自動テスト生成技術を利用した 機能等価メソッドデータセットの構築

肥後 芳樹[1,a)]

概要：現代の高水準プログラミング言語は多種多様な文法を持ち，必要な機能をさまざまな方法で実装できる．ソースコードが公開されているオープンソースソフトウェアにおいても同じ機能を異なる方法で実装したコードが大量に存在しており，それらを集めることでソフトウェア工学のさまざまな研究に役立つデータセットを構築できると著者らは考えた．本研究では，約 3 億 1,400 万行の Java のソースコードから，機能が同じであるが構造が異なるメソッドのペアのデータセットを構築する．このデータセットを構築するために，テスト自動生成技術を利用した．テスト自動生成技術は，対象メソッドに対して成功するテストケースを生成する．この性質を利用して，2 つのメソッドから生成されたテストケースを相互に実行することにより，その 2 つのメソッドの振る舞いがある程度同じかどうかを自動的に判定した．自動判定を通過したメソッドのペアは，手作業により機能の等価性を判定した．また，本論文では，構築したデータセットを利用してコードクローン検出ツールの性能評価を行った結果についても報告する．構築されたデータセットは GitHub (https://github.com/YoshikiHigo/FEMPDataset) で公開されている．

Constructing Dataset of Functionally Equivalent Java Methods Using Automated Test Generation Techniques

1. はじめに

現在利用されているプログラミング言語は豊富な文法を持ち，開発者が必要な機能を実装する方法は幾通りも存在する．例えば，Java の場合では繰り返し処理を行う場合に for 文，while 文，再帰関数，Stream 等を利用できる．また，Fowler が提唱したリファクタリングパターン [7] では，リファクタリングの適用前と適用後の実装は同じ外部的振る舞いを持つため，リファクタリングは機能の実装変更と見なすことができる．このように，ある機能を実装する方法は無数にあり，開発者は自身の好みやソフトウェア開発プロジェクトの方針に従って，必要な機能を実装する．

著者らは，オープンソースソフトウェアにおいて同じ機能を有するが異なる構造を持つソースコード（以降，同機能異構造コード）が大量に存在しており，それらはソフトウェア工学のさまざまな研究において有用なデータセットになり得ると考えている．例えば、同機能異構造コードはコードクローン検出ツールの評価に利用できる．同機能を実装したコードはコードクローンとして検出されることが望ましいので，同機能異構造コードがどの程度コードクローンとして検出されるかを調べることで，コードクローン検出ツールの性能を評価できる．また、同機能異構造コードを用いることで、メモリ使用量や実行速度などの性能面でどのような実装が優れているか、ISO/IEC 25010 [9] のようなソフトウェア品質面でどのような実装が優れているかを調査できる．

しかし，同機能異構造コードを収集することは容易ではない．オープンソースソフトウェアのソースコードから手作業でそのようなコードを見つけ出すことは現実的ではないし，既存のコードクローン検出ツールを使ってしまうと，既存のコードクローン検出技術で検出可能な同機能異構造コードのみが検出されてしまい，上記で述べたようなコードクローン検出技術の評価に利用できない．

そこで，本研究では，収集対象とする同機能異構造コードを，同じ入力（引数）を与えたときに同じ出力（返り値）を返すメソッド（以降，機能等価メソッド）のペアに限定する．同機能異構造コードの検出を機能等価メソッドの検出

[1] 大阪大学
Osaka University, Suita, Osaka 565-0879, Japan
[a)] higo@ist.osaka-u.ac.jp

