出が行われていると考えられる．

このプログラミングコンテストのデータから作成した実バグ修正データセットを，競プロ実バグ修正データセットと呼ぶ.

表 4: Project CodeNet における同一ユーザの提出回数

提出回数	ユーザ数	割合（値は切り捨て）
合計ユーザ数	1，716，162	100．000
1 ファイル	1，106，809	64．493
2 ファイル	288，544	16．813
3 ファイル	128，527	7．489
4 ファイル	68，805	4．009
5 ファイル	40，515	2．361
6 ファイル	25，344	1．477
7 ファイル	16，605	0．967
8 ファイル	10，958	0．638
9 ファイル	7，709	0．449
10 ファイル	5，398	0．314
11 ファイル	3，895	0．226
12 ファイル	2，923	0．170
13 ファイル	2，134	0．124
14 ファイル	1，616	0．094
15 ファイル	1，280	0．074
16 ファイル	983	0．057
17 ファイル	691	0．040
18 ファイル	574	0．033
19 ファイル	481	0．028
20 ファイル	374	0．021
21 ファイル以上	1，997	0．116

5　作成したデータセットを用いたモデルの性能比較

競プロ実バグ修正データセットと Richter らの研究 [11] で作成されたモデル，学習用プログラムを用いて機械学習を行い，次のモデルの性能を比較する．

- 人工バグ修正データのみを学習に用いるモデル
- 人工バグ修正データセットのみを学習に用いたモデルに，ファインチューニングとして GitHub 実バグ修正データセットをさらに学習したモデル
- 人工バグ修正データセットのみを学習に用いたモデルに，ファインチューニングとして競プロ実バグ修正データセットを混合して学習したモデル
- 人工バグ修正データセットのみを学習に用いたモデルに，ファインチューニングとして GitHub 実バグ修正データセットと競プロ実バグ修正データセットを混合したデータセットを学習したモデル

5.1　人工バグ修正データセットによる学習済みモデル

人工バグ修正データセットによる学習済みモデルは Richter らの研究 [11] で作成されたものを用いる．このモデルは 2.3.1 に挙げる 4 種類の単一ステートメントバグを含む人工バグを学習している．人工バグは 1 つの正しいコードに対する変異の回数が 5 回に設定されており，1 エポック 200,000 個の人工バグ修正データからなるデータを 300 エポック学習している．

5.2　評価手法

モデルの評価は実際のバグを含むソースコードに対するバグの検出・修正を行い，その正解率を計測することにより行う．

5.3　評価用テストデータ

Richter らの研究 [11] と同様，BUGLAB[2] の評価で用いられるテストデータを用いて評価を行う．ただし，このデータは手作業でバグ修正であることが確認された GitHub の URL とコミット SHA から構成されているため，2024 年 1 月時点でアクセスが可能であったデータのみを取得している．また 2.3.1 の 4 種類の単一ステートメントバグに含まれないもの，ソースコード中の適切な範囲の切り出しが難しいものなどを除き，1799 個の実際のバグ修正からなるテストデータを作成した．作成したテストデータに含まれるバグの種類の内訳を表 5 に示す．

表 5: テストデータ中のバグの種類

バグの種類	個数
変数の誤用	1,098 個
バイナリ演算子の誤用	73 個
比較演算子の誤用	361 個
論理演算子の誤用	178 個
代入演算子の誤用	35 個
リテラルの誤用	54 個
合計	1,799 個

5.4 評価基準

評価は次の 3 つの基準により行う.

- バグの位置の特定・修正トークンの出力の両方が正しく行われたかという，検出・修正の性能に対する評価
- バグの位置の特定が正しく行われたかという，検出の性能に対する評価
- バグの位置に対して正しい修正トークンを出力されたかという，修正の性能に対する評価

表 6: ファインチューニングによる性能変化

ファインチューニング	検出・修正率	検出率	修正率
なし (人工バグのみ)	19.01%	25.79%	54.64%
GitHub バグ	29.07%	37.24%	65.43%
競プロバグ	2.78%	3.11%	58.64%
競プロバグ・GitHub バグ	27.74%	33.07%	66.43%

6 評価

モデルを用いてバグの検出・修正を行った正解率を表 6 に示す．ただし，ファインチューニングを行っていない人工バグ修正データセットを用いた学習のみのモデル，GitHub 実バグ修正データセットを用いたファインチューニングのみのモデルは，Richter らの研究 [11] で公開されているモデルを再利用した．ただしバグ検出・修正の正解率は，本研究でファインチューニングを行ったモデルとの比較を行うため，同じテストデータを用いて再計測を行っている．

競プロ実バグ修正データを用いたファインチューニングにより，人工バグ修正データセットを用いた学習のみのモデルから検出率が著しく低下し，修正率は 4%向上している．修正率の向上幅は，GitHub 実バグ修正データセットをファインチューニングに用いた場合の約 9%の向上と比較して小さい．

