修士学位論文

題目

# 深層学習を用いた意味的コードクローン検出手法の評価：SemanticCloneBench を題材として

指導教員

肥後 芳樹 教授

報告者

鶴 智秋

2023 年 2 月 1 日

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 ソフトウェア工学講座

2022 年度 修士学位論文

深層学習を用いた意味的コードクローン検出手法の評価：SemanticCloneBench を題材として

鶴 智秋

## 内容梗概

コードクローン（クローン）とは，ソースコード中の一致もしくは類似するコード片を指す．また，構文上異なるにも関わらず同様の振る舞いが記述されているクローンを意味的クローンと呼ぶ．近年では，クローン検出の精度向上及び意味的クローンの検出を目的として，深層学習を用いたクローン検出手法が多数提案されている．これらのクローン検出手法における性能評価では，BigCloneBench と呼ばれる，クローン検出において有名かつ巨大なベンチマークが利用されている．しかし，BigCloneBench には，意味的に異なるにもかかわらずクローンとして定義されているソースコードのペアが存在するなど，意味的クローンの検出性能評価に用いる場合，多くの問題が存在する．

本研究では，前述の意味的クローン検出性能評価における問題を解決するために，SemanticCloneBench と呼ばれる，意味的クローンの性能評価に特化したベンチマークを用いて，深層学習に基づくクローン検出手法の検出性能を評価する．本研究では，深層学習を用いたクローン検出手法のうち，ASTNN ，CodeBERT ，InferCode の 3 手法を評価対象とする．適用実験では，F 値や AUC，実行時間の比較により，各手法における意味的クローン検出の性能を評価する．適用実験の結果より，ASTNN における F 値と AUC の値がそれぞれはそれぞれ 0.892，0.929 であり，評価対象となる検出手法の中で最も高い検出性能を示した．また，評価対象手法のうち，CodeBERT の実行時間が最も短かった．

### 主な用語

コードクローン，意味的コードクローン，深層学習，SemanticCloneBench

# 目次

# 図目次

# 表目次

## 1　はじめに

コードクローン（以後，クローン）とは，他と一致または類似する箇所があるソースコード片である．

一般的に，クローンは，構文的な類似度に基づいて，構文的クローンと意味的クローンの 2 種類に分類される．構文的クローンは，トークン列や命令文など，ソースコードの構造が類似するクローンを指す．一方，意味的クローンは，同じ振る舞いであるにもかかわらず，構文的に異なるクローンを指す，意味的クローンの検出は，ソースコードをクエリとしたソースコード検索 [4] [5] や，API の使用例に基づく再利用のためのソースコード推薦 [6] などへの応用に期待されている．

これまでに，構文的クローンに着目したクローン検出手法が多数提案されている [7] [8] [9] [10]．しかし，構文的クローンに着目した古典的なクローン検出手法では，意味的クローンの検出が非常に困難である，なぜならば，意味的クローンは，振る舞いだけが一致しているに過ぎず，トークン列や命令文などの構文的構造が異なるため，その検出難度は構文的クローンの検出難度よりも非常に高くなるためである．したがって，意味的クローン検出のためには，構文的クローン検出とは異なるアプローチが必須となる．近年では，意味的クローンを検出するために，深層学習的なアプローチからクローンを検出する手法が提案され始めている [11] [12] [13] [14] [15] [16] [1] [17] [2] [3]．ソースコードを入力とした深層学習では，ソースコード片同士における意味的な類似を学習できるため，意味的クローンの検出及び検出精度の向上が期待される．

多数の深層学習を用いたクローン検出手法が提案され続ける一方で，それらの検出手法に対する評価において課題が発生する．多くの深層学習を用いたクローン検出手法では，意味的クローン検出性能評価のために，BigCloneBench [18] [19] と呼ばれる，オープンソースソフトウェアに含まれるソースコードから構成された巨大なベンチマークが用いられる．BigCloneBench は，構文的クローンはもちろん多くの意味的クローンを正解クローンとして含むため，正解クローンを多く学習した場合の意味的クローン検出性能の評価が可能である．しかし，Krinke らの報告によると，異なる振る舞いを行うソースコードのペアが正解クローンとして定義されているなど，意味的クローンの検出性能評価において多くの問題が存在することが報告されている [20]．BigCloneBench を意味的クローンの検出性能評価に用いる場合，誤った正解クローンの影響で妥当性が脅されると考えられ，意味的クローンの検出を目的とする，深層学習を用いたクローン検出手法における評価に対する結果の信頼性が低下する [20]．

本研究では，深層学習を用いたクローン検出手法における評価の見直しを目的として，SemanticCloneBench [21] を題材とした深層学習に基づくコードクローン検出手法における評価を実施する．本研究の貢献は，以下の通りである．

- SemanticCloneBench を用いた評価実験の実施：SemanticCloneBench は，意味的クローンの検出性能評価に特化したベンチマークである．SemanticCloneBench では，プログラミング言

語における質問回答 Web サイトである Stack Overflow の同一質問に含まれる回答ソースコードを，意味的クローンとして定義する． SemanticCloneBench では，正解クローンの定義が明確であるため，SemanticCloneBench を用いた意味的クローン検出性能の評価における結果は，BigCloneBench を用いた意味的クローン検出性能の評価における結果よりも信頼できると考えられる．

- 教師あり学習を用いたクローン検出手法と教師なし学習を用いたクローン検出手法との比較：深層学習を用いたクローン検出では，教師あり学習を用いた手法が主流である [12] [13] [14] [15] [16] [1] [17] [2]．一方，教師なし学習を用いた手法も数件か提案されている [11] [3]．教師あり学習を用いた手法では，学習のために検出対象データセットに対して正解クローンが必須である．したがって，予め検出対象データセット内部に含まれるソースコードのペアに対して，人力で正解クローンをアノテーションする必要があり，正解クローン集合に依存してクローンの検出能力が制限される．教師あり学習を用いたクローン検出手法における正解クローンの必要性を軽減するために，教師なし手法を用いた手法が提案され始めている [3]．著者が知る限り，教師あり学習を用いた手法と教師なし学習を用いた手法を直接比較した研究は存在しない．教師あり学習を用いた手法と教師なし学習を用いた手法との未比較により，深層学習を用いたクローン検出手法の文脈において，教師あり学習と教師なし学習との位置付けが不明瞭となる．したがって，本研究では，教師あり学習を用いた手法と教師なし学習を用いた手法との比較を実施する．

本研究の適用実験では，教師あり学習を用いたクローン検出手法として，抽象構文木と回帰型ニューラルネットワークとを併用した ASTNN [1] を，ソースコードのトークンと自然言語のテキスト，及び事前学習モデルの一種である BERT [22] とを併用した CodeBERT w/ fine-tuning [2] を比較対象とする．また，教師なし学習を用いたクローン検出手法として，CodeBERT において事前学習モデルのみを用いた CodeBERT w/o fine-tuning と，木構造に基づいた畳み込みニューラルネットワーク [23] と抽象構文木とを併用した InferCode [3] を比較対象とする．評価指標として，誤検出の少なさを示す適合率，どのくらい検出漏れを防げるかどうかを示す再現率，クローン検出器における分類性能の高さを示す F 値と AUC を用いる．

適用実験の結果より，ASTNN における F 値と AUC の値はそれぞれ 0.892，0.929 となり，本実験の比較対象となるクローン検出手法の中で最も高い検出性能を得た．また，教師なし学習を用いた手法である CodeBERT w/o fine-tuning と InferCode における再現率は，それぞれ 86.3 %，84.6 % であり，古典的クローン検出手法のひとつである NiCad [24] [25] [26] の再現率 4.0 % [21] よりも非常に高い．さらに，深層学習を用いた手法に対して GPU を用いた場合，教師あり学習を用いた手法の中では CodeBERT w/ fine-tuning の実行時間が，教師なし学習を用いた手法の中では CodeBERT w/o

fine-tuning の実行時間が最も短かった.

## 2 準備

### 2.1 コードクローン

コードクローン（クローン）とは，他と一致または類似する箇所があるソースコード片である．また，互いにクローン関係にあるソースコード片のペアを，クローンペアと呼ぶ．

一般的に，クローンは，構文的な類似度の違いに基づき，以下の 4 種類に分類される [7] [27] [28].

- Type-1: 空白や改行，及びコメントの差異を除いて完全一致するクローン．
- Type-2: 変数名や関数名などの識別子名の差異を除いて一致するクローン．
- Type-3: 命令文単位での挿入や削除，変更が行われたクローン．
- Type-4: 同じ振る舞いであるにもかかわらず，構文的に異なるクローン．

上記クローンのうち，Type-1 から Type-3 までを構文的クローン，Type-4 を意味的クローンと呼ぶ．

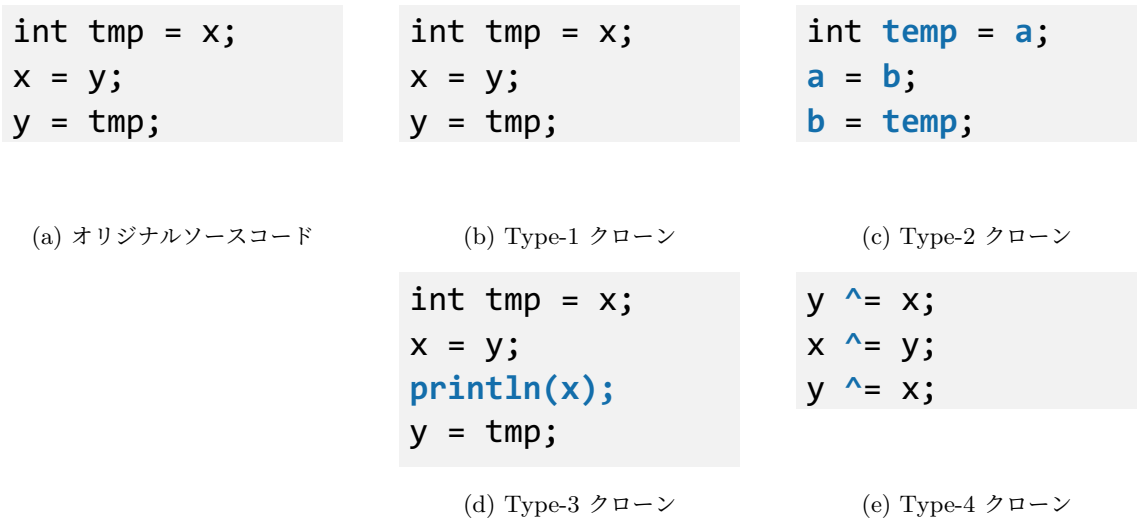
図 1 は，変数の値を交換するソースコードにおけるクローンの例である．図 1(b)，図 1(c)，図 1(d)，図 1(e) のソースコードは，図 1(a) とクローンの関係にある．図 1(b) は Type-1 クローンであり，空白や改行，及びコメントを除いて完全一致する．図 1(c) は Type-2 クローンであり，`tmp` が `temp`，`x` が `a`，`y` が `b` のように，変数名の違いを除いて完全一致する．図 1(d) は Type-3 クローンであり，変数 `x` の値に影響を及ばさない命令文である `println` が挿入されている．図 1(e) は Type-4 クローンである．図 1(b) から図 1(d) までのクローンと異なり，図 1(e) では一時変数を用いず，ビット演算における排他的論理和の組み合わせにより，変数の値交換を実装している．