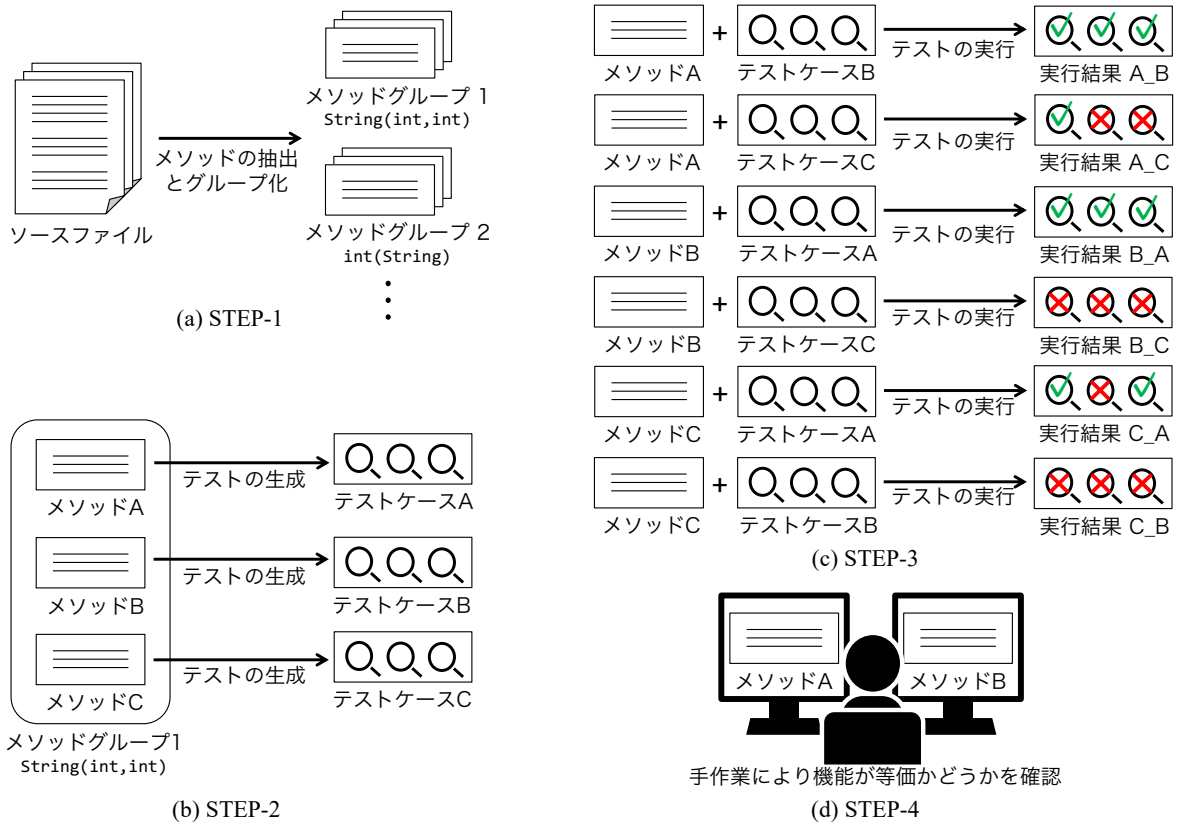


図 1: 機能等価な Java メソッドのペアを得るための手順

に限定することにより，テスト自動生成技術を用いて機能等価メソッドのペアの候補を自動的に取得することが本研究のキーアイデアである．取得した機能等価メソッドペアの候補は手作業により真に機能等価であるかを確認した．

本研究では，Java の機能等価メソッドペアの検出対象として，IJADataset [4] を選択した．このデータセットは，約 274 万のソースファイル（合計約 3 億 1,400 万行のソースコード）からなり，約 2,300 万のメソッドが含まれている．このデータセットから機能等価メソッドのペアの候補を 13,710 件自動的に検出し，そのうちの 2,194 件を目視により確認した．その結果，1,342 件の機能等価メソッドペアのデータセットを構築した．また，このデータセットには，目視により機能等価ではないと判定された 852 のメソッドペアも含まれている．構築したデータセットは GitHub で公開されている[*1]．

また構築したデータセットを利用してコードクローン検出ツールを評価した．その結果，字句解析に基づく検出手法では検出できない機能等価メソッドペアが多く存在すること，および，抽象構文木と深層学習に基づく検出手法では機能等価でないメソッドペアを誤検出してしまう傾向が強いことがわかった．

[*1] https://github.com/YoshikiHigo/FEMPDataset

2. 機能等価メソッドの候補を自動的に特定するためのキーアイデア

本研究では，Java メソッドの静的な特徴および動的な振る舞いを利用して，機能的に等価なメソッドのペアの候補を自動的に収集する．得られたメソッドペアの候補が真に機能等価であるかは目視により判定する．そのため，機能等価メソッドペアの候補をできるだけ自動的に，かつ多く集めることが重要である．

本研究で使用する Java メソッドの静的な特徴は、返り値の型と引数の型である．機能等価なメソッドペアの候補を得るための第一段階として，返り値の型と引数の型が等しいメソッドを同じグループに割り当てる．

本研究で使用する Java メソッドの動的な振る舞いは，与えられたテストケースの実行結果（成功／失敗）である．同じグループ内の全てのメソッドからテスト自動生成技術を用いてテストケースを生成する．自動テスト生成技術は，生成対象のメソッドに対して成功するテストケースを生成する．そして，メソッドの全てのペアに対して，生成したテストケースを相互に実行することでそのメソッドペアが同じ振る舞いを持つかどうかを自動的に判定する．メソッド A から生成された全てのテストをメソッド B が成功し，メソッド B から生成された全てのテストをメソッド A が成功すれば，メソッド A とメソッド B の振る舞いは

ある程度同等であり，その機能が等価である可能性があるというのが本研究のキーアイデアである．

このキーアイデアに基づき，大量のオープンソースソフトウェアから、相互テスト実行に成功したメソッドのペアを自動的に取得し，それらを目視で確認することにより機能等価なメソッドペアのデータセットを構築する．ソースコードが同一もしくは類似したメソッドも機能的には等価である可能性があるが，そのようなメソッドは既存のコードクローン検出ツールで検出できるため [8]，本研究ではソースコードが類似していない機能等価なメソッドペアのデータセットの構築を目的とする．

3. データセットの構築手順

本研究では，以下の手順で機能等価なメソッドペアのデータセットを構築する．

STEP-1 対象プロジェクトに含まれるメソッドの分類.

