Vol.2013-SE-182 No.28
2013/10/25

テキストマイニング技術を応用したメソッドクローン検出手法の提案

山中 裕樹[1,a)]　吉田 則裕[2,b)]　崔 恩瀞[1,c)]　井上 克郎[1,d)]

概要：ソフトウェア保守における問題の 1 つとしてコードクローン（ソースコード中に存在する同一または類似した部分を持つコード片）が指摘されている．コードクローンを検出し，共通する処理に対して親クラスへの引上げやライブラリ化といった集約を行うことによって，ソフトウェアの保守性や可読性を向上させることが可能となる．これまでの研究において様々なコードクローン検出手法が提案されてきたが，多くの手法がプログラムの構造的な類似性に着目しており，意味的に類似したコードクローンを検出することを目的とした手法は少ない．また，プログラムの意味的な類似性に着目した手法では，検出時間に膨大な時間がかかるという問題点がある．そこで本研究では，テキストマイニング技術を応用したメソッドクローン（メソッド単位のコードクローン）を検出する手法を提案する．テキストマイニングは文字列を対象としたデータマイニングのことであり，自然言語で書かれた文書の分類などに利用される．本手法ではこの技術を利用し，ソースコード中の識別子や予約語に利用される単語に対して重要度の重み付けを行うことによって，各メソッドの特徴ベクトルを計算する．そして，特徴ベクトル間の類似度を求めることによってメソッドクローンの検出を行う．本手法によって，類似した処理を行うメソッドを高速に検出することが可能であると考えられる．

キーワード：コードクローン，ソフトウェア保守，テキストマイニング

Method Clone Detection Using Text Mining Techniques

Abstract: Code clone (i.e., code fragment that has identical or similar fragment in source code) is one of the major problems for software maintenance. Software developers can increase the maintainability and the readability of source code by merging them (e.g., create library, pull up method). At present, a lot of techniques have been done on the detection of code clones in source code. However, most of them focus on structural similarities. Moreover, the detection techniques that focus on semantic similarities lack the scalability for large-scale source code. In this study, we propose a technique to detect method clones using text mining techniques (i.e., data mining technique intended for natural language text). In our approach, we generate feature vectors for each method by weighting words in identifier and syntactic keyword based on the important degree of them. And then, we detect method clones based on the similarity among the feature vectors. We believe that our technique can perform the scalable detection of similar method clones from source code.

Keywords: Code Clone, Software Maintenance, Text Mining

1. まえがき

ソフトウェア保守における問題の一つとしてコードクローンが指摘されている．コードクローンとは，ソースコード中に存在する同一または類似した部分を持つコード片のことであり，コピーアンドペーストなどの様々な理由により生成される [1]．一般的に，コードクローンの存

[1] 大阪大学
Osaka University
[2] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[a)] y-yuuki@ist.osaka-u.ac.jp
[b)] yoshida@is.naist.jp
[c)] ejchoi@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

在はソフトウェアの保守を困難にするといわれている．例えば，あるコード片に対して変更を加える場合，もしその部分がコードクローンであれば，対応する全てのコードクローンに対しても同様の変更が必要であるか否かの検討を行わなければならない．コードクローンを検出し，共通する処理に対して親クラスへの引上げやライブラリ化といった集約を行うことによって，ソフトウェアの保守性や可読性を向上させることが可能となる [2], [3].

これまでに様々なコードクローン検出手法が提案されてきたが，その多くの手法がプログラムの構造的な類似性のみに着目している [4]．そのため，同一の処理を実装しているにも関わらず，for 文と while 文の違いなど構文上の実装が異なる場合，コードクローンとして検出することができる手法は少ない．また，プログラムの意味的な類似性に着目した手法もいくつか提案されているが，検出時間に膨大な時間がかかるという問題点がある [5], [6], [7].