これまでに，多数のクローン検出手法が提案されている [7] [8] [9] [10]．クローンの検出により，利用 API の推薦 [29] や，冗長な記述に対するリファクタリングの推薦 [30]，同一変更を要する箇所に対する変更推薦 [31]，ライセンス違反の検出 [32]，といった様々な応用が可能となる．また Type-4 クローンの検出により，ソースコードをクエリとしたソースコード検索 [4] [5] や，API の使用例に基づく再利用のためのソースコード推薦 [6] などの応用が期待される．

### 2.2 深層学習を用いたクローン検出

クローン検出性能の向上及び Type-4 クローン検出を深層学習を目的として，深層学習を用いたクローンが提案され始めている [11] [12] [13] [14] [15] [16] [1] [17] [2] [3]．深層学習を用いたクローン検出では，正解クローンの訓練方法の違いによって，教師あり学習を用いたクローン検出手法 [12] [13] [14] [15] [16] [1] [17] [2] と教師なし学習を用いたクローン検出手法 [11] [3] の 2 種類に分類される．

教師あり学習を用いたクローン検出手法では，教師となる正解クローンをクローン検出器に訓練させ




(a) オリジナルソースコード　(b) Type-1 クローン　(c) Type-2 クローン

(d) Type-3 クローン　(e) Type-4 クローン

図 1: 変数の値を交換するソースコードにおけるクローンの例

た後，学習済みクローン検出器を用いて，検出対象のデータセットに対してクローン検出を実施する（図 2）．教師あり学習を用いたクローン検出手法では，正解クローンを予め学習できるため，古典的なクローン検出手法よりも高い検出性能を発揮する．一方，教師あり学習を用いた手法では，異なるドメインから収集された訓練用データセットと評価用データセットに対して，汎化性能が低下すると報告されている [33] [34] [35].

教師なし学習を用いたクローン検出手法には，正解クローンが不明な訓練用データセットを基に訓練する手法 [11] と，事前学習したソースコード表現の類似度を基にクローン検出を実施する手法 [3] の 2 種類が存在する．正解クローンが不明な訓練用データセットを基に訓練するクローン検出手法では，教師あり学習を用いたクローン検出手法と異なり，訓練用データセットや検出用データセットに含まれる正解クローンが未知であっても検出可能であるため，正解クローンが未知の検出対象データセットに対してクローン検出が可能である（図 3(a)）．一方，訓練用データセットに含まれる非クローンを正解クローンとして誤って訓練するため，検出性能が低下する可能性が考えられる．事前学習したソースコード表現の類似度を基にしたクローン検出手法は，巨大な事前学習用データセットからソースコード表現を訓練させた後，検出対象データセットに対してクローン検出を実施する（図 3(b)）．ソースコード表現の類似度に基づく手法は，正解クローンが不明な訓練用データセットを基に訓練する手法と同様に，正解クローンが未知である検出対象データセットに対してのクローン検出が可能である．また，正解クローンが未知なデータセットにおけるソースコード片同士の類似度を訓練に用いないため，非クローンの誤訓練による検出性能の低下は発生しない．本研究では，教師なし学習を用いたクローン検出手法として，事前学習したソースコード表現の類似度を基にした手法に着目する．

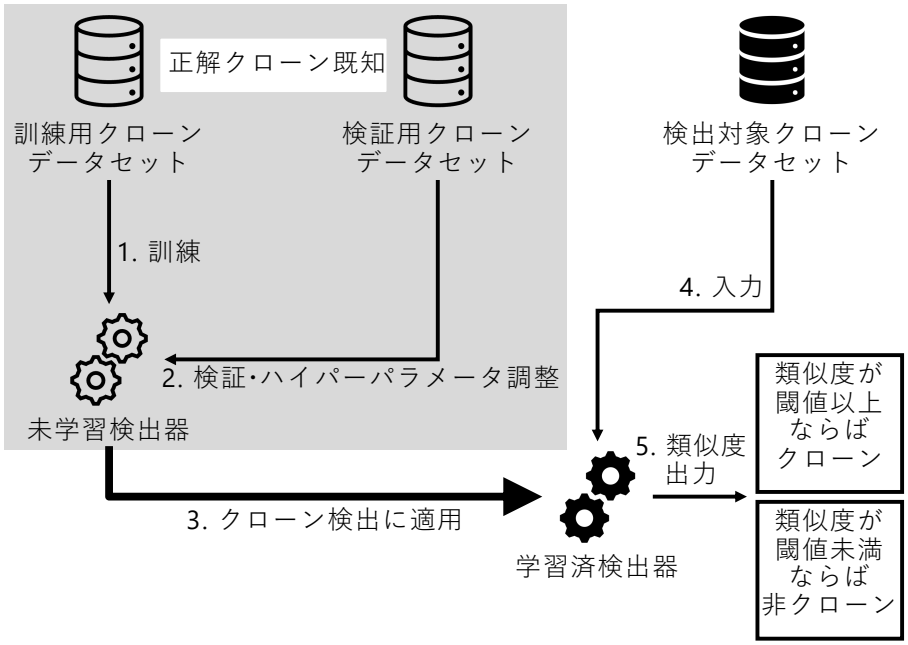



図 2: 教師あり学習を用いたクローン検出手法の概要

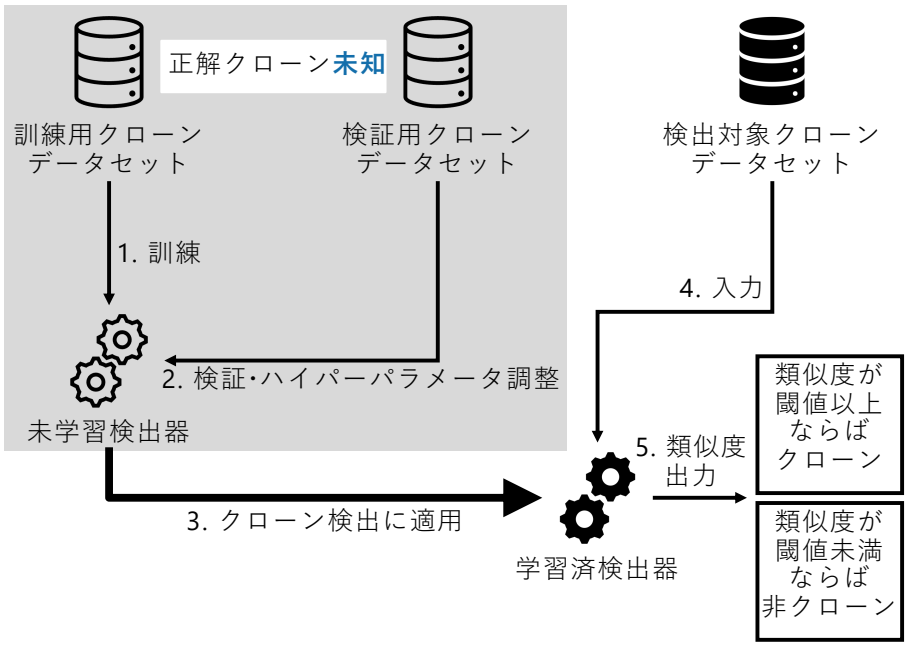
### 2.3 BigCloneBench

BigCloneBench [18] [19] は，クローン検出性能評価で用いられる大規模なベンチマークである．BigCloneBench には，プロジェクト間リポジトリの大規模データである IJaDataset 2.0 [*1] からマイニングされ，手動または自動で振り分けられた，特定の機能毎におけるクローン情報等のラベルが含まれ

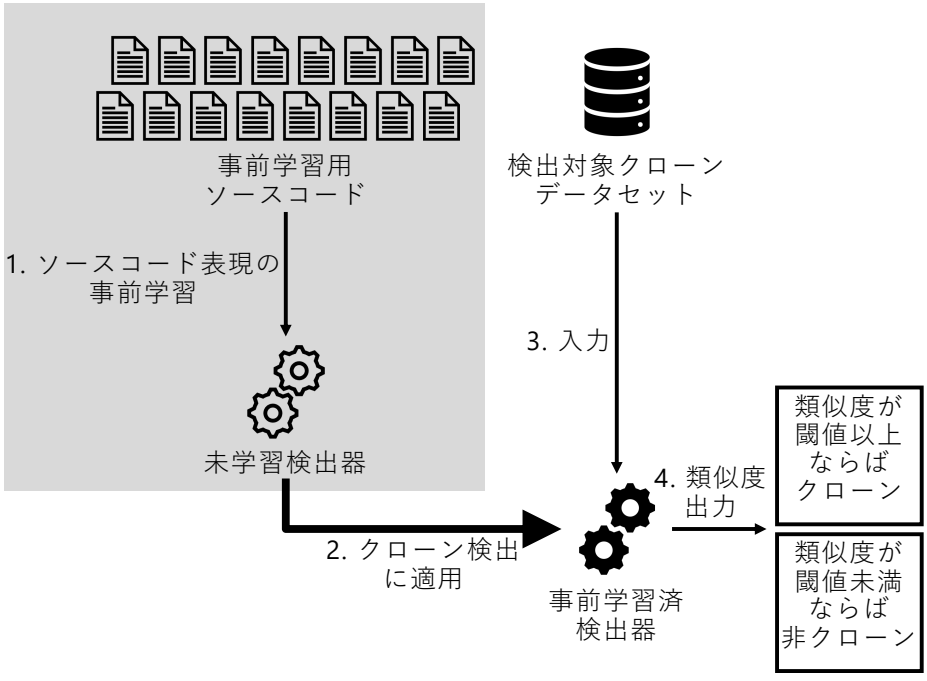
表 1: BigCloneBench におけるクローン分類の内訳

| クローン分類 | 正解クローン数 | 偽陽性クローン数 | 総計 | 割合 |
|---|---|---|---|---|
| Type-1 | 48,116 | 0 | 48,116 | 0.543 % |
| Type-2 | 4,234 | 2 | 4,236 | 0.478 % |
| Strongly Type-3 | 21,966 | 22 | 21,988 | 0.248 % |
| Moderately Type-3 | 87,480 | 406 | 87,886 | 0.992 % |
| Weakly Type-3/Type-4 | 8,422,357 | 278,602 | 8,700,959 | 98.17 % |
| 総計 | 8,584,153 | 279,032 | 8,863,185 | 100 % |