また，競プロ実バグ修正データセットと GitHub 実バグ修正データセットを混合したファインチューニングを行ったモデルは，検出率の点で GitHub 実バグ修正データセットを用いたモデルと比較して向上幅は劣るが，修正率の点でより大きく性能が向上した．

7 考察

7.1 競プロ実バグ修正データと GitHub 実バグ修正データの比較

7.1.1 検出の性能変化

競プロ実バグ修正データをファインチューニングを用いることで検出の性能は低下した．競プロ実バグ修正データのみを用いた場合の検出率が著しく低下していることに加え，GitHub 実バグ修正データと合わせてファインチューニングに用いた場合も GitHub 実バグ修正データのみを用いた場合と比較して性能が低下していることから，競プロバグ修正はバグの検出性能改善のためのファインチューニングには適さないと考えられる．

GitHub 実バグ修正データに用いられるような実際のソフトウェア開発で記述されたソースコードと比較して，競技プログラミングで提出されたコードがバグの出現箇所の面で異なる性質を有している可能性がある．

7.1.2 修正の性能変化

表 5 に挙げるバグの種類別の，競プロ実バグ修正データを学習に用いたモデルの修正の性能変化を図 3 に，GitHub 実バグ修正データを学習に用いたモデルの性能変化を図 4 に示す．

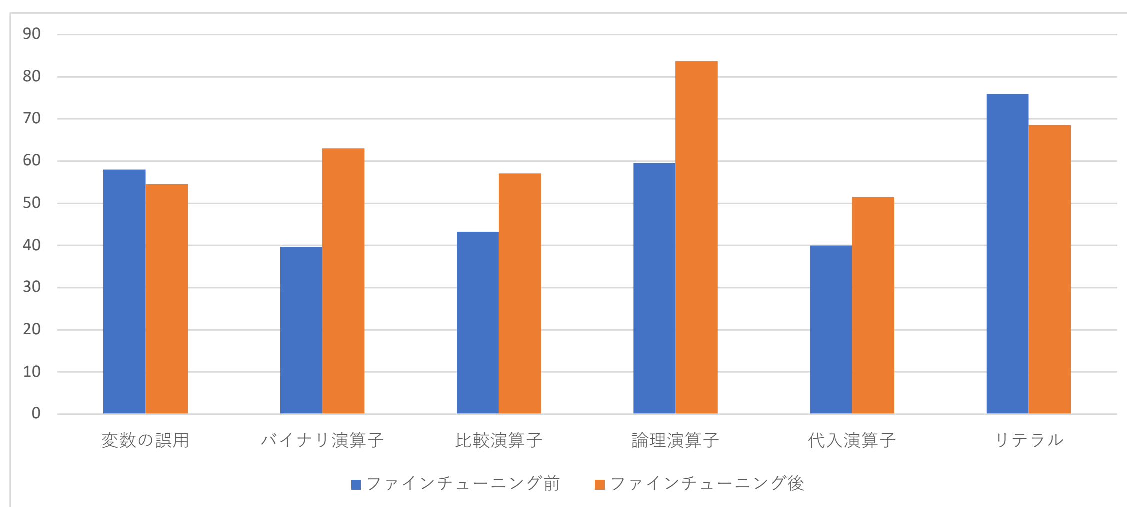

図 3: 競プロ実バグ修正データを用いたファインチューニングによる修正の性能変化

GitHub 実バグ修正データを用いたファインチューニングを行ったモデルが全てのバグの種類に対して性能の向上が見られることと比較して，競プロ実バグ修正データを用いた場合は変数の誤用，リテラルの誤用のバグに対する修正性能が低下している．このことから変数