そこで本研究では，テキストマイニング技術を応用することによって，識別子名などの情報から意味的に処理が類似したメソッド単位のコードクローン（以下，メソッドクローン）を検出する手法を提案する．テキストマイニングとは，テキストデータを対象としたデータマイニングのことであり，自然言語で書かれた文書の分類などに利用される [8]．なお，一般的なコードクローン検出手法ではコード片単位の検出を行うが，本研究ではメソッド単位の検出を行う．コード片単位で検出を行う場合，処理の途中や曖昧なところで終了するコード片など，集約を行う事が困難であるコードクローンが多く検出されることがある．一方，メソッドでは単一または少数の機能を実装しているため，処理内容がまとまっている．従って，検出の単位をメソッドとすることによって，コード片単位で検出を行う場合に比べて集約を行うことが容易であると考えられる．

本手法では，テキストマイニング技術を用いて，ソースコード中の識別子名や構文に用いられる単語に対して重み付けを行う事によって各メソッドの特徴ベクトルの計算を行う．そして，特徴ベクトル間の類似度を計算することによってメソッドクローンの検出を行う．また，LSH(Locality-Sensitive Hashing) アルゴリズム [9] を用いて特徴ベクトルをあらかじめクラスタリングしておくことによって，類似度の計算コストを削減し，既存手法に比べて高速な検出を可能とした．

以降，2 節では，本研究の関連研究として，既存のコードクローン検出手法とテキストマイニング技術について述べる．3 節では，本研究で提案するメソッドクローン検出手法について述べる．4 節では，本手法の評価実験として，検出精度と検出時間の評価について述べる．最後に，5 節でまとめと今後の課題について述べる．

2. 関連研究

本節では，本研究の関連研究として，既存のコードクローン検出手法，および，テキストマイニング技術について述べる．

2.1 コードクローン検出

これまでの研究において様々なコードクローン検出手法が提案されてきたが，そのどれもが異なったコードクローンの定義を持つ．Roy らは，コードクローンを以下の 4 つのタイプに分類している [10].

タイプ 1: 空白の有無，レイアウト，コメントの有無などの違いを除き完全に一致するコードクローン

タイプ 2: タイプ 1 の違い加えて，変数名などのユーザ定義名，変数の型などが異なるコードクローン

タイプ 3: タイプ 2 の違いに加えて，文の挿入や削除，変更などが行われたコードクローン

タイプ 4: 同一の処理を実行するが，文の並び替えなど構文上の実装が異なるコードクローン

コードクローン検出には，主に構文の類似性に着目した検出手法 [11], [12], [13], [14] と意味的な類似性に着目した検出手法 [5], [6], [7] が存在する．構文の類似性に着目した手法では主にタイプ 1 からタイプ 3 までのコードクローンが検出の対象であるが，意味的な類似性に着目した手法では全タイプのコードクローンの検出が可能である．以降，それぞれの手法について説明する．

2.1.1 構文の類似性に着目した検出手法

構文の類似性に着目した手法として，トークンベースの検出手法 [11], [12] や，抽象構文木を用いた検出手法 [13], [14] が存在する．

トークンベースの検出ツールの代表として，Kamiya らが開発した CCFinder[11] が挙げられる．この手法では，字句解析を行うことによってソースコードをトークン列に変換し，変数名や関数名などのユーザ定義名を同一のトークンに変換する．そして，閾値以上の長さの共通トークン列を探索することによって，コードクローンの検出を行う．従って，タイプ 2 までのコードクローンの検出を行うことが可能である．

また，抽象構文木を用いた検出ツールの代表として，Jiang らが開発した DECKARD[13] が挙げられる．抽象構文木とはソースコードの構文構造を木構造で表したグラフのことを意味する．この手法では，構文解析を行うことによってソースコードを抽象構文木に変換し，類似した部分木を探索することによってコードクローンの検出を行う．従って，タイプ 3 までのコードクローンの検出を行うことが可能である．

