順伝播型ニューラルネットワークを用いた類似コードブロック検索の試み

藤原 裕士[1,a)]　崔 恩瀞[2]　吉田 則裕[3]　井上 克郎[1]

概要：Ichi Tracker は，入力として与えられたコード片と類似し，再利用可能なコード片をオープンソースソフトウェアから高い精度で検索するシステムである．しかし，このシステムは入力コード片に類似しているコード片でも，構文上の差異が存在する場合，検索結果から削除されてしまう可能性がある．本研究では, この問題を解決するために順伝播型ニューラルネットワークを用いて，入力コード片と類似したコードブロックの検索を試みる．本検索手法は，事前にソースコードに対してミューテーションを行い，様々な種類の類似ソースコードを作成した後，元ソースコード及び類似ソースコードのコードブロックを抽出する．また，抽出した各コードブロックを特徴ベクトルに変換し，類似ソースコードのグループ毎にユニークなラベルを付与し，教師あり学習で順伝播型ニューラルネットワークのモデルを作成する．入力コード片の類似コードブロックを検索する際は，作成したモデルへ，入力コード片から作成した特徴ベクトルを与え，モデルが出力したラベルに該当するグループのコードブロックを出力する．評価実験では，3 つのオープンソースソフトウェアに対して本検索手法を適用し，構文的に差異がある類似コードブロックを高い精度で検索することが確認できた．

キーワード：コード検索，機械学習，順伝播型ニューラルネットワーク，BoW，doc2vec

1. まえがき

既存ソフトウェアは，ソフトウェア開発における重要な資源である．既存ソフトウェアのコード片を再利用することで，ソフトウェア開発の生産性および信頼性の向上が期待できる．そのため，新規ソフトウェアを開発する段階において，再利用可能なコード片を特定することは重要である．また，再利用可能なコード片を OSS や Q&A サイト（例：Stack Overflow[*1]）等から特定したときに，コード片をクエリとした類似コード片検索を行うと，オリジナルのコード片のライセンス記述を調べることや，より優れた脆弱性対策がされているコード片を探し出すことができ，再利用の安全性が高まる．

Ichi Tracker[1] は，入力として与えられたコード片と類似しているコード片をオープンソースソフトウェア（以下 OSS）から検索するシステムである．このシステムでは，入力コード片に含まれる識別子単語の数を測定し，インターネット上のコード検索エンジンにクエリとして渡す．そして，コード検索エンジンから返された結果に対して，トークン解析ベースコードクローン検出ツール CCFinder[2] を適用してフィルタリングを行うことによって，構文的に一致したコード片を検索する．しかし，Ichi Tracker は，CCFinder を用いて検索結果のフィルタリングを行うため，入力コード片に類似しているコード片でも，文が追加されていたり削除されていたりするなど，構文上の差異が存在する場合，検索結果から削除されてしまう可能性がある．開発者は，構文的に差異があるコード片も再利用する可能性があるが，Ichi Tracker は，これらのコード片を検索できない可能性がある．

本研究では，この問題点を解決するために順伝播型ニューラルネットワーク（以下 FFNN）を用いた機械学習モデルの利用を試みる．本検索手法は，構文的に一致するコード片だけでなく，構文的に類似したコード片も検索結果に含めることができる．本検索手法はある 2 つの要素の対応付けが得意な機械学習を使用することで，入力コード片と構文的に類似したコード片を容易に対応させることができる．

本検索手法は，学習を行う STEP A と検索を行う STEP B の 2 段階で構成されている．STEP A では，まずソー

[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
[3] 名古屋大学
Nagoya University
[a)] y-fujiwr@ist.osaka-u.ac.jp
[*1] https://stackoverflow.com/

スコードに対してミューテーションを行い，様々な種類の類似ソースコードを作成した後，元ソースコード及び類似ソースコードのコードブロックを抽出する．その後，抽出した各コードブロックを特徴ベクトルに変換し，類似コードブロックのグループ毎にユニークなラベルを付与し，教師あり学習を実行することで FFNN のモデルを作成する．

STEP B では，作成したモデルへ，入力コード片から作成した特徴ベクトルを与える．その結果，入力コード片に類似している学習済コードブロックが存在する場合は，類似している学習済コードブロックに付与されたラベルを出力する．そのため，出力されたラベルを確認することで，入力コード片に類似している学習済コードブロックを検索することができる．

評価実験では，3 つのオープンソースソフトウェアに対して本検索手法を適用した．その結果，本検索手法は構文的に差異がある類似コードブロックを高い精度で検索することが確認できた