STEP-2 各メソッドからテストケースの生成.

STEP-3 テストを相互実行することにより機能等価メソッドペアの候補を取得.

STEP-4 機能等価メソッドペアの各候補のソースコードを閲覧し，真に機能等価であるかを判定.

図 1 は上記の構築手順の概要を表している．STEP-1 から STEP-3 は著者らが作成したツールを用いて自動的な処理として実行され，STEP-4 のみが手動で行われる．以下，各 STEP の詳細について述べる．

3.1 STEP-1

STEP-1 では，対象プロジェクトのソースコードを解析してメソッドを抽出し，抽出したメソッドを返り値の型と引数の型に基づいて分類する．メソッドの抽出においては，各メソッドについて以下の情報を取得し，データベースに登録する．

- メソッド名,
- 返り値の型と引数の型,
- (原型のままの) ソースコード,
- 正規化されたソースコード,
- 文の数および条件式の数,
- ファイルパス,
- 開始行および終了行.

正規化では，全ての変数名が特殊な名前に変更される．正規化の例を図 2(b) に示す．この正規化を行うことにより，同じ構造を持つが変数のみが異なるメソッド群を 1 つのメソッドとして扱えるようになる．これにより，STEP-2 以降で同じ構造を持つが変数のみが異なる各メソッドを個別に扱う必要がなくなり，より効率的に処理を行える．

対象プロジェクト内に存在する全てのメソッドが抽出されるわけではない．本研究では，以下の特徴を持つメソッドは抽出対象とはしない.

```
   1  package com.intellij.openapi.util.text;
   …
  33  public class StringUtil extends StringUtilRt {
   …
1265    @Contract(pure = true)
1266    public static @NotNull String repeat(@NotNull String s, int count) {
1267      if (count == 0) return "";
1268      assert count >= 0 : count;
1269      StringBuilder sb = new StringBuilder(s.length() * count);
1270      for (int i = 0; i < count; i++) {
1271        sb.append(s);
1272      }
1273      return sb.toString();
1274    }
   …
```

(a) 元のソースコード

```
String $method(String $variable,int $variable){
  if ($variable == 0)   return "";
  assert $variable >= 0 : $variable;
  StringBuilder $variable=new StringBuilder($variable.length() * $variable);
  for (int $variable=0; $variable < $variable; $variable++) {
    $variable.append($variable);
  }
  return $variable.toString();
}
```

(b) 正規化されたソースコード

```
  1  import java.util.*;
  2  public class Target {
  3    String __target__(String s,int count){
  4      if (count == 0)   return "";
  5      assert count >= 0 : count;
  6      StringBuilder sb=new StringBuilder(s.length() * count);
  7      for (int i=0; i < count; i++) {
  8        sb.append(s);
  9      }
 10      return sb.toString();
 11    }
 12  }
```

(c) 単一ファイルへと切り出されたソースコード

図 2: 対象メソッドのソースコードに対する処理の例

- java.lang および java.util パッケージ以外の参照型を返り値，引数およびメソッド本体に含むメソッド.
- 返り値が void であるメソッド.
- 単一のプログラム文のみを含むメソッド.

1 つ目の条件を用いる理由は，Java では java.lang パッケージ以外の参照型を使う場合，ソースファイルの冒頭に import 文を記述する，もしくはその参照型を完全修飾名で記述する必要があるからである．また，標準の Java ライブラリに含まれない参照型の場合は，そのクラスファイル（jar ファイル）を用意する必要があり，コンパイルの事前準備が必要になる．java.util パッケージに含まれる参照型を含んでいても抽出対象のメソッドとした理由は，java.util.List や java.util.Set など頻繁に使われる型がこのパッケージ内に多く含まれているためである．これらの参照型も取り扱うことで，抽出できるメソッドの数は飛躍的に増加すると著者らは考えた.

2 つ目の条件を用いる理由は，戻り値が void であるメソッドはメソッド内のどの値がメソッドの最終的な計算結果であるかを自動的に判断することが難しいからである．返り値が void でない場合は，メソッドの返り値がメソッドの計算の最終結果と判断できる.