これらの手法では，比較的高速にコードクローンの検出

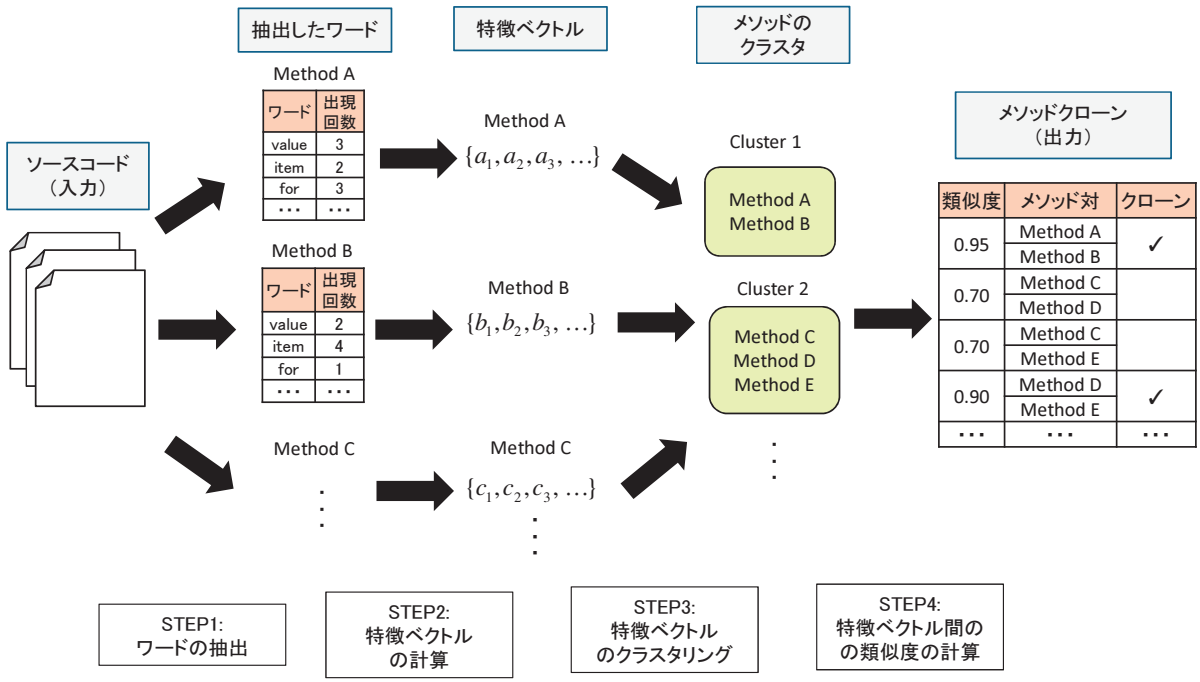

図 1 検出手法の概要
Fig. 1 Overview of our detection technique

を行うことができる．しかし，プログラムの構造的な類似性にのみ着目しているため，for 文や while 文の置き換えといったタイプ 4 のコードクローンの検出を行うことが困難である．

2.1.2 意味的な類似性に着目した検出手法

意味的な類似性に着目した手法として，プログラム依存グラフを用いた検出手法 [5], [6] や，メモリベースの検出手法 [7] などが存在する．

プログラム依存グラフを用いた検出ツールの代表として，肥後らが開発した Scorpio[5] が挙げられる．プログラム依存グラフとは，プログラム内の要素間に存在する依存関係を表した有効グラフのことを意味する．この手法では，ソースコードからプログラム依存グラフを構築し，類似した構造の部分グラフを探索することによってコードクローンの検出を行う．

また，メモリベースの検出手法の代表として，Kim らが開発した MeCC[7] が挙げられる．この手法では，静的解析を行うことによって，各手続きが終了した時点における抽象的なメモリの状態を予測する．そして，それらの状態の比較を行い，コードクローンの検出を行う．

これらの手法では，タイプ 4 までのコードクローンの検出を行うことが可能である．しかし，プログラム依存グラフを用いた手法では，グラフの頂点が膨大な数となるため，コードクローンの検出に時間がかかる．また，メモリベースの手法においても，メモリの状態の予測を行うための静的解析に膨大な時間がかかる．

本研究では，識別子名などに基づいてソースコード中の各メソッドを特徴ベクトルに変換し，それらの類似度を計算することによってタイプ 4 までのコードクローンの検出を行う．また，あらかじめ特徴ベクトルをクラスタリングしておくことによって，既存のタイプ 4 のコードクローン検出手法に比べて高速な検出を行うことが可能である．

2.2 テキストマイニング

テキストマイニングとは，テキストデータを対象にしたデータマイニングのことである．テキストマニングでは，定型化されていないドキュメントの集まりを自然言語解析の手法を用いて単語やフレーズに分割し，それらの出現頻度や相関関係を分析することによって有用な情報を抽出する [8]．

Uramoto らは，テキストマイニングを行い，互いに類似した内容が記載されているドキュメントの分類を行っている [15]．この手法では，自然言語処理解析を行うことによって，各ドキュメントから単語の抽出を行う．次に，単語の出現頻度や希少さに基づいて，ドキュメント中に存在する各単語に対して重要度の重み付けを行い，それらの値を特徴量として各ドキュメントを特徴ベクトルに変換する．そして，特徴ベクトル間の類似度を計算することによって，互いに内容が類似したドキュメントのクラスタリングを行っている．