以降，2 章では本研究の背景について述べる．3 章では，本研究で提案する手法について述べる．4 章では，本研究の評価実験について述べる．5 章では，関連研究について述べる．最後に，6 章でまとめと今後の課題について述べる．

2. 背景

本章では，本研究の背景として，Roy らのミューテーション [3]，FFNN，既存の類似コード検索システム Ichi Tracker の説明し，その後，Ichi Tracker の問題点について述べる．

2.1 Roy らのミューテーション

Roy らのミューテーション [3] とは，機械的にソースコードを変更することによって，構文的に一致または類似したソースコードを作成することである．Roy らは，ソースコードの変更方法をミューテーションオペレータと呼び，13 種類のミューテーションオペレータを定義している．

mCW： 空白の数を変更する．
mCC： コメントを変更する．
mCF： 改行などのコーディングスタイルを変更する．
mSRI： 変数名などのユーザー定義名，変数の型などを規則的に変更する．
mARI： 変数名などのユーザー定義名，変数の型などを不規則的に変更する．
mRPE： 変数単体の式を別の式に置き換える．
mSIL： ある文にわずかな挿入を行う．
mSDL： ある文の一部を削除する．
mILs： いくつかの文を挿入する．
mDLs： いくつかの文を削除する．
mMLs： いくつかの文を修正する．
mRDS： いくつかの宣言文を並べ替える．

```
1  public static  void BubbleSort()
2  {
3     int temp;
4     for (int j = 0; j < num.length - 1; j++) {
5        if (num[j] > num[j + 1]) {
6           temp = num[j];
7           num[j] = num[j + 1];
8           num[j + 1] = temp;
9        }
10    }
11 }
```

(a)

```
1  public static  void BubbleSort()
2  {
3     int temp;
4     for (int j = 0; j < num.length - 1; j++) {
5        if (num[j] > num[j + 1]) {
6           ;
7           num[j] = num[j + 1];
8           num[j + 1] = temp;
9        }
10    }
11 }
```

(b)

図 1 ミューテーションの例 (mSDL)

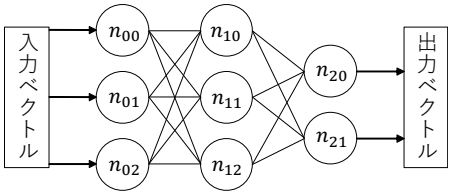

図 2 FFNN の例

mROS： いくつかの宣言文以外の文を並べ替える．
mCR： 制御構造を別のものに置き換える．

図 1 はミューテーションオペレータ mSDL をソースコードに適用した例をに示す．図 1（a）の 6 行目の文の一部を削除することで，元のソースコードと構文的に類似したソースコード（図 1（b））が作成された．

2.2 FFNN

FFNN とは，歴史上初めて登場したニューラルネットワークであり，ニューロンと呼ばれる，並列に動作する単純な要素を接続することで構成されている [4]．図 2 は n_{00}〜n_{21} で構成される，3 層の FFNN の例である．FFNN の入力と出力はともにベクトルであり，そのベクトルの次元はネットワーク構造に依存する．図 2 の例では入力が 3 次元，出力が 2 次元のベクトルである．また，ネットワークの機能は，各ニューロン間の接続に設定されている重みによって決まる．入力ベクトルと出力ベクトルの組を FFNN に与え，その入力ベクトルを FFNN に与えた際にその出力ベクトルが出力されるように重みの値を調整することによって，入力ベクトルと出力ベクトルを対応づける FFNN への訓練ができる．このように訓練した FFNN に，学習させた入力ベクトルに類似したベクトルを入力として与えた場合，学習させた出力ベクトルに類似したベクトルが出力される．従って，出力ベクトルを分析することによって，入力ベクトルを特徴ごとに分類することができる．

2.3 Ichi Tracker

我々の研究グループが開発した Ichi Tracker[1] は，入力として与えられたコード片を，インターネット上の OSS リポジトリから検索するためのシステムである．