[*1] https://github.com/clonebench/BigCloneBench




(a) 正解クローンが不明な訓練用データセットを基に訓練する手法の概要




(b) 事前学習したソースコード表現の類似度を基にクローン検出を実施する手法の概要

図 3: 教師なし学習を用いたクローン検出手法の概要

```
38 public static void copyFile2(File srcFile, File destFile) throws IOException {
39     FileUtils.copyFile(srcFile, destFile);
40 }
```

(a) 機能 “Copy File” にラベリングされたメソッドの例（`CopyFileSamples.java` 内の Snippet 23677115）

```
51 private void dumpConfig() throws Exception {
52     IOUtils.copy(new FileInputStream(m_snmpConfigFile), System.out);
53 }
```

(b) 標準出力に設定ファイルをダンプするメソッドであるにもかかわらず，機能 “Copy File” としてラベリングされたメソッドの例（`1362837.java` 内の Snippet 2571845）

```
42 private static final void makeWF_BasicJavaWriterFormat_jwf (Hashtable pWriterFormats) {
43     pWriterFormats.put ("BasicJavaWriterFormat.jwf", "...");  // 第二引数の詳細は省略
44 }
```

(c) 引数で指定されたハッシュテーブルに文字列定数を格納するメソッドであるにもかかわらず，機能 “Copy File” としてラベリングされたメソッドの例（`184404.java` 内の Snippet 19962035）

図 4: BigCloneBench において，異なる振る舞いを行うソースコードのペアが正解クローンとして定義されているクローンの例

る．Type-3 及び Type-4 の境界が曖昧であるため，BigCloneBench では，命令文単位での構文的類似度に基づいて，0 以上 1 未満の範囲で，Type-3/Type-4 クローンを定義している．

- Strongly Type-3: 0.7 以上 1.0 未満
- Moderately Type-3: 0.5 以上 0.7 未満
- Weakly Type-3/Type-4: 0 以上 0.5 未満

BigCloneBench は，多くの古典的なクローン検出手法において，クローン検出性能評価のベンチマークとして幅広く使用されている [36] [37] [38] [39] [40]．また，BigCloneBench に含まれる正解クローンペアの大多数が Weakly Type-3/Type-4 であるため（表 1），深層学習を用いた手法における性能評価ベンチマークとしても使用されている [12] [13] [14] [15] [1] [17].

しかし，Krinke らの報告によると，BigCloneBench には，Type-4 クローンの検出性能評価におい

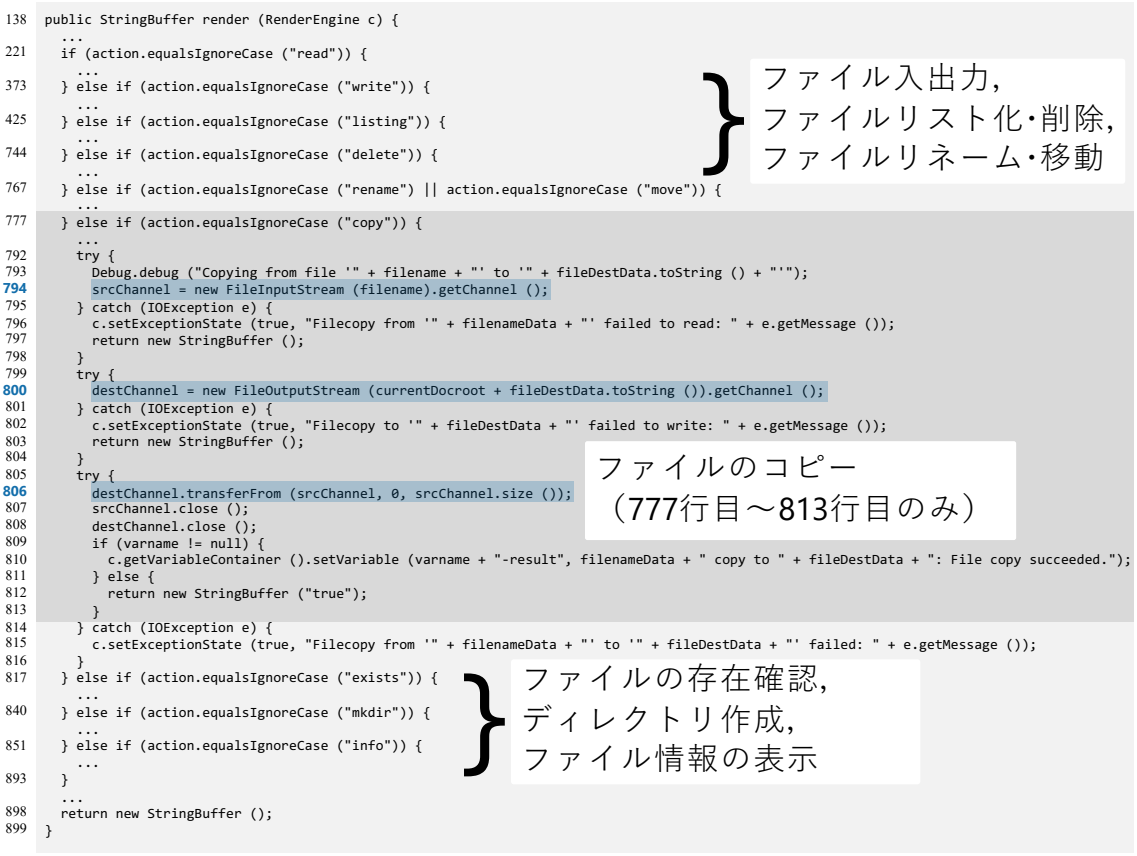



図 5: ファイルのコピー以外の機能を複数有するにもかかわらず，機能 “Copy File” のみにラベリングされたメソッドの例（`402201.java` 内の Snippet 23094500）

て下記の問題が存在する [20].

1. 異なる機能同士におけるソースコードのペアに関する情報を含有していない.
2. 異なる機能としてラベル付けされているにもかかわらず，クローンペアとみなせるソースコードのペアが存在する．例えば，BigCloneBench に含まれるソースコードのうち，ファイルをコピーする機能 “Copy File” を有するメソッドと，ディレクトリをコピーする機能 “Copy Directory” を有するメソッドはクローンであるとみなせる．なぜならば，“Copy File” は “Copy Directory” の一部であり，“Copy Directory” として分類されたクローンのうち，少なくとも 77 個のメソッドがメソッド `copyFile` を呼び出しているためである．しかし，BigCloneBench では，機能 “Copy Directory” を有するメソッドと機能 “Copy File” を有するメソッドのペアの多くを，正解クローンとしてみなしていない.
3. 同様の機能であるにもかかわらず，クローンとしてみなされていないクローンペア，もしくは誤って同様の機能として分類されたソースコードのペアが存在する.

4. 異なる振る舞いを行うソースコードのペアが正解クローンとして定義されている（図 4）.
5. 複数機能を有するソースコードの存在を考慮していない．例えば，BigCloneBench には，複数の機能を有するにもかかわらず, 機能 “Copy File” のみにラベリングされたメソッド（`402201.java` 内の Snippet 23094500）が存在する（図 5）．図 5 のソースコード行数は 762 であるにもかかわらず，ファイルのコピーに関する処理を行うソースコードは全体の約 5 % である（777 行目から 816 行目までの 40 行）．また，実際にファイルのコピーを実施する命令文の行数は全体の 0.4 % 以下（全 762 行の内，794 行目，800 行目，806 行目の 3 行）である．したがって，図 5 のような，複数の機能を有するソースコード片と，図 4(a) のような，単一の機能 “Copy File” のみを有するソースコード片とを，同一の機能を持つクローンとしてラベリングすべきではない.
6. 機能毎における分布は不均衡であり，バイアスが存在する．BigCloneBench には，合計で 43 種類の機能を持つ正解クローンが含まれる．BigCloneBench に含まれる正解クローンの 90 % が 8 種類の機能で構成されている一方で，22 種類の機能を足し合わせた正解クローンは全体の 1 % 未満であり，バイアスが存在している．特に，“Copy File” に含まれる正解クローンは，BigCloneBench に含まれる全ての正解クローンの 40 % を占める.

BigCloneBench を Type-4 クローンの検出性能評価に用いる場合，上記の問題により，誤った正解クローンの影響で妥当性が脅されるため，Type-4 クローンの検出を目標とする，深層学習を用いたクローン検出手法における評価に対する結果の信頼性が低下する.