表 1　識別子名の分割例
Table 1　Examples of segmentation of identifier

識別子	ワードへの分割
value_of_item	value, of, item
itemValue	item, value

Uramoto らの手法では分類の対象が新聞であるが，本研究ではソースコード中の各メソッドを対象とする．すなわち，各メソッドに存在する識別子名や予約語に利用される単語に対して重み付けを行い，それらの値を特徴量として利用する．

3. メソッドクローン検出手法

本節では，本研究で提案するメソッドクローン検出手法の概要について説明する．本手法ではテキストマイニング技術を応用することによって，ソースコード中の**ワード**から意味的に処理が類似したメソッドクローンを高速に検出することを目的とする．ここでワードとは，以下の 2 つを対象とする．

- 変数や関数などに付けられた識別子名を構成する単語
- 条件文や繰り返し文などの構文に利用される予約語を構成する単語

図 1 は本手法の概要を表している．本手法ではソースコードを入力とし，クローンペア（互いに処理の内容が類似したメソッドの対）の集合をリストとして出力する．本手法は以下の 4 つのステップから構成される．

STEP1: ソースコード中の各メソッドからワードの抽出を行う．
STEP2: STEP1 で抽出したワードに重み付けを行うことによって，各メソッドの特徴ベクトルを計算する．
STEP3: LSH アルゴリズム [9] を用いて，STEP2 で求めた特徴ベクトルのクラスタリングを行う．
STEP4: STEP3 で求めたクラスタにおいて，特徴ベクトル間の類似度の計算を行い，メソッドクローンを検出する．

以降の節で，それぞれのステップの詳細について説明する．

3.1 STEP1:ワードの抽出

まず最初に，ソースコードの各メソッドに含まれる識別子名や構文に利用される予約語から，ワードの抽出を行う．

識別子名が複数の単語から構成される場合，以下の方法でワード単位に分割する．

- ハイフンやアンダースコアなどの区切り記号（デリミタ）による分割
- 識別子名中の大文字になっているアルファベットによる分割

識別子名の分割例を**表 1** に示す．また，2 文字以下の識別子名に対してはそれらをまとめて 1 つのワードとして扱う．この理由は，繰り返し文などによく利用される “i” や “j” といった意味情報が込められない変数を全て同一のものとして扱うためである．このように識別子の情報を利用することによって，各メソッドが実装する機能を表すことができると考えられる．これまでの研究においても，識別子にその実態を表す名前を付けることがソフトウェアの品質を保つ上で重要であることが指摘されている [16], [17], [18].

また，構文に利用される単語とは，条件文に用いられる “if” や “switch”，繰り返し文に用いられる “for” や “while” といった予約語のことを示しており，それらをワードとして扱う．

3.2 STEP2:特徴ベクトルの計算

次に，STEP1 で抽出したワードに重み付けを行うことによって，各メソッドから特徴ベクトルの計算を行う．ここでは，TF-IDF 法 [19] を利用して各ワードの重みを計算し，その値を特徴量として利用する．TF-IDF 法はテキストマイニングにおいて，文書の類似性の判定や情報抽出に利用されている．TF-DIF 法による値は tf 値（メソッド中のワードの出現頻度）と idf 値（メソッド集合中のワードの希少さ）の積で与えられる．ワード w の重み v_w の計算式を以下に示す．

$$tf_w = \frac{\text{メソッド中のワード } w \text{ の出現回数}}{\text{メソッド中に出現する全ワードの出現回数の合計}}$$

$$idf_w = \log \frac{\text{全メソッド数}}{\text{ワード } w \text{ が出現するメソッド数}}$$

$$v_w = tf_w idf_w$$

本手法では，全メソッド中の各ワードに対して重要度の重みを計算し，それらを特徴量として用いることによって，特徴ベクトルを求める．従って，各メソッドの特徴ベクトルの次元はソースコード中に存在する全ワードの数となる．

3.3 STEP3:特徴ベクトルのクラスタリング

このステップでは，STEP2 で計算した各メソッドの特徴ベクトルに対してクラスタリングを行うことによって，クローンペアと成り得る候補を絞ることを目的とする．

ここでは，近似最近傍探索アルゴリズムの一種である LSH(Locality-Sensitive Hashing) アルゴリズム [9] を用いて特徴ベクトルのクラスタリングを行う．LSH アルゴリズムは，高次元ベクトル空間において，ハッシュ関数を用いることによって確率的に近傍点探索を高速に行う手法である．なお，本手法では LSH アルゴリズムの実装である E2LSH[*1] [20] を利用している．