2.3.1 Ichi Tracker の類似コード検索方法

この節では，Ichi Tracker の類似コード検索方法の概要

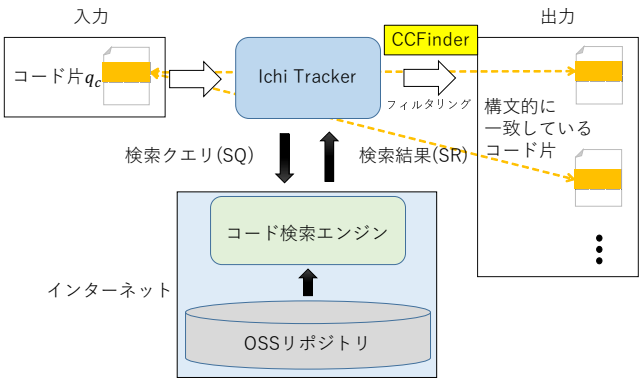

図 3 Ichi Tracker の概要

について説明する．図 3 は Ichi Tracker の概要を表している．

Ichi Tracker は入力として与えられたコード片 q_c で使用されている識別子を解析することで検索クエリ SQ を作成し，SPARS/R[5]，Koders[6] などの，インターネット上のコード検索エンジンに渡すことで，インターネット上の OSS リポジトリを対象とした検索を実行する．そして，コード検索エンジンから検索結果 SR を渡されると，OSS リポジトリから SR に含まれるコード片 $sr_1, sr_2, \ldots$ をダウンロードし，q_c と $sr_1, sr_2, \ldots$ が構文的に一致しているかを CCFinder[2] を用いて確認し，一致しているコード片を出力する．

2.3.2 Ichi Tracker の問題点

開発者は，文の追加や削除が行われている構文的に差異があるコード片も再利用する可能性がある．しかし，Ichi Tracker は，コード検索エンジンから渡された検索結果 SR を CCFinder を利用してフィルタリングをするため，SR が入力コード片と構文的に異なる場合，その検索結果を出力しない可能性がある．この問題の解決方法として，検索結果のフィルタリングの際に CCFinder 以外のコードクローン検出ツールを使用する方法も考えられるが，コードクローン検出ツールは自らの類似度の定義に基づいてコードクローンを検出するため，特定のコード片を検索できない可能性が存在する．この問題を解決するために，本研究では，FFNN を用いた機械学習モデルにより，構文的に一致するコード片だけでなく，構文的に類似したコード片も検索できる手法を提案した．また，機械学習を用いると，検索に失敗したコード片を新たに学習させることで次からは正しく検索することができる．

3. 提案手法

本研究では，FFNN を使用することで，コードブロック単位の類似コード片検索を試みる．本検索手法は機械学習を使用することで，ある 2 つの要素の対応付けを学習データから経験的に取得することができ，入力として与えられたコード片と構文的に類似したコード片を容易に対応させることが可能である．また，機械学習を用いることで構文上類似しているが検索できなかったコード片を新たに学習させることで次からは正しく判定でき，検索漏れや誤検索を減らすことができる．

本検索手法は，FFNN による機械学習を行う STEP A と，コードブロックの検索を行う STEP B の 2 段階で構成されている．

3.1 用語の定義

コードブロック： 本検索手法では，以下の 2 つの条件のいずれかを満たすコード片をコードブロックと定義する．

条件 1 関数の“{ }”で囲まれた範囲

条件 2 if，else，for，while，do-while，switch 文の“{ }”で囲まれた範囲

類似度： 本検索手法では，2 つのソースコードを正規化した後に抽出した行の集合を $t1$,$t2$ とし，$t1, t2$ 内で重複している行を $t1 \cap t2$ としたとき，類似度 $Similarity$ を以下のように定義する．

$$Similarity = \frac{2 * |t1 \cap t2|}{|t1| + |t2|}$$

類似コードブロック・類似コードブロックセット： 本検索手法では，構文的に一致，または差異はあるが一致部分も存在する（類似度が 0 より大きい）コードブロックを類似コードブロックと定義する．また，類似コードブロックの同値類を類似コードブロックセットと定義する．

ネガティブデータ： 類似コード検索を行う際，類似コードブロックを出力するという事象の他に，検索結果なしという事象が存在するが，本検索手法では，FFNN を使用し，コード検索を分類問題へと置換しているため，学習の際に，分類における“検索結果なし”クラスを作成するためのベクトルが必要となる．本検索手法では，以下の条件 3 に当てはまるコードブロックから作成した特徴ベクトルをネガティブデータとして定義し，“検索結果なし”クラスを作成するためのベクトルとして学習に使用する．

条件 3 検索対象のリポジトリ内には存在せず，ポジティブデータすべてと構文的に類似していないため，特徴ベクトルに“検索結果なし”を意味するラベル 0 が付与されるコードブロック