## 3 実験

図 6 は，本研究における適用実験の手順である．まず，SemanticCloneBench に含まれるソースコードを，実験対象のクローン検出器で使われる深層学習モデルに入力できるように，トークン列や抽象構文木 (AST) 等のコード表現に変換する．次に，変換されたコード表現と，SemanticCloneBench から作成した訓練用データセットを用いて，クローン検出器で使われる深層学習モデルを訓練させる．なお，訓練時には，検証用データセットに対する検出結果を基にして，深層学習モデル内部のハイパーパラメータを調節する．最後に，評価用データセットに対して，訓練済みのクローン検出器を用いたクローン検出を実施する．クローンの検出結果から得られた検出精度及び実行時間により，評価を実施する．

### 3.1 実験環境

適用実験は，10 コアの 2.2 GHz Intel Xeon E5-2630 v4 CPU と，5.5 TB HDD，256 GB メインメモリ，NVIDIA Tesla V100S GPU 上で実施される．

### 3.2 比較対象

本研究では，教師あり学習を用いたクローン検出手法として ASTNN [1]，CodeBERT w/ fine-tuning [2], 教師なし学習を用いたクローン検出手法として CodeBERT w/o fine-tuning , InferCode [3] を比較対象として適用実験を実施する．

深層学習を用いたクローン検出では，教師あり学習を用いた手法が多数提案されている [12] [13] [16] [14] [15] [1] [17] [2]．一方，教師なし学習を用いた手法も数件か提案されている [11] [3]．教師あり学習を用いた手法では，学習のために検出対象データセットに対して正解クローンが必須である．したがって，予め検出対象データセット内部に含まれるソースコードのペアに対して，人力で正解クローンをアノテーションする必要があり，正解クローン集合に依存してクローンの検出能力が制限される．教師あり学習を用いたクローン検出手法における正解クローンの必要性を軽減するために，教師なし手法を用いた手法が提案され始めている [3]．

著者が知る限り，教師あり学習を用いた手法と教師なし学習を用いた手法を直接比較した研究は存在しない．教師あり学習を用いた手法と教師なし学習を用いた手法との未比較により，深層学習を用いたクローン検出手法の文脈において，教師あり学習と教師なし学習との位置付けが不明瞭となる．したがって，本研究では，教師あり学習を用いた手法と教師なし学習を用いた手法との比較を実施する．

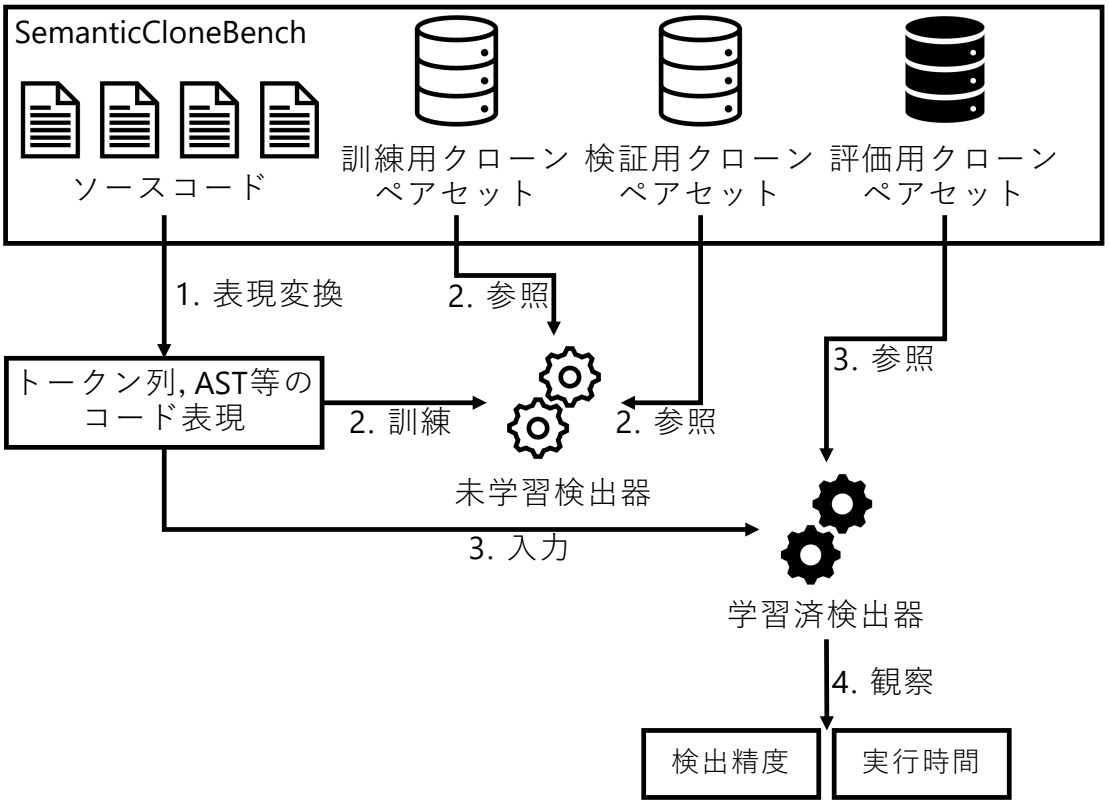



図 6: 適用実験の概要

#### 3.2.1 ASTNN

ASTNN は，抽象構文木ベースのニューラルネットワークモデルである [1]．抽象構文木と回帰型ニューラルネットワーク (RNN) との併用により，字句情報や命令文単位での構文情報を学習できる (図 7)．クローン検出では，ソースコードのペア同士における類似度として，L1 ノルムを使用する．

ASTNN を提案した Zhang らは，閾値を 0.5 に設定し，BigCloneBench [18] と OJClone [23] を題材とした適用実験を実施した．彼らは，OJClone では，適合率が 98.9 %，再現率が 92.7 %，F 値が 95.5 % であったと報告している．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 では，適合率が 99.8 %，再現率が 88.3%，F 値が 0.938 であったと報告している．Zhang らは，ASTNN の評価に BigCloneBench をベンチマークとして使用しているため，Type-4 クローン検出に対する評価結果の信頼性は低い．

適用実験では，GitHub で公開されている ASTNN リポジトリ [*2]を利用する．クローン検出のため

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

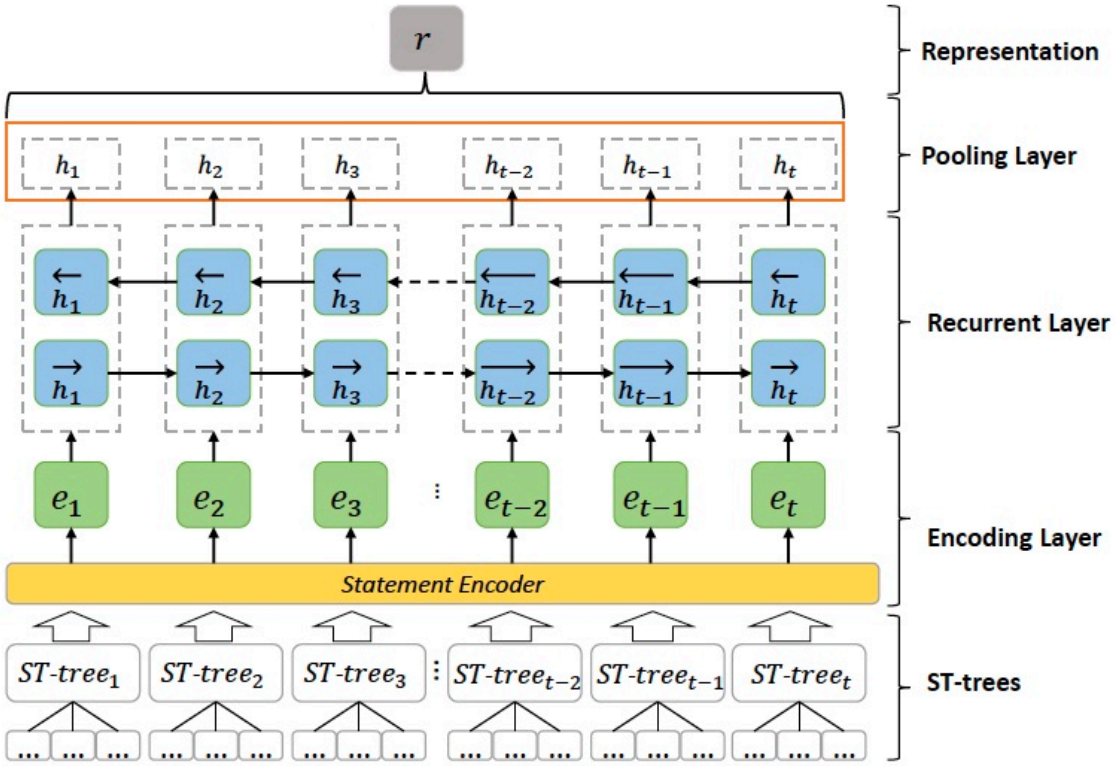



図 7: ASTNN の概要（Zhang ら [1] より引用）

に，Skip-gram アルゴリズムを利用した Word2vec [41] のエンベディングシンボルを訓練する．エンベディングシンボルのサイズを 128 に，隠れ層のサイズを 100 に設定する．また，バッチサイズを 32 に，訓練時のエポック数を 5 に設定する．最後に，訓練時のオプティマイザ AdaMax [42] における学習率を 0.002 に設定する．

### 3.2.2 CodeBERT

CodeBERT は，トークンベースの事前学習モデルである [2]．ソースコードのトークンと自然言語のテキスト，及び BERT [22] の併用により，プログラミング言語及び自然言語のマルチモーダルな情報を学習できる（図 8）．CodeBERT では，ソースコードのトークンと自然言語のテキストとを事前学習したのち，ソースコードドキュメント生成などのタスクのためにファインチューニングする．

CodeBERT を提案した Feng らは，CodeBERT を用いたクローン検出に対して評価を実施していない．代わりに，Guo らが，BigCloneBench に対して適用実験を実施した [43]．彼らは，適合率が 94.8 %，再現率が 94 %，F 値が 95 % であったと報告している．Guo らは，CodeBERT の評価に BigCloneBench をベンチマークとして使用しているため，Type-4 クローン検出に対する評価結果の信

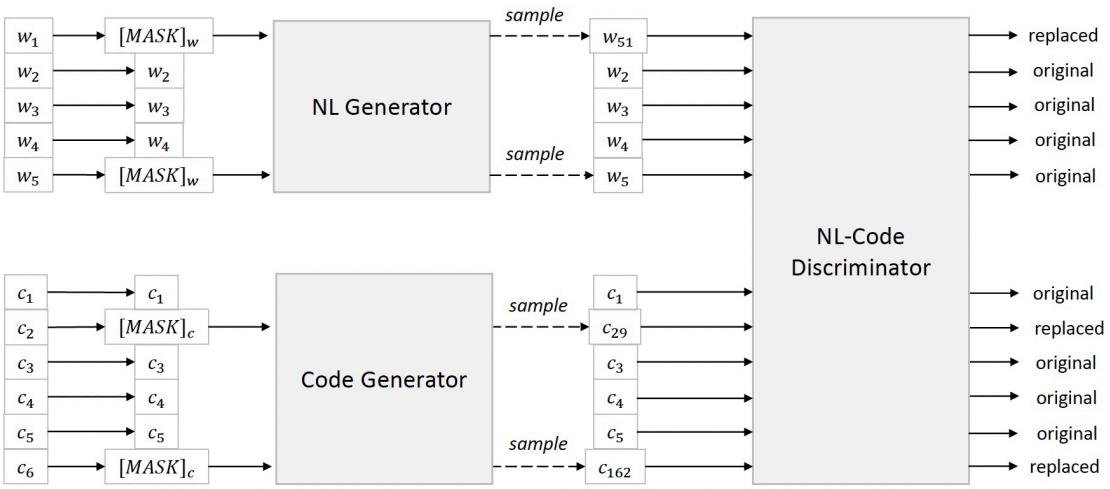



図 8: CodeBERT における，事前学習部の概要．一部トークンのマスキングや置換により，ソースコード表現を学習できる（Feng ら [2] より引用）

頼性は低い．また，クローン分類毎に検出性能を評価していないため，Type-4 クローンの検出性能が不明瞭である．さらに，クローン分類における閾値が明示されていないため，Type-4 クローンだけでなく，Type-1，Type-2，Type-3 クローンの検出性能に対しても信頼性は低い．