データセットの次元を d，データセットの数を n，確率に関するパラメータを ρ としたとき，LSH のクラスタリ

[*1] http://www.mit.edu/~andoni/LSH/

表 2　ベンチマークによる検出精度の評価
Table 2　Evaluation of detection accuracy using benchmark set

	タイプ 1	タイプ 2	タイプ 3	タイプ 4
ベンチマーク	3	4	5	4
検出結果	3	2	5	4

ングの時間計算量は $O(dn^{\rho}\log n)$ と表される．一方，全メソッドに対して特徴ベクトル間の類似度を計算する場合の時間計算量は $O(dn^2)$ となる．従って，本ステップであらかじめクラスタリングを行い，クローンペアと成り得る候補を絞ることによって，検出時間にかかる計算コストを削減できると考えられる．

3.4　STEP4:特徴ベクトルの類似度の計算

最後に，STEP3 で求めた各クラスタ中のメソッドの対に対して，コサイン類似度を用いてクローンペアであるか否かの判定を行う．コサイン類似度は多次元ベクトルの類似度を測定するものであり，次元が d である 2 つの特徴ベクトル $\vec{a},\vec{b}$ 間の類似度は以下の式で表すことができる．

$$sim(\vec{a},\vec{b}) = cos(\vec{a},\vec{b}) = \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} {a_i}^2}\sqrt{\sum_{i=1}^{d} {b_i}^2}}$$

TF-IDF 法の計算式から分かるように，特徴量は常に正の値となるため，コサイン類似度は 0 から 1 の範囲となる．もし，コサイン類似度が閾値以上であれば，それら 2 つのメソッドはクローンペアであると判定する．本手法では，類似度の閾値として 0.9 を利用している．

このように，テキストマイニング技術を応用して各メソッドに出現するワードから特徴ベクトルの計算を行い，それらの類似度を求めることによって，タイプ 4 のメソッドクローンの検出が可能となる．本手法では，各クローンペアに対して類似度を計算しているため，類似度によるメソッドクローンのランキング付けも可能である．また，あらかじめ特徴ベクトルのクラスタリングを行い，クローンペアと成り得る候補を絞ることによって，より高速なメソッドクローンの検出を実現できる．

4.　評価実験

本節では，3 節で述べたメソッドクローン検出手法の評価実験について述べる．本実験では，ベンチマークを用いた検出精度の評価と，検出時間の評価を行った．4.1 節と 4.2 節で，それぞれの詳細な実験手法と結果について述べる．また，4.3 節では，MeCC[7] と比較を行い，本実験の考察について述べる．

4.1　検出精度

検出精度の評価では，Roy らのベンチマーク [10] を用いた実験と，OSS への適用実験を行った．なお，メソッドクローンとして検出される類似度の閾値は，本手法のデフォルト値である 0.9 を利用した．

表 3　提案手法の適用対象の OSS
Table 3　OSS for application of our detection technique

OSS	規模	概要
Apache Ant	109KLOC	アプリケーションサーバ
ArgoUML	192KLOC	UML モデリングツール

表 4　OSS を用いた検出精度の評価
Table 4　Evaluation of detection accuracy using OSS

OSS	検出クローンペア数	適合率	再現率
Apache Ant	474 個	0.92	0.62
ArgoUML	880 個	0.96	0.55

表 5　メソッドクローンのタイプ別の検出数
Table 5　Statistics of method clones by type

	タイプ 1	タイプ 2	タイプ 3	タイプ 4
Apache Ant	56	139	220	22
ArgoUML	222	219	371	33

4.1.1　Roy らのベンチマークを用いた評価

最初に，Roy らのベンチマーク [10] を用いた検出精度の評価実験について説明する．このベンチマークでは，タイプ 1 からタイプ 4 までの合計 16 個のクローンペアが用意されており，文献 [7] の評価実験でも利用されている．

表 2 は，ベンチマークで用意されているクローンペアのタイプ毎の個数と，本手法で検出できたクローンペアのタイプ毎の個数を示している．結果として，全体で 16 個中 14 個のクローンペアを検出することができた．また，タイプ 1，タイプ 3，タイプ 4 において，本手法を用いて全てのクローンペアを検出することを確認できた．タイプ 4 では，文の並び替えや for 文と while 文の繰り返し処理文の置き換えが存在するクローンペアが用意されているが，本手法を用いることによってそれらをコードクローンとして検出することができた．