ポジティブデータ： 本検索手法では，以下の条件 4 に当てはまるコードブロックから生成された特徴ベクトルをポジティブデータとして定義する．

条件 4 検索対象のリポジトリ内に存在し，特徴ベクトルに 0 以外のラベルが付与されるコードブロックと，その類似コードブロック

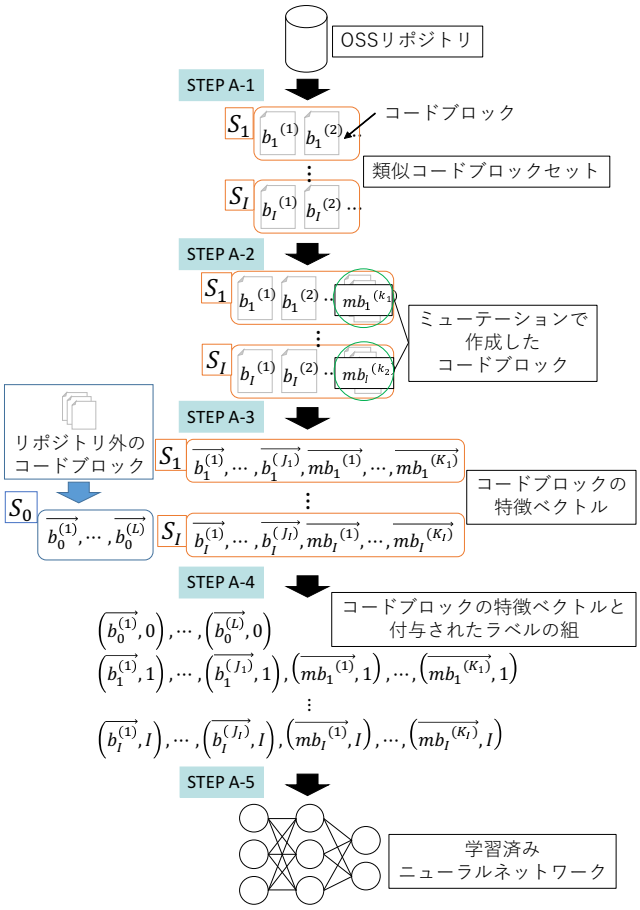

図 4　STEP A の概要図

3.2　FFNN による機械学習

この節では，FFNN による機械学習を行う STEP A について説明する．STEP A では，OSS リポジトリから取得したソースコードを基に学習データを作成し，そのデータを用いて FFNN のモデルを作成する．STEP A は 5 つの手順から構成されている．STEP A の概要を図 4 に示す．

STEP A-1

OSS リポジトリ内のソースコードに対して構文解析を行い，コードブロックを抽出した後，我々のグループで開発している類似コード片検出ツール CCFinder[2] および，横井らのブロッククローン検出ツール [7] を使用し，類似コードブロックセット $S_i(1 \leq i \leq I)$ を作成する．類似コードブロックセット S_i には構文的に類似したコードブロック ${b_i}^{(j)}(1 \leq j \leq J)$ が属する．また，ネガティブデータ (3.1 節) 生成用のソースコードを用意し，コードブロック ${b_0}^{(l)}(1 \leq l \leq L)$ を抽出する．

STEP A-2

類似コードブロックセット S_i 内のコードブロック ${b_i}^{(j)}$ を構文的な差異が含まれるように変更したコードブロック ${mb_i}^{(k)}(1 \leq k \leq K)$ を，2.1 節で説明した 13 種類のミューテーションオペレータを適用して作成する．このとき，${mb_i}^{(k)}(1 \leq k \leq K)$ は，類似コードブロックセット S_i に属する．

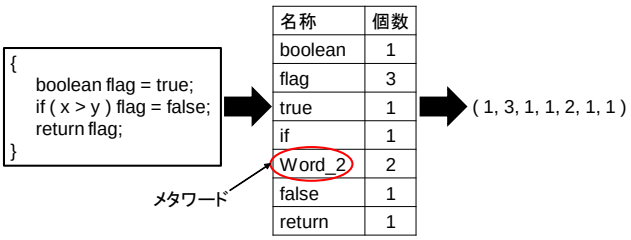

図 5　BoW をコード片に適用した例

STEP A-3

コードブロック ${b_i}^{(j)}$，${mb_i}^{(k)}$ を特徴ベクトルに変換する．その際に用いた手法については 3.2.1 節で説明する．

STEP A-4

類似コードブロックセット S_i に属するコードブロックから生成された特徴ベクトルに対して，ラベル i を付与する．また，コードブロック集合 ${b_0}^{(l)}$ には，OSS リポジトリ内に類似コードブロックはないという意味で，特徴ベクトルにラベル 0 を付与する．