3 つ目の条件を用いる理由は，Java のソースコードには単一のプログラム文のみをもつセッターやゲッターが多数

含まれており，それらは同機能異構造コードの検出対象としては不適切だからである．

抽出されたメソッドの分類は，メソッドの返り値の型と引数の型に基づいて行われる．返り値の型と引数の型が完全に同じメソッドが同じグループに分類される．全てのメソッドの分類が終了した後に，各グループ内に正規化されたソースコードが完全に一致するメソッドが複数ある場合はそのうちの 1 つだけがグループ内に保持される．その理由は，同じ実装のメソッドは同じ振る舞いを持つことが明らかであり，そのような同じ実装を持つメソッドペアの検出はこの研究の目的にはそぐわないからである．なお，単一のメソッドのみからなるグループは，STEP-2 以降の処理の対象にはならない．

3.2 STEP-2

STEP-2 では，各メソッドを 1 つのファイルに切り出してそのテストケースを生成する．メソッドをファイルに切り出す際に，以下の処理が行われる．

- ファイルの先頭に ‘`import java.util.*;`’ を挿入する．これは，対象となるメソッドで `java.util` パッケージのクラスが使われている場合でもコンパイル可能にするための処理である．
- 対象となるメソッドをクラス宣言で囲む．現在は ‘`Target`’ というクラス名を使用している．また，対象メソッドの名前を ‘`__target__`’ に変更する．これは，著者らが作成したスクリプトにおいて全ての対象メソッドを統一的に扱うためである．
- 対象メソッドのシグネチャからアノテーションと ‘`static`’ を削除する．これも，スクリプトにおいて全ての対象メソッドを統一的に扱うための処理である．

図 2(a) のソースコードから得られたメソッドを 1 つのファイルとして切り出したあとのソースコードを図 2(c) に示す．この図から，切り出されたファイルには対象のメソッドのみが含まれ、クラス名とメソッド名が統一され、メソッドシグネチャに付いていたアノテーションと `static` 修飾子が削除されていることがわかる．

次に，ファイルに切り出した各メソッド対してテストケースを生成する．テストケースの生成には `EvoSuite` [6] を用いているが，`Randoop` [14]，`Agitar` [1] など他のテスト生成ツールも使用可能である．本実験では，5 つ以上のテストケースが生成できた場合に，そのメソッドを STEP-3 で用いた．この理由は，対象メソッドの構造によっては，テスト生成ツールが十分にテストケースを生成できないからである [20]．十分にテストケースが生成できなかった場合にそのテストを用いて機能等価なメソッドペアの候補の検出を行うには不適切であると著者らは考えた．

3.3 STEP-3

STEP-3 では，同じグループに属し，テストケースの生成に成功した各メソッドに対して，相互にテストを実行する．図 1(c) では，`Method-A`, `Method-B`, `Method-C` の 3 つのメソッドに対してテストを相互に実行している．`Method-A` は `Method-B` から生成された全てのテストに成功し，`Method-B` も `Method-A` から生成された全てのテストに成功している．したがって，`Method-A` と `Method-B` は，機能等価なメソッドペアの候補となる．`Method-C` は `Method-A` から生成されたテストの 1 つを成功しておらず，`Method-B` から生成された全てのテストに成功していない．したがって，`Method-A` と `Method-C` のペア，および `Method-B` と `Method-C` のペアは，機能等価なメソッドペアの候補にはならない．

3.4 STEP-4

STEP-4 では，STEP-3 で得られた機能等価なメソッドペアの候補のソースコードを目視で確認し，真に同じ振る舞いを持つメソッドのペアかどうかを確認する．

4. FEMP データセット

ここでは，構築したデータセット（以降，FEMP データセット[*2]と呼ぶ）について述べる．FEMP データセットは，`IJADataset` [4] に含まれるソースコードに対して構築された．`IJADataset` は約 274 万のソースファイル，合計 3 億 1,400 万行のソースコードからなり，約 2,300 万のメソッドを含んでいる．STEP-1 では，257,012 のメソッドが抽出され，それらは 14,030 のグループに分類された．STEP-2 では、27,759 のメソッドから 5 つ以上のテストケースを生成することに成功した．STEP-3 では、13,710 の機能等価なメソッドペアの候補を得た．

STEP-4 の目視確認は大阪大学大学院情報科学研究科に所属する 3 名の博士前期課程の学生によって行われた．3 名の学生は Java を用いたプログラミングの経験を有する．まず 3 名の学生は個別に各候補ペアのソースコードを閲覧し，機能等価か否かを判断した．次に，3 名の評価が分かれた各候補ペアについて，3 名で議論を行うことにより，機能等価か否かを判断した．

しかしながら，STEP-3 で得た 13,710 の全てのメソッドペアを目視確認することは現実的ではないため，以下の手順により，その一部を抽出した．

(1) 目視対象メソッドペア P，および P に含まれるメソッドの集合 M を空集合で初期化する．
(2) 13,710 のメソッドペアを ID[*3]の昇順に並べ，下記の処理を順に行う．
   - そのメソッドペアを構成する双方のメソッドが M に

[*2] Functionally Equivalent Method Pair Dataset
[*3] STEP-3 で得られた 13,710 のメソッドペアは，一意の整数値をその ID として持つ．

```
double min(double y0, double y1, double y2){
  if ((y0 <= y1) && (y0 <= y2)) {
    return y0;
  } else if (y1 <= y2) {
    return y1;
  } else {
    return y2;
  }
}
```