一方で，タイプ 2 では，2 つのクローンペアの検出を行うことができなかった．これらのクローンペアは，元の変数名が 1 文字のアルファベットに省略されており，意味をもたない変数名に変換されていたため，本手法ではコードクローンとして検出できなかったと考えられる．しかし，実際のソフトウェア開発では，変数名を意味のないものに置換する場合は少ないと考えられる．

4.1.2　OSS への適用

次に，Java 言語で実装された 2 つの OSS（**表 3**）に対して本手法を適用し，Tempero らのベンチマーク [21] を用いて適合率と再現率の評価を行った．適合率は全検出結果に対して，ベンチマークで正解集合と判定されているクローンペアの割合を表している．また，再現率は，ベンチマーク中のクローンペアに対して，本手法によって検出された

```
1:  private void fireTargetSet(TargetEvent targetEvent){
2:      if (listenerList == null) {
3:          listenerList=collectTargetListeners(this);
4:      }
5:      Object[] listeners=listenerList.getListenerList();
6:      for (int i=listeners.length - 2; i >= 0; i-=2) {
7:          if (listeners[i] == TargetListener.class) {
8:              ((TargetListener)listeners[i+1]).targetSet(targetEvent);
9:          }
10:     }
11: }
```

(a) 変数の null チェック有り

```
1:  private void fireTargetSet(TargetEvent targetEvent){
2:      Object[] listeners=listenerList.getListenerList();
3:      for (int i=listeners.length - 2; i >= 0; i-=2) {
4:          if (listeners[i] == TargetListener.class) {
5:              ((TargetListener)listeners[i+1]).targetSet(targetEvent);
6:          }
7:      }
8:  }
```

(b) 変数の null チェック無し

図 2 タイプ 3 のメソッドクローンの例（ArgoUML）

Fig. 2 Examples of detected type 3 method clones in ArgoUML

```
1:  public void log(String message,int loglevel){
2:      if (managingPc != null) {
3:          managingPc.log(message,loglevel);
4:      }else {
5:          if (loglevel > Project.MSG_WARN) {
6:              System.out.println(message);
7:          }else {
8:              System.err.println(message);
9:          }
10:     }
11: }
```

(a) if 文を用いた条件分岐処理

```
1:  public void log(String message,int loglevel){
2:      if (managingPc != null) {
3:          managingPc.log(message,loglevel);
4:      }else {
5:          (loglevel > Project.MSG_WARN ? System.out :
                                  System.err).println(message);
6:      }
7:  }
```

(b) 三項演算子を用いた条件分岐処理

図 3 タイプ 4 のメソッドクローンの例（Apache Ant）

Fig. 3 Examples of detected type 4 method clones in Apache Ant

クローンペアの割合を表している．

結果を**表 4** に示す．実験の対象とした 2 つの OSS に対して再現率は 6 割前後であるが，適合率は 9 割を超えており，誤検出をほとんど含まずメソッドクローンを検出することができた．

さらに，検出精度の評価で正解集合と判定したクローンペアを手作業で各タイプに分類した．結果を**表 5** に示す．この表から分かるように，本手法を用いることによって，

```
1:  private boolean isValidEventRemove(PropertyChangeEvent e){
2:      boolean valid=false;
⋮           ⋮
9:          Collection col=(Collection)getChangedElement(e);
10:         Iterator it=col.iterator();
11:         if (!col.isEmpty()) {
12:             valid=true;
13:             while (it.hasNext()) {
14:                 Object o=it.next();
15:                 if (!isValidElement(o)) {
16:                     valid=false;
17:                     break;
18:                 }
19:             }
20:         }else {
⋮           ⋮
```

(a) while 文を用いた繰り返し処理

```
1:  private boolean isValidEvent (PropertyChangeEvent e){
2:      boolean valid=false;
⋮           ⋮
9:          Collection col=(Collection)getChangedElement€;
10:         if (!col.isEmpty()) {
11:             valid=true;
12:             for (      Object o : col) {
13:                 if (!isValidElement(o)) {
14:                     valid=false;
15:                     break;
16:                 }
17:             }
18:         }else{
⋮           ⋮
```