STEP A-5

特徴ベクトルと，付与したラベルを用いて，教師あり学習を行い，FFNN のモデルを作成する．

3.2.1　特徴ベクトルの計算

本研究における評価実験では，機械学習に用いられる代表的なベクトル化方式の中で最も単純な BoW（Bag of Words）と，自然文書を対象とした機械学習において有効性が実証されているベクトル化方式である doc2vec[8] の 2 種類の手法を使用してそれぞれモデルを作成した．

BoW:　学習データ用コードブロックに現れる各予約語・識別子の数を特徴量として，各コードブロックを特徴ベクトルに変換する．その際，2 字以下の識別子はメタワードとして，まとめて数える．

また，検索するコード片を特徴ベクトルに変換する際は，学習データ用コードブロックに現れていた各予約語・識別子がそのコード片に現れている数を特徴量としてベクトルに変換する．

BoW を用いたベクトル変換の例を図 5 に示す．図 5 中の Word_2 はメタワードであり，変数 x および y がこれに該当する．

doc2vec　doc2vec は，ニューラルネットワークによる教師なし機械学習に基づくベクトル化方式であり，自然文書を対象とした機械学習において有効性が実証されている [8]．本検索手法では，doc2vec を用いてコードブロックをベクトル化するにあたって，ライブラリは gensim[*2]を利用している．doc2vec を用いたベクトル変換の例を図 6 に示す．

[*2] https://radimrehurek.com/gensim/

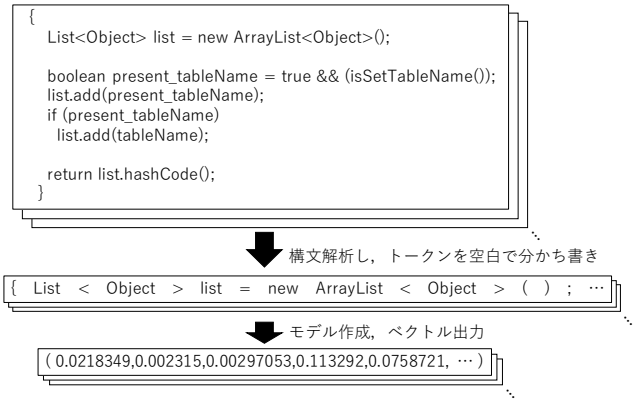

図 6 doc2vec をコード片に適用した例

まず，学習データ用コードブロックを ANTLR[*3]を用いてトークン単位に分割する．次に，コードブロックのトークンを 1 行にスペースで分かち書きをする．そのような処理を行った文字列を学習データとして与えると，機械学習によってトークンの分散表現ベクトル情報を持ったモデルを作成する．そのモデルにコードブロックのトークン列を入力として与えると，そのコードブロックの分散表現ベクトルを得ることができる．

3.3 類似コードブロックの検索

この節では，類似コードブロックの検索を行う STEP B について説明する．STEP B では，STEP A で作成したモデルを使用し，検索を実行する．STEP B は 3 つの手順から構成されている．STEP B の概要を図 7 に示す．

STEP B-1

検索したいコード片を構文解析し，STEP A-3 と同様，3.2.1 節で説明した手法で特徴ベクトルに変換する．

STEP B-2

STEP A-5 で作成したモデルへ，STEP B-1 で作成した特徴ベクトルを入力として与えると，その特徴ベクトルがどの類似コードブロックセットに属するかの確率を示すベクトルが出力されるので，それを基にラベル $x(0 \leq x \leq I)$ を算出する．

STEP B-3

算出したラベル x に対応する類似コードブロックセット S_x に属するコードブロック ${b_x}^{(j)}(1 \leq j \leq J)$ を類似コードブロックとして出力する．ただし，$x = 0$ の場合は，該当コードブロック無しとする．

4. 評価実験

本章では，本研究で提案した類似コードブロック検索手法の評価実験について述べる．評価実験では，3 つの OSS から作成したベンチマークを使用し，類似コードブロックの検索精度を，適合率，再現率，F 値の観点から評価した．本実験で対象とした OSS は HBase 2.0[*4]，OpenSSL 0.9.1〜1.1.0[*5]，FreeBSD 11.1.0[*6]である．OpenSSL はアップデート等により生じるバージョン間の類似コードブロックを評価に使用するため，複数のバージョンを使用している．