本研究では，クローン検出をタスクとしてファインチューニングし，CodeBERT におけるクローン検出性能の評価を実施する．また，事前学習のみにおけるクローン検出の性能評価を行うために，ファインチューニングあり CodeBERT (CodeBERT w/ fine-tuning) とファインチューニングなし CodeBERT (CodeBERT w/o fine-tuning) の 2 種類を比較対象とする．

適用実験では，Hugging Face にて公開されている CodeBERT のモデル [*3] [*4] を利用する．また，ソースコードのペア同士における類似度として，コサイン類似度を使用する．事前学習モデルにおける入力ソースコードトークンの長さを 256 に，バッチサイズを 16 に設定する．また，ファインチューニング時における学習率を 0.00002，エポック数を 1 に設定する．最後に，ファインチューニング時のオプティマイザとして Adam を利用する．

### 3.2.3 InferCode

InferCode は，抽象構文木ベースの事前学習モデルである．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 (TBCNN) [23] との併用により，事前学習モデルとして，ソースコードクラスタリングなどの教師なし学習タスクや，ソースコード分類などの教師あり学習タスクに応用できる

[*3] https://github.com/microsoft/CodeBERT
[*4] https://huggingface.co/microsoft/codebert-base

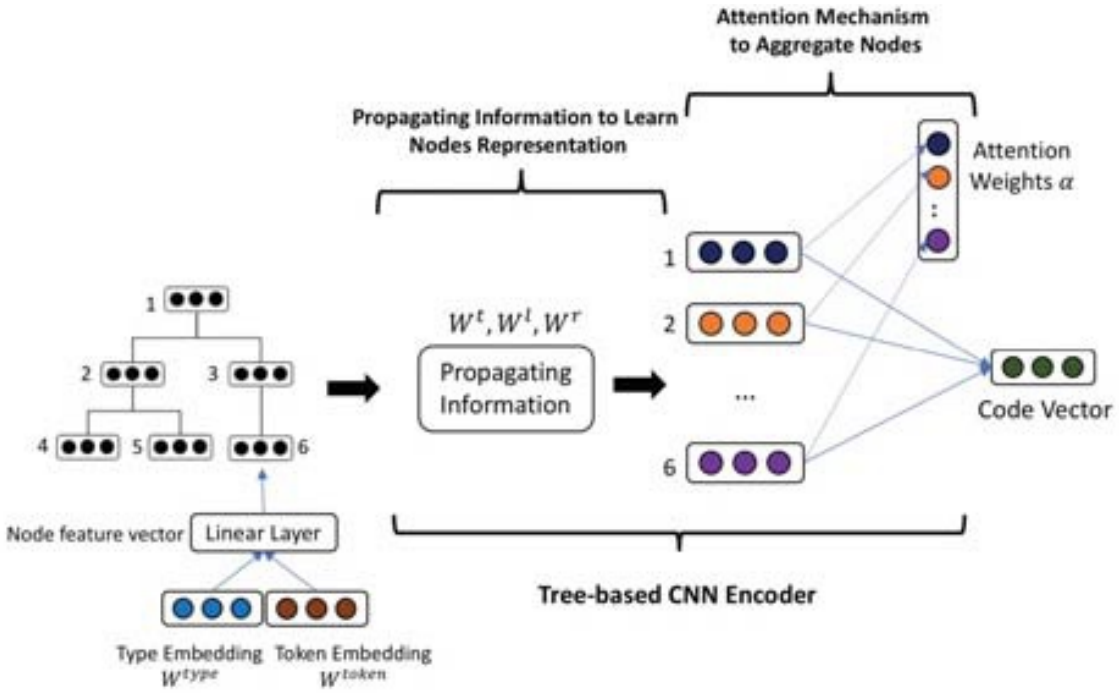



図 9: InferCode の概要（Bui ら [3] より引用）

（図 9）．Bui らは，InferCode をクローン検出を教師なし学習タスクとして実装している．本研究でも，InferCode は教師なし学習を用いたクローン検出手法として，適用実験の比較対象とする．クローン検出では，ソースコードのペア同士における類似度として，コサイン類似度を使用する．

InferCode を提案した Bui らは，閾値を 0.8 に設定し，BigCloneBench [18] と OJClone [23] を題材とした適用実験を実施した．彼らは，BigCloneBench では，適合率が 90 %，再現率が 56 %，F 値が 75 %，OJClone では，適合率が 61 %，再現率が 70 %，F 値が 64 % であったと報告している．Bui らは，ASTNN の評価に BigCloneBench をベンチマークとして使用しているため，Type-4 クローン検出に対する評価結果の信頼性は低い．また，クローン分類毎に検出性能を評価していないため，Type-4 クローンの検出性能が不明瞭である．

適用実験では，PyPI にて公開されている InferCode のモデル [*5] [*6] を利用する．また，クローン検出時におけるバッチサイズを 5 に設定する．

[*5] https://github.com/bdqnghi/infercode
[*6] https://pypi.org/project/infercode

### 3.3 SemanticCloneBench

適用実験では，Type-4 クローン検出の性能評価に用いるベンチマークとして，SemanticCloneBench [21] を用いる．SemanticCloneBench は，Type-4 クローンの検出性能評価に特化したベンチマークである．SemanticCloneBench に含まれるクローンは，プログラミング言語における質問回答 Web サイトである Stack Overflow [*7] から抽出されている．SemanticCloneBench では，同一質問に含まれる回答ソースコードを，Type-4 クローンとして定義している．SemanticCloneBench では，C，C#，Java，Python で記述されたソースコードが収集されており，言語毎にそれぞれ 1,000 個の Type-4 クローンのペアが含まれている．

SemanticCloneBench は，オンラインプログラミングコンテスト上に提出されたソースコードから Type-4 クローンを収集した Google Code Jam [16] や CodeNet [44] とは異なり，Stack Overflow から Type-4 クローンを抽出している．オンラインプログラミングコンテスト上に提出されたソースコードの数は膨大であり，かつ，同一問題に対する解答ソースコードの集合をクローンとして定義できるため，Google Code Jam や CodeNet をベンチマークとして使用可能である．しかし，オンラインプログラミングコンテスト上に提出されたソースコードには，実際のプロジェクトには見られないプログラミングコンテスト特有のテクニックが存在する [45]．一方，Stack Overflow は，プログラミング言語における質問回答 Web サイトである．Stack Overflow に含まれる回答ソースコードは，実際のプロジェクトに含まれていない可能性が考えられる．しかし，SemanticCloneBench に含まれるクローンは，実際の開発者による Stack Overflow に投稿された質問に対する解決例の集合であるため，実際のプロジェクト開発で発生した問題に対する解決ソースコードが Stack Overflow に存在する可能性は，オンラインプログラミングコンテスト上におけるソースコードの集合に存在する可能性よりも高いと考えられる．本研究では，深層学習を用いたクローン検出手法を応用する場合を考慮して，信頼性の高い，かつ実際のプロジェクトで用いられるソースコードに近い評価用ベンチマークとして SemanticCloneBench を採用する．

適用実験では，SemanticCloneBench に含まれる Java 言語で記述された 1,000 個のクローンのうち，Python の javalang [*8] パッケージで構文解析可能な 997 個のクローンを対象とする．また，正解クローンと非正解クローンの比が 1：1 となるように，非正解クローンを作成する．具体的には，SemanticCloneBench に含まれるメソッドをランダムに 2 個抽出し，抽出したメソッドのペアが SemanticCloneBench でクローンとして定義されていないならば，このメソッドのペアを非正解クローンとして定義する．さらに，上記の作業により作成された非正解クローンを含めた

[*7] https://stackoverflow.com/

[*8] https://github.com/c2nes/javalang

SemanticCloneBench を 6 : 1 : 3 に分割し，それぞれ訓練用データセット・検証用データセット・評価用データセットとして，学習及び評価を実施する．なお，本研究の比較対象となる教師なし学習を用いた手法は事前学習モデルである．事前学習モデルによる教師なし学習を用いたクローン検出では，正解クローンの学習なしで，事前学習モデルの出力値を基にした類似度計算に基づき，クローンの検出を実施する．したがって，事前学習モデルによる教師なし学習を用いた手法では，正解クローンの学習が不要であるため，適用実験における教師なし学習を用いた手法の評価の際，訓練用データセット及び検出用データセットを用いず，評価用データセットのみを用いる．

### 3.4 評価指標

本研究では，真陽性クローン，真陰性クローン，偽陽性クローン，偽陰性クローンを下記の通り定義する．

- 真陽性 (True positive): クローン検出器において検出されるクローン．
- 真陰性 (True negative): クローン検出器において検出されない非クローン．
- 偽陽性 (False positive): クローン検出器において検出される非クローン．
- 偽陰性 (False negative): クローン検出器において検出されないクローン．

適用実験では，以下の評価指標を用いて，各手法における Type-4 クローンの検出性能を評価する．

- 閾値を固定した時における評価指標.
  - 適合率 (Precision): 真陽性クローン数を，真陽性クローン数と偽陽性クローン数の和で割った割合．検出誤りの度合いを測るために用いる．

$$Precision = \frac{True\_positive}{True\_positive + False\_positive}$$

  - 再現率 (Recall): 真陽性クローン数を，真陽性クローン数と偽陰性クローン数の和で割った割合．どのくらい検出漏れを防げるかどうかを測るために用いる．

$$Recall = \frac{True\_positive}{True\_positive + False\_negative}$$

  - F 値 (F1-Score): 適合率と再現率の調和平均．F 値が高ければ高いほど，クローン検出器の分類性能が高いと言える．

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

- 閾値を変化させた時における評価指標.
  - ROC 曲線：縦軸が真陽性クローン率，横軸が偽陽性クローン率の座標上で，閾値の変化により描画されて出来る曲線.