(b) for 文を用いた繰り返し処理

図 4 タイプ 4 のメソッドクローンの例（ArgoUML）

Fig. 4 Examples of detected type 4 method clones in ArgoUML

タイプ 1 からタイプ 4 までの全てのメソッドクローンの検出を行うことができた．

図 2 は，本手法によって検出したタイプ 3 のメソッドクローンの例である．図 2(a) では変数 listenerList の null チェックが行われているが，図 2(b) では行われていない．この例のように，本手法を用いて null チェック漏れなど，不具合に直接関わるメソッドクローンを検出できることを確認できた．

また，検出されたタイプ 4 のメソッドクローンとしては，文の並び替え，if 文と三項演算子を用いた条件分岐処理の置き換え，for 文と while 文を用いた繰り返し処理の置き換えなどの違いが存在するメソッドクローンを検出することができた．**図 3**，**図 4** に本手法で検出したタイプ 4 のメソッドクローンの例を示す．

表 6　検出時間の評価
Table 6　Evaluation of detection time

OSS	規模	検出時間（パラメータ計算有）	検出時間（パラメータ計算無）
Python	435KLOC	294 秒	83 秒
Apache HTTPD	343KLOC	213 秒	68 秒
PostgreSQL	937KLOC	447 秒	196 秒

図 3 は，条件分岐処理が置き換わっている例である．図 3(a) では if-else 文を用いて標準出力と標準エラー出力の条件分岐処理を実装しているが，図 3(b) では三項演算子を用いて条件分岐処理を実装している．図 4 は，繰り返し岐処理が置き換わっている例である．図 4(a) では 13-19 行目で while 文を用いて繰り返し処理を実装しているが，図 4(b) では 12-17 行目で for 文を用いて全く同一の繰り返し処理を実装している．これらの例のように，本手法を用いて，同一または類似している処理を実装しているにも関わらず，構文上の実装が異なるメソッドクローンを検出できることを確認できた．

4.2　検出時間

検出時間の評価として，本実験では，MeCC[7] のスケーラビリティの評価で用いられている C 言語で実装された 3 つの OSS（Python，Apache HTTPD，PostgreSQL）に対して適用を行った．なお，本実験は以下のワークステーションの下で行った．

- OS: Windows 7 64-bit
- CPU: Intel Xeon 2.40GHz
- RAM: 16.0GB

結果を**表 6** に示す．表中のパラメータ計算とは，特徴ベクトルのクラスタリングで利用している E2LSH に対するものである．E2LSH は LSH のパラメータを自動的に決定する機能を持つ．この機能はデータセットの中からいくつかのデータをランダムに選択し，その値の傾向を見て適当なパラメータを自動的に設定する．従って，1 つの適用対象に対してパラメータの値を一度計算すれば，その後の検出においてもその値を再利用することが可能であると考えられる．

結果として，パラメータ計算を行う場合において，500 秒以下でメソッドクローンを検出できることを確認できた．また，パラメータ計算を行わない場合においては，全ての OSS に対して 200 秒以下でメソッドクローンを検出できることを確認できた．

4.3　考察

本実験の考察として，本手法同様にタイプ 4 のメソッドクローンの検出を目的とする MeCC[7] と比較を行った，

検出精度の評価において，本手法では，Roy らのベンチマークを用いた実験で 16 個中 14 個のクローンペアを検出することができた（表 2）．また，OSS への適用実験では適合率が 9 割以上と高い値であることがわかった（表 4）．一方 MeCC では，Roy らのベンチマークを用いた実験で 16 個 15 個のクローンペアの検出に成功している．しかし，3 つの OSS に対して適用実験を行っているが，適合率は全て 9 割未満と本手法に比べて低い．本手法では集約の対象となるメソッドクローンを効率よく開発者に提示することが目的であるため，適合率が高いことは非常に重要である．従って，本手法の検出精度は高いと言える．

また，検出時間の評価では，LSH アルゴリズムのパラメータを計算する場合で 500 秒以下，パラメータの計算を行わない場合においては 200 秒と現実的な時間でメソッドクローンを検出できることを確認できた（表 6）．MeCC における評価実験と本手法の実行環境は異なっているが，MeCC では 3 つの OSS に対して静的解析に 1 時間以上かかるため，本手法は高速にクローン検出を行うことが可能であると言える．