4.1 トレーニングパラメータ

本実験では，深層学習フレームワーク Chainer3.4.0[*7]を使用し，FFNN を実装した．層の数は，入力層，隠れ層，出力層の 3 層で，入力層の次元数は，入力する特徴ベクトルの次元数に合わせており，出力層の次元数は類似コードブロックセットの種類に合わせて実験を行った．隠れ層の次元数は経験的に定めており，徐々に増やしていったところ，100 から 200 次元ほどで，学習がうまく進むようになった．出力層にはソフトマックス関数と閾値 0.95 のステップ関数を使用し，その結果出力が 0 ベクトルになった場合は，特徴ベクトルにラベル 0 を割り当てる．

4.2 実験手順

検索精度の評価は以下の 3 つの手順で行った．

STEP 1

各 OSS に対して学習用データセットと評価用データセットを作成する．

STEP 2

学習用データセットを用いて FFNN の機械学習モデルを作成する．

STEP 3

作成したモデルに評価用データセットを入力として与え，適合率，再現率，F 値を算出する．

4.3 データセットの作成方法

本実験で使用した学習用データセットと評価用データセットは以下通り作成した．

STEP a

モデルの学習・評価に使用するためには多くの類似コードブロックが必要なので，各 OSS 内の類似コードブロックセットを抽出し，その中からコードブロックが 100 個以上存在するセットを選択する．

STEP b

学習に必要なデータを確保しつつ，機械学習モデルにとって未知のデータをできるだけ多くするため，選択したセット内の類似コードブロックの内 2 割に対してミューテーションを行い，類似コードブロックを新たに作成し，それらの特徴ベクトルを学習用データセッ

[*3] http://www.antlr.org/
[*4] https://hbase.apache.org/
[*5] https://www.openssl.org/
[*6] https://www.freebsd.org/
[*7] https://chainer.org/

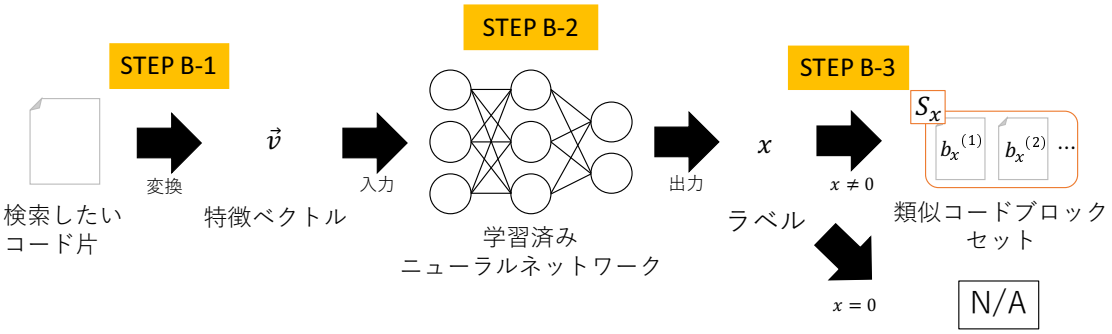

図 7　STEP B の概要図

トのポジティブデータとする．また，各セット毎にユニークなラベルを付与する．

STEP c

STEP b で作成したポジティブデータと構文的に類似していないコードブロックの特徴ベクトルを，学習データセットのネガティブデータとして 30000 個追加する．その際に使用したソースコードは，HBase に関しては BigCloneBench[9] から，OpenSSL に関しては FreeBSD から，FreeBSD に関しては OpenSSL から取得した．

STEP d

実験対象の OSS から抽出したコードブロックの特徴ベクトルを評価用データセットとする．評価用データセットに存在する，STEP a で抽出した類似コードブロックセットに属するコードブロックの特徴ベクトルには，STEP b で付与した各セット毎に対応するラベルを付与する．

各 OSS から作成したデータセットにおけるポジティブデータとネガティブデータの内訳を表 1 に示す．表 1 の評価用データセットにはかなりの偏りがあるが，これは，評価用データセットは実験対象の OSS に含まれているすべてのコードブロックであり，OpenSSL は複数バージョンを使用しているからである．