  - AUC(Area Under the Curve) [46]: ROC 曲線下の面積．AUC の値が 1 に近いほど，適切な閾値を設定した場合における F 値が高い．
- 実行時間．

### 3.5 Research Questions

本研究では，以下に示す 2 個の Research Question（RQ）を立てる．

- RQ1: 閾値に基づく評価指標より，どのクローン検出器の分解性能が高いか．
  RQ1 の調査により，Type-4 クローン検出において，クローン検出手法毎の適切な閾値を明らかにする．なお，SemanticCloneBench 以外のデータセットにおける適切な閾値は不明であるため，SemanticCloneBench に対して F 値が最大となる閾値付近を，本研究における適切な閾値の範囲と定義する．ここで適切な閾値の範囲とは，F 値が (F 値の最大値) – (F 値の最大値の 0.95 倍) となる時の閾値とする．
  また，AUC の比較により，精度の観点から見て，どのクローン検出手法が一番 Type-4 クローンを検出できるかを明確にする．
- RQ2: どのクローン検出器が，実行時間において優位か？
  一般に，深層学習では並列処理を用いて実装される．したがって，計算機資源等の制限により，正解クローンの学習及びクローン検出に多大な実行時間を要する場合が考えられる．RQ2 の調査により，GPU の有無も考慮した実行時間の観点から見て，どのクローン検出手法が一番優位かを明らかにする．

# 4 実験結果

## 4.1 RQ1: 閾値に基づく分解性能の評価

### 4.1.1 適合率，再現率，F 値による評価

表 2 は，SemanticCloneBench 上で，各クローン検出器の F 値が最大となる閾値における，適合率，再現率，F 値である．

ASTNN は，閾値が 0.380 の時に F 値が最大（0.892）となり，適合率と再現率はそれぞれ 83.9 %，95.3 % を得る．他のクローン検出器と比べると，再現率の値及び F 値が約 5 % - 10 % 高く，本実験において最も高い検出性能を示す．ASTNN では，検出対象データセット毎に，そのデータセットに含まれるトークンや抽象構文木のノードに対するエンベディングを構成するため，再現率及び F 値の値が高いと考えられる．一方，ASTNN の適合率は，CodeBERT w/ fine-tuning の適合率 84.1 % よりも 0.2 % 低い．ASTNN で用いられる深層学習モデル RNN は，CodeBERT で用いられているモデル BERT と比べて，モデル内部で記憶可能な系列データ長が短い [47]．そのため，ASTNN では，長い系列の抽象構文木において学習ができなかった結果，CodeBERT w/ fine-tuning よりも適合率が低くなると考えられる．

CodeBERT w/ fine-tuning は，閾値が 0.961 の時に F 値が最大（0.870）となり，適合率と再現率はそれぞれ 84.1 %，90.0 % を得る．他のクローン検出器と比べると，適合率の値が約 15 % 高く，CodeBERT w/ fine-tuning は，本実験において最も誤検出を起こしにくいクローン検出器であると言える．CodeBERT w/ fine-tuning は，CodeSearchNet [48] で事前学習された学習済みモデルである．正解クローンのファインチューニングによるクローン検出能力の強化された結果として，CodeBERT w/ fine-tuning の適合率が他のクローン検出手法よりも高くなると考えられる．

教師なし学習を用いたクローン検出器である CodeBERT w/o fine-tuning は，閾値が 0.984 の時に F 値が最大（0.779）となり，適合率と再現率はそれぞれ 70.9 %，86.3 % を得る．また，InferCode は，閾値が 0.881 の時に F 値が最大（0.756）となり，適合率と再現率はそれぞれ 68.2 %，84.6 % を

表 2: SemanticCloneBench に対する各クローン検出器の適合率，再現率，F 値

| | 閾値 | 適合率 | 再現率 | F 値 |
|---|---|---|---|---|
| ASTNN | 0.380 | 83.9 % | 95.3 % | 0.892 |
| CodeBERT w/ fine-tuning | 0.961 | 84.1 % | 90.0 % | 0.870 |
| CodeBERT w/o fine-tuning | 0.984 | 70.9 % | 86.3 % | 0.779 |
| InferCode | 0.881 | 68.2 % | 84.6 % | 0.756 |

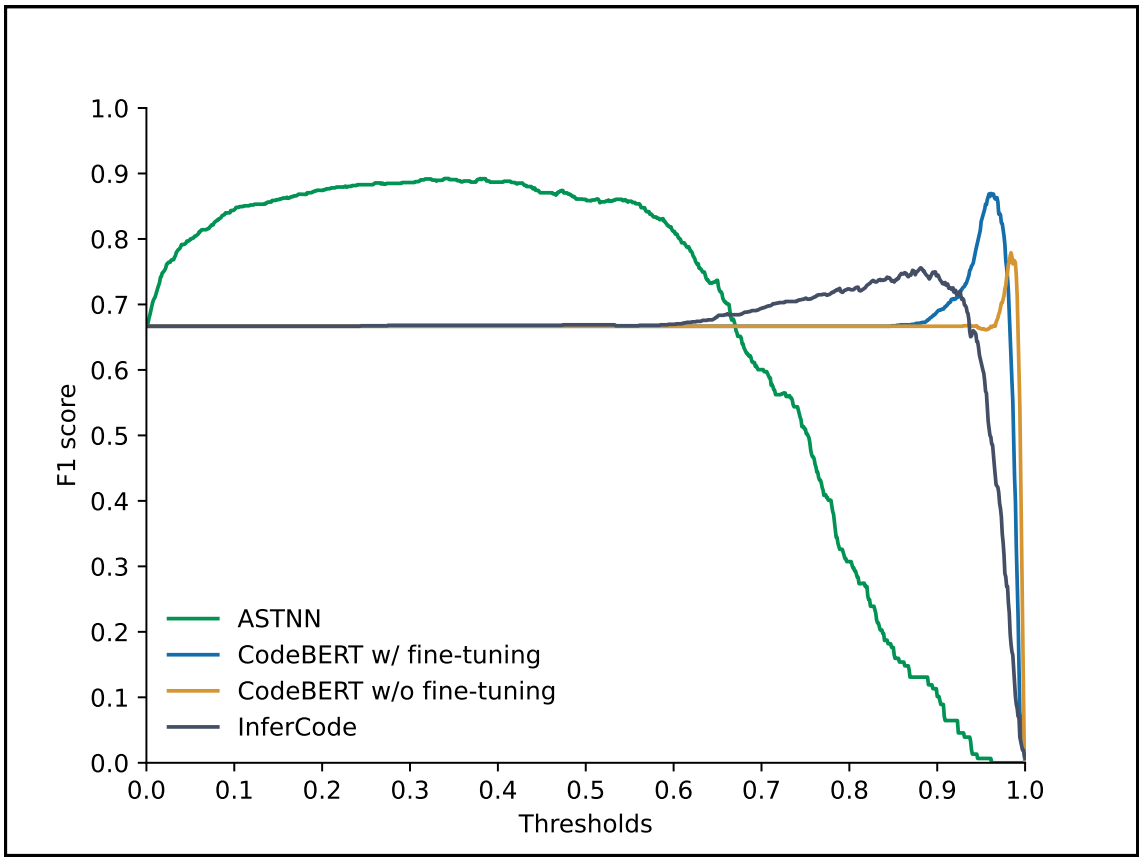



図 10: SemanticCloneBench に対して閾値を変化させた場合の F 値曲線

得る．両検出器の適合率及び F 値は，教師あり学習を用いた検出器と比べると，約 10 % - 15 % 低い．SemanticCloneBench に含まれるクローンは全て Type-4 クローン であるため，構文的クローンである Type-1，Type-2，Tyep-3 よりも検出難度が高い．また，教師なし学習を用いた手法では，正解クローンを直接学習しないため，コード表現のみの学習結果を基にクローンを検出する必要がある．Type-4 クローンにおける検出難度の高さ，及びコード表現のみの学習でのクローン検出により，教師なし学習を用いたクローン検出器における適合率及び F 値は，教師あり学習を用いた検出器と比べて低い値になると考えられる．一方，CodeBERT w/o fine-tuning と InferCode における再現率は，それぞれ 86.3 %，84.6 % であり，古典的手法のひとつである NiCad [24] [25] の再現率（4.0 %） [21] よりも非常に高い．また，教師あり学習を用いた手法と比べて，再現率の減少が高々約 10 % に抑えられている．

図 10 は，SemanticCloneBench に対して閾値を変化させた場合の F 値曲線である．

ASTNN は，閾値が [0.102, 0.567] の間，F 値が 0.847 を超える．したがって，ASTNN を使用した一般的な Type-4 クローン検出では，閾値を [0.102, 0.567] の範囲で選択すると，検出性能が高くなるような検出結果が得られると予想される．

CodeBERT w/ fine-tuning は，閾値が [0.951, 0.972] の間，F 値が 0.827 を超える．したがって，CodeBERT w/ fine-tuning を使用した一般的な Type-4 クローン検出では，閾値を [0.951, 0.972] の範囲で選択すると，検出性能が高くなるような検出結果が得られると予想される．

CodeBERT w/o fine-tuning は，閾値が [0.979, 0.990] の間，F 値が 0.740 を超える．したがって，CodeBERT w/o fine-tuning を使用した一般的な Type-4 クローン検出では，閾値を [0.979, 0.990] の範囲で選択すると，検出性能が高くなるような検出結果が得られると予想される．

InferCode は，閾値が [0.780, 0.919] の間，F 値が 0.718 を超える．したがって，InferCode を使用した一般的な Type-4 クローン検出では，閾値を [0.780, 0.919] の範囲で選択すると，検出性能が高くなるような検出結果が得られると予想される．