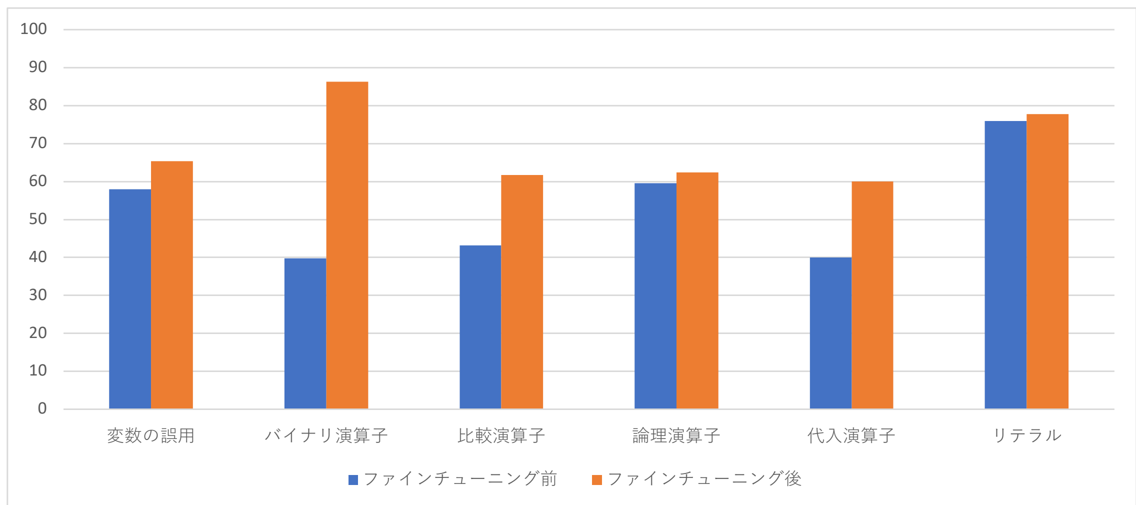

図 4: GitHub 実バグ修正データを用いたファインチューニングによる修正の性能変化

やリテラルの使用方法について，競技プログラミングコンテストの提出コードと実際のソースコードの間に差異があるのではないかと考えられる．

7.2　競プロ実バグ修正データと GitHub 実バグ修正データを混合したデータによるファインチューニング

競プロ実バグ修正データと GitHub 実バグ修正データを用いてファインチューニングを行ったモデルのバグの種類別の性能変化を図 5 に示す．

ファインチューニングを行う前の人工バグ修正を用いた学習のみのモデルに対して，すべての項目の性能が最大 49%向上している．また実バグ修正データをファインチューニングに用いた場合と比較しても向上幅は大きく，競プロバグ修正データを合わせてファインチューニングに用いることで修正性能がさらに改善すると考えられる．

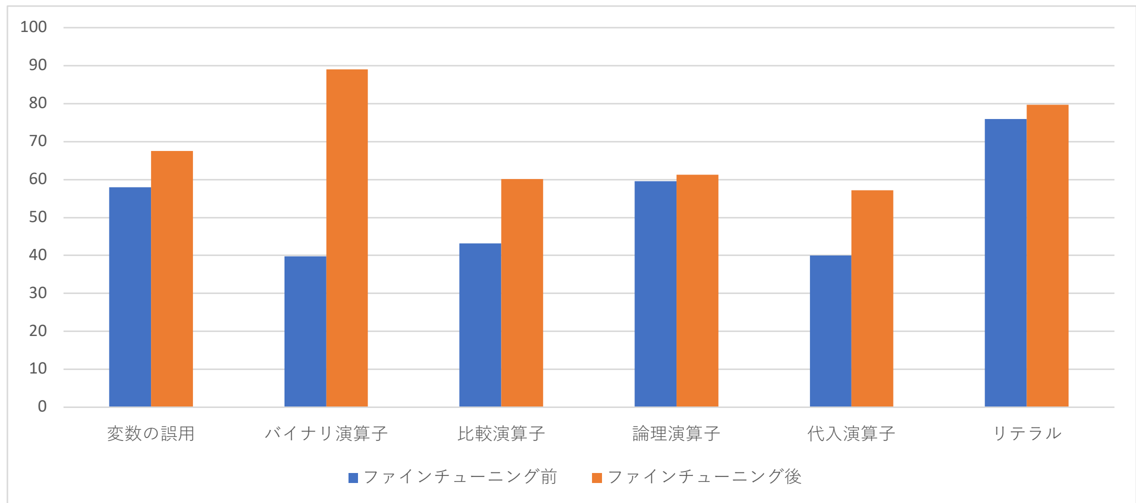

図 5: 混合実バグ修正データを用いたファインチューニングによる修正の性能変化

8 妥当性の脅威

8.1 内的妥当性への脅威

8.1.1 テストデータセットの評価

本研究においてモデルの性能評価に用いたテストデータは BUGLAB[2] より公開されているデータセットから作成しており，Richter らの研究 [11] で用いられたテストデータを再現している．Richter らの研究で用いられたテストデータは公開されていないが，同じモデルによる正解率の誤差の大きさにより適切に再現されているかを論ずる．

表 1 と表 6 より，同じモデルによる正解率は 2.4%〜2.5%低下している．誤差は僅かに生じているがモデルによる変化の幅は小さいため，テストデータが特定のモデルに対して有利となることはなく，適切に作成されていると考えられる．

8.1.2 データセットの形式について

本研究ではプログラミングコンテスト中のバグ修正のデータとして，提出されたソースコード全体を採用している．対して Richter らの研究 [11] では対象を関数定義のコードスニペットのみに限定している．プログラミングコンテストの提出コードは一般的な GitHub 上のコードと比較して短く，両者ともバグ検出・修正モデルの対象とする範囲内の長さではあるが，バグ検出の点で差異がありモデルの性能に影響を与えている場合が想定されるため，適切に検出率の性能変化の比較が行われていない可能性がある．