(a) メソッド min

```
double minimum(double val1, double val2, double val3){
  if (val1 < val2) {
    return val1 < val3 ? val1 : val3;
  } else {
    return val2 < val3 ? val2 : val3;
  }
}
```

(b) メソッド minimum

図 3: 機能等価なメソッドペアの例

含まれない場合は，そのメソッドペアを P に追加し，その双方のメソッドを M に追加する．

- そのメソッドペアを構成する少なくとも 1 つのメソッドが M に含まれる場合は，何もしない．

上記の手順終了後に，2,194 のメソッドペアが P に含まれていた．この 2,194 のメソッドペアは全て異なるメソッドから構成されている．この 2,194 のメソッドペアに対して，各学生はそれぞれ 44 時間 48 分，33 時間 19 分，43 時間 25 分を要して機能等価か否かを判定した．その後に，評価が分かれた各メソッドペアに対する議論では 9 時間 28 分を要した．

STEP-4 の結果，2,194 のメソッドペアのうち，1,342 のペアが機能等価であり，残りの 852 が機能等価ではないと判定された．なお，3 名の学生の評価が分かれ，議論されたメソッドペアの数は 296 だった．このデータセットは GitHub で公開されている[*4]．

図 3 は STEP-4 で機能等価と判定されたメソッドペアの例である．どちらのメソッドも引数で与えられた 3 つの double 型の数値のうち，最も小さい値を返す機能が実装されている．メソッド min はその機能が if 文のみで，メソッド minimum では if 文と三項演算子を利用して実装されている．一方，図 4 は STEP-4 で機能が等価ではないと判定されたメソッドペアの例である．どちらのメソッドも引数で与えられた 2 つの二次元配列の等価性を判定している．メソッド ArrayEquals は空配列が与えられた場合に false を返すのに対して，メソッド equals は空配列が与えれた場合に true を返すという点で機能が異なる．EvoSuite はメソッド ArrayEquals に対しては 9 つのテストケースを，メソッド equals に対しては 8 つのテストケースを生成していたが，この 2 つのメソッドの機能差を見つけるテストケースは生成されなかった[*5]．

[*4] https://github.com/YoshikiHigo/FEMPDataset

[*5] FEMP データセットには EvoSuite が生成したテストケースも含まれている．

```
boolean ArrayEquals(boolean[][] a,boolean[][] b){
  if (a.length < 1)   return false;
  if (b.length < 1)   return false;
  if (a.length != b.length)   return false;
  for (int y=0; y < a.length; y++) {
    if (a[y].length != b[y].length)     return false;
    for (int x=0; x < a[y].length; x++) {
      if (a[y][x] != b[y][x])       return false;
    }
  }
  return true;
}
```

(a) メソッド ArrayEquals

```
boolean equals(boolean[][] m1,boolean[][] m2){
  if (m1.length != m2.length)   return false;
  for (int i=0; i < m1.length; i++) {
    if (m1[i].length != m2[i].length)     return false;
    for (int j=0; j < m1[i].length; j++) {
      boolean b1=m1[i][j];
      boolean b2=m2[i][j];
      if (b1 != b2)       return false;
    }
  }
  return true;
}
```

(b) メソッド equals

図 4: 機能等価でないメソッドペアの例

5. クローン検出ツールの精度評価

ここでは，作成した FEMP データセットの活用例として，クローン検出ツールの精度評価を行った結果を述べる．この評価では，STEP-4 によって機能等価と判定された 1,342 のメソッドペアをクローンペアとして検出すべき，機能が等価ではないと判定された 852 のメソッドペアをクローンペアとして検出すべきではない，とした．以降，前者の集合を EMP ($Equivalent\ Method\ Pairs$)，後者の集合を IMP ($Inequivalent\ Method\ Pairs$) と呼ぶ．

5.1 評価対象のクローン検出ツール

この評価では NIL [12], InferCode [5], および ASTNN [18] の 3 つのクローン検出ツールを対象とした．

NIL は，対象ソースコードから得た字句列の N-gram と転置インデックスを用いてクローンの候補となりうるメソッドのペアを高速に特定する．そしてそれらの候補に対して最長共通部分列アルゴリズムを適用することにより，クローンペアかどうかを判定する．NIL は従来のクローン検出技術では検出が難しかった large-variance クローンの検出を目的としたツールである．2 つのオープンソースソフトウェアのソースコードを対象にして行われた既存のクローン検出ツール LVMapper [17] および CCAligner [16] との比較では，NIL の large-variance クローンの検出数が 354 と 398（適合率は 86%および 88%）だったのに対して，LVMapper の検出数は 355 および 389（適合率は 64%および 60%），CCAligner の検出数は 184 および 284（適