表 3 は，適用実験の比較対象である各クローン検出手法における適切な閾値の範囲の一覧である．本実験より，SemanticCloneBench に対する，CodeBERT や InferCode などの事前学習モデルを用いた手法における適切な閾値の範囲が，RNN を用いた手法である ASTNN における適切な閾値の範囲よりも高い，かつ狭い範囲に分布していることが判明した．事前学習モデルを用いたクローン検出手法では，クローン検出の前にコード表現を事前学習するため，RNN などの正解クローンを直接学習する手法よりも，類似度の値が高く出力される．したがって，事前学習モデルに基づくクローン検出手法を Type-4 クローン検出に用いる場合，SemanticCloneBench に対するクローン検出の時と同様に，閾値を高めに設定する必要があると考えられる．また，CodeBERT を含む事前学習モデルに基づくクローン検出手法は，F 値における最大値近傍の値が出せる閾値の範囲が狭い．したがって，事前学習モデルを用いたクローン検出器において，閾値を適切に設定できない場合，適切でない閾値を設定した時の F 値は最大値と比べて非常に小さくなるため，事前学習モデルを用いたクローン検出器が有するはずの高い検出性能を発揮できないと予想される．一方，ASTNN は F 値における最大値近傍が出せる閾値の範囲が広い．したがって，ASTNN において，仮に F 値が最大となるような閾値を設定できなかったとしても，ASTNN の F 値が最大となるような状態に近い検出性能を発揮できる可能性が高い．

表 3: ASTNN，CodeBERT w/ fine-tuning，CodeBERT w/o fine-tuning，InferCode における適切な閾値の範囲

| | SemanticCloneBench における F 値の範囲 | 閾値の範囲 |
|---|---|---|
| ASTNN | [0.847, 0.892] | [0.102, 0.567] |
| CodeBERT w/ fine-tuning | [0.827, 0.870] | [0.951, 0.972] |
| CodeBERT w/o fine-tuning | [0.740, 0.779] | [0.979, 0.990] |
| InferCode | [0.718, 0.756] | [0.780, 0.919] |

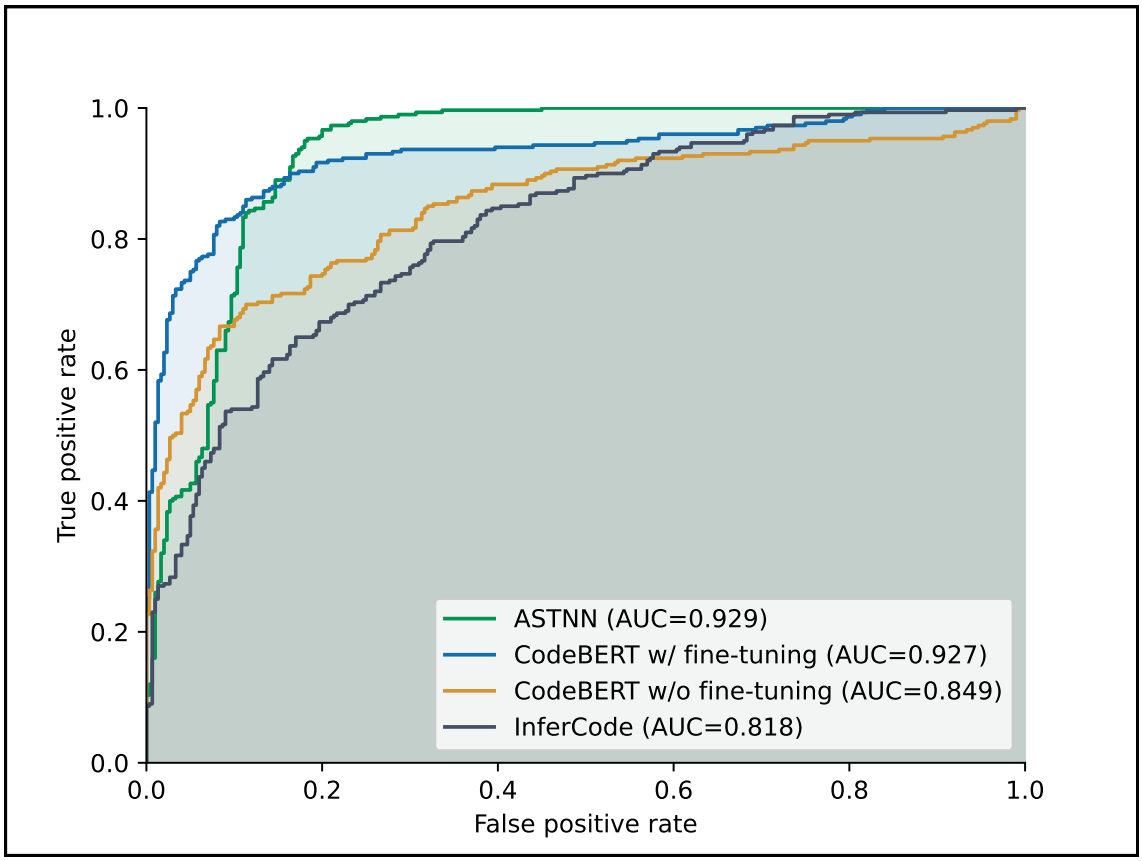



図 11: SemanticCloneBench に対する各クローン検出器の ROC 曲線と AUC

#### 4.1.2 ROC 曲線，AUC による評価

SemanticCloneBench 上における，各クローン検出器の ROC 曲線と AUC を図 11 に示す．本実験では，ASTNN の AUC（0.929）が，他のクローン検出器と比較して最も高く，ASTNN の F 値が他検出器よりも高い結果を示す表 2 と一致する．また，教師なし学習を用いたクローン検出器である CodeBERT w/o fine-tuning （0.849）と InferCode （0.818）も，教師あり学習を用いたクローン検出器に劣るものの，0.8 を超えている．

### 4.2 RQ2: 実行時間の比較

表 4 は，SemanticCloneBench 上における各クローン検出器の実行時間である．準備時間には，深層学習モデルの初期化や入力ソースコードの表現変換など，訓練やクローン検出の評価以外の処理に要する時間が含まれる．

GPU を用いた場合，総実行時間が一番短かったクローン検出器は CodeBERT w/o fine-tuning

（24.28 秒）である．また，教師あり学習を用いた検出器に限った場合，CodeBERT w/ fine-tuning の総実行時間（58.71 秒）が一番短い．CodeBERT はトークンベースの事前学習モデルである．一般に，トークンベースのクローン検出器は抽象構文木ベースの検出器よりもコストが小さいため，抽象構文木を用いたクローン検出器である ASTNN や InferCode よりも，訓練や評価に要する時間が短いと考えられる．また，CodeBERT では，PyTorch [*9]の `DataParallel` [*10] と呼ばれる，マルチ GPU による並列処理を用いている．したがって，本実験における CodeBERT では，マルチ GPU 処理による高速化により，訓練や評価に要する時間の大幅な短縮化が行われたと考えられる．

GPU を用いなかった場合，総実行時間が一番短かったクローン検出器は InferCode（1 分 31.84 秒）である．また，教師あり学習を用いた検出器に限った場合，ASTNN の総実行時間（3 分 28.89 秒）が一番短い．CodeBERT では，事前学習モデルの一種である BERT を用いて実装されている．一般に，深層学習モデルでは，並列処理により，モデル内部におけるパラメタ計算に必要な時間の軽減が行われる．BERT は一般の深層学習モデルよりも多くのパラメタを有するため，本実験においても，訓練や評価に要する時間が大幅に増加したと考えられる．

表 4: SemanticCloneBench に対する各クローン検出器の実行時間

| | GPU | 準備時間 | 訓練時間 | 検出時間 | 総実行時間 |
|---|---|---|---|---|---|
| ASTNN | あり | 10.07 秒 | 1 分 25.75 秒 | 5.10 秒 | 1 分 40.92 秒 |
| | なし | 7.29 秒 | 3 分 16.90 秒 | 4.70 秒 | 3 分 28.89 秒 |
| CodeBERT w/ fine-tuning | あり | 17.52 秒 | 35.41 秒 | 5.78 秒 | 58.71 秒 |
| | なし | 15.15 秒 | 12 分 32.03 秒 | 1 分 49.93 秒 | 14 分 37.11 秒 |
| CodeBERT w/o fine-tuning | あり | 17.18 秒 | N/A | 7.10 秒 | 24.28 秒 |
| | なし | 14.86 秒 | N/A | 2 分 11.65 秒 | 2 分 26.51 秒 |
| InferCode | あり | 2.75 秒 | N/A | 1 分 22.40 秒 | 1 分 25.15 秒 |
| | なし | 8.49 秒 | N/A | 1 分 23.35 秒 | 1 分 31.84 秒 |

[*9] `https://pytorch.org/`

[*10] `https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html`

# 5 妥当性への脅威

## 5.1 内的妥当性への脅威

CodeBERT では，PyTorch の `DataParallel` と呼ばれるマルチ GPU による並列処理により，訓練時や評価時における高速化が行われる．一方，ASTNN や InferCode では，マルチ GPU による並列処理による高速化が実装されていない．したがって，本実験の実行時間は，クローン検出手法よりも，マルチ GPU による並列処理の有無による影響が大きい可能性が考えられる．

## 5.2 外的妥当性への脅威

適用実験で使用した SemanticCloneBench に含まれる正解クローンペア数は 997 個であり，BigCloneBench に含まれる Weakly Type-3/Type-4 の正解クローンペア数（約 8,400,000 個）と比べて遥かに小さい．そのため，RQ1 の結果に影響を及ぼすと考えられる．今後，実際のプロジェクトで頻繁に出現すると期待される意味的クローンを抽出して，巨大な，かつ，多くの研究者によって審議された意味的クローンの検出性能を評価できるベンチマークの作成が必要である．

また，本研究では，SemanticCloneBench に限定して，Type-4 クローン検出における適切な閾値を予測しており，これは，RQ1 に対する外的妥当性への脅威である．今後，SemanticCloneBench 以外の Type-4 クローン検出評価ベンチマークに対する評価も実施し，一般的な Type-4 クローンにおける適切な閾値の有効性を検証したい．

## 6 おわりに

本研究では，教師あり学習を用いたクローン検出手法である ASTNN と CodeBERT w/ fine-tuning，教師なし学習を用いたクローン検出手法である CodeBERT w/o fine-tuning と InferCode に対して，クローン検出性能の評価を実施した．SemanticCloneBench を題材とした適用実験により，教師あり学習を用いた手法と教師なし学習を用いた手法の両手法において，Type-4 クローンを検出できた．教師あり学習を用いた手法と教師なし学習を用いた手法の F 値は，それぞれ 0.85，0.75 を超えた．