また，学習させるコードブロックと検索したいコード片の間に存在する構文的な差異の程度と検索精度の関係を調べるため，学習用データセットにおけるポジティブデータと評価用データセットにおけるポジティブデータの類似度を，OSS 毎に変更した．その内訳を表 2 に示す．Hbase からは構文的にほとんど一致しているコードブロックを，OpenSSL からは構文的に大部分が一致しているコードブロックを，FreeBSD からは構文的に一致している割合は低

表 1　データセット

	学習用データセット		評価用データセット	
OSS 名	ポジティブ	ネガティブ	ポジティブ	ネガティブ
Hbase	28,822	30,000	740	12,688
OpenSSL	36,772	30,000	281	99,719
FreeBSD	27,852	30,000	747	8,177

ブロック名	出力されるべきラベル	モデルが出力したラベル
ブロックA	0	0
ブロックB	0	1
ブロックC	1	1
ブロックD	2	2
ブロックE	3	3
ブロックF	4	3

$$precision = \frac{3}{5} = 0.6$$
$$recall = \frac{3}{4} = 0.75$$
$$F\_measure = \frac{2 \times \frac{3}{5} \times \frac{3}{4}}{\frac{3}{5} + \frac{3}{4}} = 0.67$$

図 8　検索精度指標の計算例

いが似たような動作を行うコードブロックを，ポジティブデータとして使用した．

4.4　検索精度の指標の定義

本実験では，適合率，再現率，F 値を用いて検索精度の評価を行った．これらの評価指標の説明を以下に示す．

適合率：　検索結果に対して本当に正しかった割合を指し，正確性に関する指標として用いられる．本実験では，評価用データセットに含まれているデータのうち，類似コードブロックを学習済みだとモデルが判定したデータ全体に対する，ポジティブデータの割合によって適合率を求める．

再現率：　正解に対して実際に検出された割合を指し，網羅性に関する指標として用いられる．本実験では，評価用データセットに含まれるポジティブデータ全体のうち，モデルへ入力すると正しい検索結果を返すデータの割合によって再現率を求める．

F 値：　適合率と再現率の総合的な評価として用いられ，適合率と再現率の調和平均によって求められる．

図 8 は検索精度指標の計算例を示す．この例では，ブロック B〜F の 5 つに対して，モデルは類似コードブロックを検索結果として出力しているが，そのうち類似コードブロックを正しく検索できているのはブロック C〜E の 3 つであるので，適合率は $\frac{3}{5}$ となる．また，入力として与

表 2　ポジティブデータ間の類似度

OSS 名	類似度
Hbase	0.9〜1.0
OpenSSL	0.7〜0.8
FreeBSD	0.1〜0.2

```
List<Object> list = new ArrayList<Object>();

boolean present_message = true && (isSetMessage());
list.add(present_message);
if (present_message)
    list.add(message);

return list.hashCode();
```

(a) 入力コード片

```
List<Object> list = new ArrayList<Object>();

boolean present_tableName = true && (isSetTableName());
list.add(present_tableName);
if (present_tableName)
    list.add(tableName);

return list.hashCode();
```

(b) 検索結果

図 9 構文一致コードブロックの検索成功例（HBase）

えたブロックのうち，実際に類似コードブロックをモデルが学習済みであるのは，ブロック C〜F の 4 つであるが，そのうちモデルが正しい検索結果を返したのは，ブロック C〜E の 3 つであるので，再現率は $\frac{3}{4}$ となる．F 値は，適合率と再現率の調和平均なので，$\frac{2}{3}$ となる．

4.5 実験結果

表 3 は評価実験から算出された適合率，再現率，F 値をに示す．この表でわかるように Hbase と OpenSSL から作成したデータセットの再現率は 1.000 であった．この結果から，コードブロックを事前に学習させると，構文的に大部分が一致しているコードブロックは，極めて高い確率で検索できることが分かった．

検索に成功したコードブロックの例を図 9，10 に示す．これらの図の上のコードブロックは学習データであり，4.2 節の STEP 2 で作成したモデルは，下のコードブロックから生成された特徴ベクトルを入力として与えられたときに，それが上のコードブロックと類似していると判定している．赤文字で示した部分は，コード片における互いに異なる部分である．

検索に失敗した例を図 11 に示す．これらは構文的に類似していないが，上下のコードブロックが類似しているとモデルは判定している．青文字で示した部分は，モデルがこれら 2 つのコード片が類似していると判定した原因だと考えられる文である．詳しくは 4.6 章で述べる．