合率は 43%および 49%）だった．

InferCode は，抽象構文木と木構造に基づいた畳み込みニューラルネットワーク [11] を利用した事前学習モデルである．ソースコードのクラスタリングなどの教師無し学習タスクや，ソースコード分類などの教師有り学習タスクに利用できる．Bui らは InferCode におけるクローン検出を教師無し学習タスクとして実装している．本研究でも，教師無し学習に基づくクローン検出手法として，InferCode を利用する．クローン検出では，2 つのメソッドの類似度として，コサイン類似度を使用する．Bui らは，クローンペアとしてみなすコサイン類似度を閾値を 0.8 に設定し，BigCloneBench [2] と OJClone [19] を対象として実験を行った．BigCloneBench に対しては，適合率が 90%，再現率が 56%，OJClone に対しては，適合率が 61%，再現率が 70%だった．

ASTNN は，抽象構文木と回帰型ニューラルネットワークに基づいたモデルである．対象ソースコードに含まれる字句情報や命令文単位での構文情報を学習できる．クローン検出においては，ASTNN は 0 以上 1 以下の値を出力する．この値が閾値以上の場合にクローンとみなされる．Zhang らは 0.5 を閾値として，BigCloneBench と OJClone に対して実験を行った．彼らは，OJClone に対しては適合率が 98.9%，再現率が 92.7%，また BigCloneBench に対しては，クローン分類毎に検出性能を評価し，Weakly Type-3/Type-4[*6] では，適合率が 99.8%，再現率が 88.3%だったと報告している．

5.2 検出方法

NIL

NIL の GitHub[*7]にある説明に従い，NIL をインストールした．*EMP* および *IMP* に含まれる各メソッドを個別にファイルとして出力し，NIL にその 2 つのファイルをクローン検出対象として与えることにより，クローン検出を行った．つまり，NIL を 2,194 回（*EMP* に対して 1,342 回，*IMP* に対して 852 回）実行した．サイズの小さいメソッドも存在するため，クローンとして検出するメソッドの最小行数と最小字句数はどちらも 1 にして NIL を実行した[*8]．

InferCode

InferCode の GitHub[*9]にある説明に従い，InferCode をインストールした．InferCode を利用して *EMP* および *IMP* に含まれる各メソッドのベクトルデータを取得し，メソッドペアのコサイン類似度の閾値を 0 から 1 まで 0.001 ずつ変化させながらクローンを検出した．

ASTNN

ASTNN の GitHub[*10]にある説明に従い，ASTNN の事前学習モデルを取得した．次に，*EMP* に含まれる 1,342 のメソッドペアを 10 のブロックに分けた．ファインチューニング，検証，テストのデータの割合が 8∶1∶1 になるようにブロックを分類し，全てのブロックが一度テストデータになるように ASTNN を用いたクローン検出を行った．ASTNN の出力は 0 以上 1 以下の数値であるため，クローンとする閾値を 0 から 1 まで 0.001 ずつ変化させてその結果を評価した[*11]

5.3 検出結果の評価尺度

この評価では $recall^E(t)$, $recall^I(t)$, $accuracy(t)$ の 3 つの尺度を利用して，ツール t のクローン検出結果を評価した．まず，これらの評価尺度で利用する $EMP(t)$ および $IMP(t)$ の定義を以下に示す．

$EMP(t)$　*EMP* に含まれるメソッドペアのうち，ツール t によって正しく機能等価と判定された（クローンとして検出された）メソッドペアの集合

$IMP(t)$　*IMP* に含まれるメソッドペアのうち，ツール t によって正しく機能が等価でないと判定された（クローンとして検出されなかった）メソッドペアの集合

上記の定義を利用して，$accuracy(t)$, $recall^E(t)$, $recall^I(t)$ を下記のように定義する．なお $|A|$ は集合 A の要素数を表す．

$$recall^E(t) = \frac{|EMP(t)|}{|EMP|}$$

$$recall^I(t) = \frac{|IMP(t)|}{|IMP|}$$

$$accuracy(t) = \frac{|EMP(t)| + |IMP(t)|}{|EMP| + |IMP|}$$

5.4 検出結果

NIL

NIL の検出結果を表 1 に示す．$recall^E(NIL)$ は 34.5%(=463/1,342)，$recall^I(NIL)$ は 72.54%(=618/852)，$accuracy(NIL)$ は 49.27%(=1,081/2,194) であった．以上の結果から，NIL によるクローン検出は機能等価なメソッ

表 1: NIL の検出結果

		検出結果	
		EMP	*IMP*
真値	*EMP*	463	879
	IMP	234	618

[*6] 構文的な類似度が 50%以下のクローン [3].