本研究における今後の課題として，以下が考えられる．

- 実際の開発において発生するクローンで構成された，巨大な Type-4 クローンの検出性能評価ベンチマークの作成：適用実験で使用した SemanticCloneBench に含まれる正解クローンペア数は 997 個であり，BigCloneBench に含まれる Weakly Type-3/Type-4 の正解クローンペア数（約 8,400,000 個）と比べて遥かに小さい．Type-4 クローンの巨大な検出性能評価ベンチマークとして，Google Code Jam データセット [16] が知られている．Google Code Jam とは，Google が年一回開催するオンラインプログラミングコンテストである．Google Code Jam データセットには 12 個の異なるコンテスト問題から 1,669 個の Java ソースファイルが含まれる．しかし，プログラミングコンテストで使用されるソースコードは，プログラミング特有のテクニックが含まれている [45] ため，実際の開発で使われているコーディングとかけ離れている可能性が高い．ソースコード検索 [4] [5] などの応用は，プログラミングコンテストだけでなく実際の開発にも利用されると考えられる．したがって，実際の開発において発生するクローンで構成された，巨大な Type-4 クローンの検出性能評価ベンチマークの作成が必要となる．
- ASTNN，CodeBERT，InferCode 以外における，深層学習を用いたクローン検出手法に対する Type-4 クローン検出性能の評価：本研究では，ASTNN，CodeBERT，InferCode に対して，Type-4 クローン検出の評価を実施した．深層学習を用いたクローン検出手法には，上記の 3 手法に加えて，RtvNN [11] や FA-AST+GMN [17] など，教師あり学習を用いた手法や教師なし学習を用いた手法が存在する．これらのクローン検出手法では，BigCloneBench を主軸とした Type-4 クローン検出性能の評価が行われている．BigCloneBench による Type-4 クローン検出性能の評価により，深層学習を用いたクローン検出手法における評価に対する結果の信頼性が低下する．したがって，RtvNN や FA-AST+GMN に対しても，SemanticCloneBench を含む，信頼性への脅威が小さいベンチマークを用いた Type-4 クローン検出性能の評価が必須である．

## 謝辞

肥後 芳樹 教授は，私の学部時代から大変お世話になっており，研究活動のいろはや中間報告会等における御助言等，数えきれないほどのご指導を賜りました．研究に関するご指摘は鋭い，かつ的確であったため，修士論文の質を向上させる手助けとなりました．

井上 克郎 名誉教授は，私が研究するにあたり，数多くの研究に関する御助言や激励のお言葉を賜りました．また，研究だけでなく，私の進路等に関する相談に対しても，真摯に傾聴くださり，ご助言授かりました．

松下 誠 准教授は，研究や修士論文を進めるにあたり，数多のキーアイデアや研究の進め方，修士論文執筆に関するご助言やご支援を賜りました．そして，突然の研究テーマ変更等でご迷惑をおかけしたにもかかわらず，熱心に，かつ長時間も相談に乗っていただきました．

神田 哲也 助教は，研究や修士論文を進めるにあたり，修士論文の質や妥当性を高めるためのご助言を賜りました．また，ソフトウェア工学をはじめとした近年の情報技術に関する知識やスキルを提供してくださいました．

立命館大学情報理工学部 吉田 則裕 教授は，鋭い観点で私の研究に対するご指摘を賜りました．また，私が研究に関して右往左往している時は，適切なご助言で私を導いてくださりました．

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 博士後期課程修了者 藤原 裕士 氏は，在学時に，私に対して数多くのご助言や激励のお言葉を賜りました．特に，私の研究におけるキーアイデアやウィークポイントに対して共に考えてくださり，私の研究において，大きなお力添えとなりました．

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ソフトウェア工学講座 事務職員 軽部 瑞穂 氏は，私たち学生がより良い研究を行うため，研究室の環境整備や事務的業務などといったご支援を賜りました．また，私が悩んでいる時は，親身に相談に乗ってくださり，精神面で多くの，かつ多様なご助言を賜りました．

また，本研究を進めるにあたってお世話になった，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ソフトウェア工学講座の皆様に心から感謝申し上げます．

最後に，本研究に至るまでに講義やコンピュータサイエンスセミナー等でお世話になりました，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の諸先生方に，この場をお借りして感謝の言葉を述べさせていただきます．

## 参考文献


[1] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proc. International Conference on Software Engineering*, pp. 783–794, 2019.

[2] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Proc. Conference on Empirical Methods in Natural Language Processing*, pp. 1536–1547, 2020.

[3] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. InferCode: Self-supervised learning of code representations by predicting subtrees. In *Proc. International Conference on Software Engineering*, pp. 1186–1197, 2021.

[4] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY: A code-to-code search engine. In *Proc. International Conference on Software Engineering*, pp. 946–957, 2018.

[5] Yuji Fujiwara, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Code-to-code search based on deep neural network and code mutation. In *Proc. International Workshop on Software Clones*, pp. 1–7, 2019.

[6] Shamsa Abid, Shafay Shamail, Hamid Abdul Basit, and Sarah Nadi. FACER: An API usage-based code-example recommender for opportunistic reuse. *Empirical Software Engineering*, Vol. 26, No. 6, pp. 1–58, 2021.

[7] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[8] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, Vol. 137, No. 10, pp. 1–21, 2016.

[9] Hou Min and Zhang Li Ping. Survey on software clone detection research. In *Proc. International Conference on Management Engineering, Software Engineering and Service Sciences*, pp. 9–16, 2019.

[10] Andrew Walker, Tomas Cerny, and Eungee Song. Open-source tools and benchmarks for code-clone detection: past, present, and future trends. *ACM SIGAPP Applied Computing Review*, Vol. 19, No. 4, pp. 28–39, 2020.

[11] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proc. International Conference on Automated Software Engineering*, pp. 87–98, 2016.

[12] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. CClearner: A deep learning-based clone detection approach. In *Proc. International Conference on Software Maintenance and Evolution*, pp. 249–260, 2017.

[13] Liuqing Li, He Feng, Na Meng, and Barbara Ryder. CCLearner: Clone detection via deep learning. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 75–89. 2021, Springer.

[14] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 354–365, 2018.

[15] Vaibhav Saini, Farima Farmahinifarahani, Hitesh Sajnani, and Cristina Lopes. Oreo: Scaling clone detection beyond near-miss clones. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 63–74. 2021, Springer.

[16] Gang Zhao and Jeff Huang. DeepSim: Deep learning code functional similarity. In *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 141–151, 2018.

[17] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proc. International Conference on Software Analysis, Evolution and Reengineering*, pp. 261–271, 2020.

[18] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proc. International Conference on Software Maintenance and Evolution*, pp. 476–480, 2014.

[19] Jeffrey Svajlenko and Chanchal K. Roy. BigCloneBench. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 93–105. Springer, 2021.

[20] Jens Krinke and Chaiyong Ragkhitwetsagul. BigCloneBench considered harmful for machine learning. In *Proc. International Workshop on Software Clones*, pp. 1–7, 2022.

[21] Farouq Al-Omari, Chanchal K. Roy, and Tonghao Chen. SemanticCloneBench: A semantic code clone benchmark using crowd-source knowledge. In *Proc. International Workshop on Software Clones*, pp. 57–63, 2020.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of

deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proc. AAAI Conference on Artificial Intelligence*, pp. 1287–1293, 2016.

[24] Chanchal K. Roy and James R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. International Conference on Program Comprehension*, pp. 172–181, 2008.

[25] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proc. International Conference on Program Comprehension*, pp. 219–220, 2011.

[26] Manishankar Mondal, Chanchal K. Roy, and James R. Cordy. NiCad: A modern clone detector. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 45–50. Springer, 2021.

[27] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[28] Katsuro Inoue. Introduction to code clone analysis. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 3–27. Springer, 2021.

[29] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proc. International Conference on Software Engineering*, pp. 664–675, 2014.

[30] Norihiro Yoshida, Seiya Numata, Eunjong Choiz, and Katsuro Inoue. Proactive clone recommendation system for extract method refactoring. In *Proc. International Workshop on Refactoring*, pp. 67–70, 2019.

[31] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An exploratory study on change suggestions for methods using clone detection. In *Proc. International Conference on Computer Science and Software Engineering*, pp. 85–95, 2016.

[32] Akito Monden, Satoshi Okahara, Yuki Manabe, and Kenichi Matsumoto. Guilty or not guilty: Using clone metrics to determine open source licensing violations. *IEEE software*, Vol. 28, No. 2, pp. 42–47, 2010.

[33] 藤原裕士, 森彰, 井上克郎. 回帰モデルを用いたコードクローン検出手法の提案と汎化性能の評価. 電子情報通信学会論文誌 D, Vol. J104-D, No. 9, pp. 678–689, 2021.

[34] 福家範浩, 藤原裕士, 吉田則裕, 崔恩瀞, 井上克郎. 深層学習を用いたコードクローン検出器の汎化性能に関する調査. 情報処理学会研究報告, Vol. 2021-SE-207, No. 12, pp. 1–7, 2021.

[35] Saad Arshad, Shamsa Abid, and Shafay Shamail. CodeBERT for code clone detection: A replication study. In *Proc. International Workshop on Software Clones*, pp. 39–45, 2022.

[36] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proc. International Conference on Software Engineering*, pp. 1157–1168, 2016.

[37] Hitesh Sajnani, Vaibhav Saini, Chanchal K. Roy, and Cristina Lopes. SourcererCC: Scalable and accurate clone detection. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 51–62. 2021, Springer.

[38] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. CCAligner: A token based large-gap clone detector. In *Proc. International Conference on Software Engineering*, pp. 1066–1077, 2018.

[39] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K. Roy. LVMapper: A large-variance clone detector using sequencing alignment approach. *IEEE access*, Vol. 8, pp. 27986–27997, 2020.

[40] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. NIL: Large-scale detection of large-variance clones. In *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 830–841, 2021.

[41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proc. International Conference on Neural Information Processing Systems*, pp. 3111–3119, 2013.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[43] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. GraphCodeBERT: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[44] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[45] Antti Laaksonen. *Guide to competitive programming: Learning and improving algorithms through contests*. Springer, second edition, 2020.

[46] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine

learning algorithms. *Pattern recognition*, Vol. 30, No. 7, pp. 1145–1159, 1997.

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.

[48] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.