8.1.3 プログラミングコンテストの特徴について

プログラミングコンテストは一般的なソフトウェア開発におけるコーディングと比較して，時間制限やレーティングの条件などが科されるなどの違いがある．そのためバグの発生やその修正も異なる性質を有している可能性が考えられ，作成した競プロ実バグ修正データセットには実バグのデータして不適切なコードが含まれる可能性がある．

8.2 外的妥当性への脅威

8.2.1 データセット作成に用いたソースコードについて

本研究では Project CodeNet[10] に含まれる Python により記述されたコードのみを対象としている．プログラミングコンテストによる条件の違いや対象とする言語の差により，モデルの性能変化にも違いが生じる可能性がある．

9　おわりに

本研究ではプログラミングコンテストのデータから新しい実バグ修正データセットを作成し，バグ検出器の性能改善を試みた．結果，バグ検出器の修正性能のみに改善が見られ，検出性能は改善しなかった．バグ検出器の検出性能はプログラムの文脈に依存するものであり，プログラミングコンテストでのバグ修正と人工バグ修正の傾向の差異が検出性能に影響している可能性が高いと考えられる．

一方，バグ検出器の修正性能はバグの種類によって改善が見られた．プログラミングコンテストの提出コードと一般的なソースコードの違いや特徴について分析を深め，バグの種類に応じて適切なファインチューニングを行うことでさらなるバグ修正の性能改善を行うことができると考えられる．

また，プログラミングコンテストのデータを用いた実バグ修正データは GitHub のスクレイピングなどほかの手法と比較してデータ拡張が容易である．プログラミングコンテストの提出コード数は多く，用いられるプログラミング言語も多様であるため，今回対象とした Python 以外の言語への適用も可能である．これらのデータセットを有効に用いるファインチューニングの方法について調査を行い，バグ検出器のさらなる性能の改善を目指したい．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後 芳樹 教授には，研究活動に対して多くの御指導・御助言を賜りました．

松下 誠 准教授には，本研究を通して様々な御指導・御助言と共に，日頃より御声援の御言葉を賜りました．また研究の原案から方針，研究の進め方について何度も相談に乗っていただきました．心より深く感謝の意を表します．

神田 哲也 助教には，研究を進めるにあたり，研究を行う上での基礎から具体的な御助言に至るまで幅広く御指導を賜りました．慣れない研究活動に不安を抱く中，大変大きな支えとなっていただきました．心より深く感謝申し上げます．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 川渕 皓太 氏には，研究を行う上での疑問について幾度となく相談に乗っていただき，また，精神面での支えとなっていただきました．深く感謝申し上げます．

また大阪大学情報科学研究科コンピュータサイエンス専攻の 田畑 彰洋 氏， 岸本 理央 氏，音田 渉 氏には本研究を進めるにあたり，情報技術に関する知識やスキルの面で多くのお力添えをいただきました．

最後に，様々な御指導・御助言を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻肥後研究室の皆様ならびに事務職員 軽部 瑞穂 氏に心より深く感謝いたします．

参考文献

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs, 2018.

[2] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. In *NeurIPS*, 2021.

[3] AtCoder 株式会社. 日本最大のプログラミングコンテストサイト atcoder 全世界での登録者数が 50 万人を突破！ `https://prtimes.jp/main/html/rd/p/000000038.000028415.html/`, May. 2023. Accessed: May, 2023.

[4] Aram Ebtekar and Paul Liu. An elo-like system for massive multiplayer competitions. *eprint arXiv:2101.00400*, Jan. 2021. DOI:10.48550/arXiv.2101.00400.

[5] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, p. 317–328, New York, NY, USA, 2018. Association for Computing Machinery.

[6] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, pp. 4–12, 12 2001. DOI:10.1147/sj.411.0004.

[7] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[8] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur?: The manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR ’20. ACM, June 2020.

[9] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, Vol. 18, No. 5, pp. 423–433, 05 1992. 著作権 - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) May 1992; 最後にアップデートされた日 - 2023-11-25; CODEN - IESEDJ.

[10] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[11] C. Richter and H. Wehrheim. How to train your neural bug detector: Artificial vs real bugs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1036–1048, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society. DOI:10.1109/ASE56229.2023.00104.

[12] W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent catastrophic accidents: Investigating how software was responsible. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pp. 14–22, 2010.

[13] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707–740, 2016.

[14] 有隆渡部. オンラインジャッジの開発と運用 -aizu online judge-. 情報処理, Vol. 56, No. 10, pp. 998–1005, 09 2015.