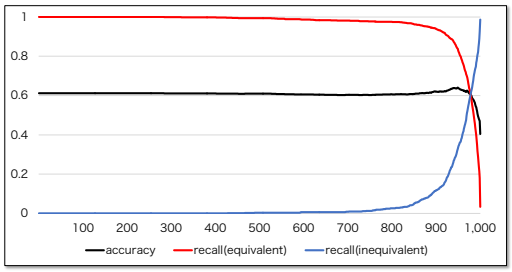
[*7] https://github.com/kusumotolab/NIL

[*8] ただし，NIL は内部で N=5 とした N-gram を用いた処理をしているため，クローンとして検出されるためには最小で 5 つの字句が必要．

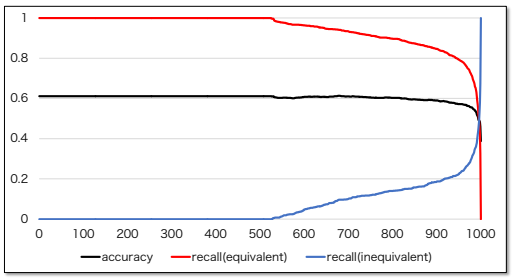
[*9] https://github.com/bdqnghi/infercode

[*10] https://github.com/zhangj111/astnn

[*11] ファインチューニング時に利用したハイパーパラメータは以下の通り．batch size:32, epoche:5, learning size:2e-3, verctor size of word2vec:128, hidden dimension:100, encode dimension:128.



(a) InferCode



(b) ASTNN

図 5: 検出の精度と閾値の関係．横軸の左端は閾値 0，右端は閾値が 1 を表す．

ドの発見には漏れが多く，また誤検出もある程度含まれることが分かった．

InferCode

InferCode の評価結果を図 5(a) に示す．閾値が 0.557 以下の場合において，$recall^E(InferCode)$ は 99%以上となるが，このときに $recall^I(InferCode)$ は 0.35%以下しかない．つまりほぼ全ての機能等価メソッドペアをクローンとして検出できているが，機能等価ではないメソッドのペアもほぼ全てクローンとして検出してしまっている．閾値を高くすると $recall^I(InferCode)$ は改善するが，同時に $recall^E(InferCode)$ は悪化してしまう．閾値 0.949 のときに $accuracy(InferCode)$ の値は最も高く，64.04%となった．このとき，$recall^E(InferCode)$ は 83.83%，$recall^I(InferCode)$ は 32.86%だった．

ASTNN

ASTNN の評価結果を図 5(b) に示す．全体的に ASTNN のグラフの形状は InferCode と似ている．閾値は 0.531 以下の場合において，$recall^E(ASTNN)$ は 99%以上となるが，このときに $recall^I(ASTNN)$ は 0.59%しかない．閾値 0.677~0.681 のときに $arrucary(ASTNN)$ は最も高い値 61.30%となり，このときの $recall^E(ASTNN)$ と $recall^I(ASTNN)$ の値はそれぞれ 94.19%と 9.62%だった．

まとめ

字句の並びに基づくクローン検出ツール NIL は，機能等価メソッドペアの多くをクローンとして検出できず，機能等価メソッドの検出能力が高いとはいえないことがわかった．また，抽象構文木と深層学習に基づくクローン検出ツールである InferCode と ASTNN は，機能等価メソッドペアをクローンとして検出できるが，機能等価ではないメソッドについてもクローンとして検出してしまうことがわかった．以上のことから，機能等価メソッドペアを適切に見つけるためには新しい手法の開発が必要であるといえる．

6. 関連研究

Svajlenko らは BigCloneBench データセットを構築した [15]．BigCloneBench は類似したメソッドを集めたデータセットであり，メソッドの機能別に 45 のグループに分類されている．データセット構築手順の最後に人手による確認作業も行われている．しかし，人手による確認作業の対象はキーワードとコードパターンにより発見された Java メソッドである．そのため，それらにより発見されなかった機能的に等価なメソッドは，BigCloneBench データセットには含まれない．また，データセットの構築過程では対象のメソッドは実行されておらず，動的な観点からの機能等価の確認作業は行われていない．

Liu らは，過去の競技プログラミングのデータを用いて機能的に等価なプログラムのデータセットを構築した．彼らは約 5,000 の問題に対して機能的に等価なプログラムを収集している [10]．このデータセットでは，ある問題に対する複数のユーザの回答プログラムが機能的に等価なプログラムとして扱われている．Zhao らは Google Code Jam で同じ機能を持つプログラムのデータセットを公開しており [19]，Mou らもプログラミング教育支援システムに提出されたプログラムのデータセットを公開している [11]．一方，我々のデータセットは彼らのデータセットと異なり，競技プログラミングのプログラムではなく，OSS に含まれる機能的に等価な機能を持つメソッドである．

Rabin らは，与えられたプログラムの構造を変化させるツール ProgramTransformer を開発した [13]．ProgramTransformer はプログラムの構造を変化させるためのルールをいくつか所有しており，そのルールに基づいて構造を変更する．例えば，for 文で記述された繰り返しを while 文に書き換えたり，変数名を変更したりという変更を自動で行う．