ただし，本実験では検出精度の評価に，Java 言語で実装された 2 つの OSS に対してのみ適用を行っている．そのため，他の言語で実装された OSS に対しても適用し，検出精度の評価を行う必要がある．また，検出時間の評価についても，さらに規模が大きい OSS に対して適用を行い，評価を行う必要がある．

5.　まとめと今後の課題

本研究では，テキストマイニング技術を応用したメソッドクローン検出手法の提案を行った．本手法では，ソースコード中の各ワードに対して TF-IDF 法を用いて重みを計算し，それらを特徴量として各メソッドの特徴ベクトルを計算する．そして，特徴ベクトル間の類似度を計算することによって，意味的に処理内容が類似したメソッドクローンの検出を行う．また，LSH アルゴリズムを用いてあらかじめ特徴ベクトルのクラスタリングを行うことによって，高速なメソッドクローンの検出を実現した．

評価実験では，検出精度の評価として，Roy らのベンチマークと Java 言語で実装された 2 つの OSS に対して適用を行い，高い精度で検出できることを確認した．さらに，繰り返し処理や条件分岐処理の実装が異なっているタイプ 4 のメソッドクローンの検出を行う事を確認できた．また，C 言語で実装された 3 つの OSS に対して検出時間の評価を行い，既存手法に比べて高速にメソッドクローンを検出

できることを確認できた．

今後の課題として，以下が挙げられる．

- 再現率の精度を向上させる必要がある．具体的には，類義語や同位語の関係を用いたワードのクラスタリングや，ワードの品詞による重み付けを行うことによって，リネームが行われた場合のメソッドクローンの検出精度を向上させる．
- 現状では Java 言語と C/C++言語にのみ対応しているが，手法を拡張させて COBOL など他の言語にも対応する必要がある．
- 他の OSS に対して適用し，既存手法と比較しながら本手法の有用性を評価する必要がある．

謝辞 本研究は JSPS 科研費 25220003，21240002 の助成を得たものである．

参考文献

[1] 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会論文誌，Vol. J91-D, No. 6, pp. 1465–1481 (2008).
[2] Fowler, M.: *Refactoring: improving the design of existing code*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999).
[3] 肥後芳樹，神谷年洋，楠本真二，井上克郎：コードクローンを対象としたリファクタリング支援環境，電子情報通信学会論文誌，Vol. 88, No. 2, pp. 186–195 (2005).
[4] Rattan, D., Bhatia, R. and Singh, M.: Software clone detection: a systematic review, *Information and Software Technology*, Vol. 55, pp. 1165–1199 (2013).
[5] 肥後芳樹，楠本真二：プログラム依存グラフを用いたコードクローン検出法の改善と評価，情報処理学会論文誌，Vol. 51, No. 12, pp. 2149–2168 (2010).
[6] Komondoor, R. and Horwitz, S.: Using slicing to identify duplication in source code, *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pp. 40–56 (2001).
[7] Kim, H., Jung, Y., Kim, S. and Yi, K.: MeCC: memory comparison-based clone detector, *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 301–310 (2011).
[8] Hearst, M. A.: Untangling text data mining, *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ACL '99, pp. 3–10 (1999).
[9] Indyk, P. and Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality, *Proceedings of the thirtieth annual ACM Symposium on Theory of Computing*, STOC '98, pp. 604–613 (1998).
[10] Roy, C. K., Cordy, J. R. and Koschke, R.: 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).
[11] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).
[12] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192 (2006).
[13] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: scalable and accurate tree-based detection of code clones, *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 96–105 (2007).
[14] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone detection using abstract syntax trees, *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pp. 368–377 (1998).
[15] Uramoto, N. and Takeda, K.: A method for relating multiple newspaper articles by using graphs, and its application to Webcasting, *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics*, ACL '98, pp. 1307–1313 (1998).
[16] Kernighan, B. W. and Pike, R.: *The practice of programming*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999).
[17] Hunt, A. and Thomas, D.: *The pragmatic programmer: from journeyman to master*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999).
[18] McConnell, S.: *Code complete, Second Edition*, Microsoft Press, Redmond, WA, USA (2004).
[19] 徳永健伸：情報検索と言語処理，東京大学出版会 (1999).
[20] Andoni, A. and Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, *Communications of the ACM*, Vol. 51, No. 1, pp. 117–122 (2008).
[21] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. and Noble, J.: The qualitas corpus: a curated collection of Java code for empirical studies, *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, APSEC '10, pp. 336–345 (2010).