7. おわりに

本研究では，オープンソースソフトウェアから抽出したメソッドに対してテストケースを自動生成し，生成したテストを互いに実行することで、機能的に等価なメソッドペアを特定した．IJADataset データセットに含まれる 3 億 1,400 万行の Java のソースコードから，機能等価なメソッドペアの抽出を行った．その結果，13,710 の機能等価なメソッドペアの候補を得た．このうち，2,194 のメソッドペアを目視確認し，1,342 の機能等価なメソッドペアを得た．また，本研究では，このデータセットを利用してコードクローン検出ツールの性能評価も行った．このデータセット

は，コードの保守性や理解容易性などのコードの品質検討にも利用できる．

現在の課題は，返り値の型とパラメータの型が等しく，テストが生成できたメソッドの全ての組み合わせに対して互いのテストするため，機能等価なメソッドペアの候補を見つけるのに非常に長い時間を要することである．今回の実験では，STEP-3 のみで約 40 日を要した．今後は，より早く機能的に等価なメソッドペアの候補を検出できるよう，相互実行するメソッドの組合せを減らすヒューリスティックを導入する予定である．

また，目視調査で機能的に同等でないと判断された手法群の割合が大きい（852/2,194=38.8 %）ことについても，改善を考慮する必要がある．もし，目視確認によりそのような機能的に等価でないと判定されるメソッドのペアがほとんどないのであれば（テストケースの相互実行により機能が等価でないメソッドがほとんどふるい落とされるのであれば），目視確認の必要がなくなり，巨大なデータセットをほぼ自動的に作成できるようになる．こちらについては，EvoSuite のテスト生成アルゴリズムを改善することにより，より品質の高いテスト自動生成を目指す．

参考文献

[1] : AgitarOne, http://www.agitar.com/solutions/products/agitarone.html.

[2] : BigCloneBench, https://github.com/clonebench/BigCloneBench.

[3] : BigCloneEval, https://github.com/jeffsvajlenko/BigCloneEval.

[4] : IJaDataset 2.0, https://1drv.ms/u/s!AhXbM6MKt_yLj_N3FAIGw3CJb1JGOg?e=NsP59Z.

[5] Bui, N. D. Q., Yu, Y. and Jiang, L.: InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees, *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1186–1197 (online), DOI: 10.1109/ICSE43902.2021.00109 (2021).

[6] Evosuite: Evosuite: Automatic Test Suite Generation for Java, https://www.evosuite.org/ (2021).

[7] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc., USA (1999).

[8] Inoue, K. and Roy, C. K.: *Code Clone Analysis: Research, Tools, and Practices*, Springer, Singapore (2021).

[9] ISO/IEC 25010: ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) —System and software quality models (2011).

[10] Liu, H., Shen, M., Zhu, J., Niu, N., Li, G. and Zhang, L.: Deep Learning Based Program Generation from Requirements Text: Are We There Yet?, *IEEE Transactions on Software Engineering*, (online), DOI: 10.1109/TSE.2020.3018481 (2020).

[11] Mou, L., Li, G., Zhang, L., Wang, T. and Jin, Z.: Convolutional Neural Networks over Tree Structures for Programming Language Processing, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, AAAI Press, p. 1287–1293 (2016).

[12] Nakagawa, T., Higo, Y. and Kusumoto, S.: NIL: Large-Scale Detection of Large-Variance Clones, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, p. 830–841 (online), DOI: 10.1145/3468264.3468564 (2021).

[13] Rabin, M. R. I. and Alipour, M. A.: ProgramTransformer: A tool for generating semantically equivalent transformed programs, *Software Impacts*, Vol. 14, p. 100429 (online), DOI: https://doi.org/10.1016/j.simpa.2022.100429 (2022).

[14] Randoop: Randoop: Automatic unit test generation for Java, https://randoop.github.io/randoop/ (2022).

[15] Svajlenko, J. and Roy, C. K.: BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench, *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, (online), DOI: 10.1109/icsme.2016.31 (2016).

[16] Wang, P., Svajlenko, J., Wu, Y., Xu, Y. and Roy, C. K.: CCAligner: A Token Based Large-Gap Clone Detector, *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1066–1077 (online), DOI: 10.1145/3180155.3180179 (2018).

[17] Wu, M., Wang, P., Yin, K., Cheng, H., Xu, Y. and Roy, C. K.: LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach, *IEEE Access*, Vol. 8, pp. 27986–27997 (online), DOI: 10.1109/ACCESS.2020.2971545 (2020).

[18] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K. and Liu, X.: A Novel Neural Source Code Representation Based on Abstract Syntax Tree, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794 (online), DOI: 10.1109/ICSE.2019.00086 (2019).

[19] Zhao, G. and Huang, J.: DeepSim: Deep Learning Code Functional Similarity, *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, New York, NY, USA, Association for Computing Machinery, p. 141–151 (online), DOI: 10.1145/3236024.3236068 (2018).

[20] 渡邉凌雅，肥後芳樹，楠本真二：プログラム構造が自動生成テストの網羅率に与える影響の調査，電子情報通信学会技術報告，Vol. 122, No. 432 (2023).