4.6 考察

まず，提案手法の検索精度について考察を行う．

表 3 検索精度の評価

	BoW			Doc2Vec		
OSS 名	適合率	再現率	F 値	適合率	再現率	F 値
Hbase	0.924	1.000	0.960	0.830	1.000	0.907
OpenSSL	0.733	1.000	0.846	0.652	1.000	0.789
FreeBSD	0.497	0.822	0.620	0.519	0.529	0.524

```
{
int num,i;
char *p;
(中略)
for (;;)
        {
        (中略)
        if (num >= arg->count)
                {
                char **tmp_p;
                int tlen = arg->count + 20;
                tmp_p = (char **)OPENSSL_realloc(arg->data,
                        sizeof(char *)*tlen);
                if (tmp_p == NULL)
                        return 0;
                arg->data  = tmp_p;
                arg->count = tlen;
                for (i = num; i < arg->count;i++)
                        arg->data[i] = NULL;
                }
        arg->data[num++]=p;
        (中略)
        }
*argc=num;
*argv=arg->data;
return(1);
}
```

(a) 入力コード片

```
{
int num,len,i;
char *p;
(中略)
for (;;)
        {
        (中略)
        if (num >= arg->count)
                {
                arg->count+=20;
                arg->data=(char **)OPENSSL_realloc(arg->data,
                        sizeof(char *)*arg->count);
                if (argc == 0) return(0);
                }
        arg->data[num++]=p;
        (中略)
        }
*argc=num;
*argv=arg->data;
return(1);
}
```

(b) 検索結果

図 10 構文類似コードブロックの検索成功例（OpenSSL）

```
ArrayList<Long> timestamps = new ArrayList<>(filterArguments.size());

for (int i = 0; i<filterArguments.size();i++) {
    long timestamp =
        ParseFilter.convertByteArrayToLong(filterArguments.get(i));
   timestamps.add(timestamp);
  }
return new TimestampsFilter(timestamps);
```

(a) 入力コード片

```
List<Object> list = new ArrayList<Object>();

boolean present_tableName = true && (isSetTableName());
list.add(present_tableName);
if (present_tableName)
  list.add(tableName);

return list.hashCode();
```

(b) 検索結果

図 11 検索に失敗した例（HBase）

HBase，OpenSSL から作成したデータセットに対しては，再現率が 1.000 という結果になった．これは，構文の類似度が高い場合は，ほぼ確実に正しい検索結果を返すというこ

とを示している．その反面，FreeBSD は適合率が少し低いという結果になった．適合率が低い原因として，学習データと入力コード片の間に共通の文が存在する場合に，その 2 つのコードブロックが似ているとモデルが判定する確率が少し高くなる可能性が考えられる．例えば，List<Object> list = new Arraylist<Object>(); という文が含まれるコードブロックが OSS リポジトリに存在し，そのコードブロックを学習データにしてモデルを作成した場合，そのモデルに，List<Object> list = new Arraylist<Object>(); などの List 型のオブジェクトを作成する文を含むコードブロックを与えると，これら 2 つのコードブロックは，全体的には構文的に類似していなくても，類似しているとモデルが誤判定しやすくなる．本実験でも，図 11 の青文字で示した文が原因で，図 11 の 2 つのコード片は類似しているとモデルが判定した．FreeBSD から作成したデータセットは，HBase・OpenSSL から作成したデータセットに比べると検索精度が悪かった．このデータセットのコードブロックは，構文的に類似している割合が他のデータセットに比べて少なかったので，データセット間の類似コードブロックを検出するための本検索手法とは相性が悪かったと考えられる．

次に，Ichi Tracker と本検索手法の検索結果の違いについて述べる．図 9 のコードブロックは，構文的に一致しているので，Ichi Tracker でも検索することができる．しかし，図 10 のコードブロックは構文的に類似しているので，CCFinder の連続一致トークン数の閾値によっては，Ichi Tracker では検索することができない可能性がある．従って，本検索手法は，Ichi Tracker で検索できない可能性のあるコードブロックも，学習させることによって検索できることが確認できた．

4.7 ミューテーションの評価

本節では，ミューテーションを行うことによる検索結果への影響の評価実験について述べる．モデルは出力層で，どのラベルを割り当てることが適当かを表す確率を出力する．この評価実験では，モデルが特徴ベクトルを入力されたときに，正しいラベルを割り当てる確率について分析し，学習用データセットに使用するポジティブデータの数と，作成したモデルが類似コードブロックを入力として与えられたときに正しいラベルを割り当てる確率の関係を調べることで，ミューテーションの効果を評価している．

4.7.1 実験手順

ミューテーションの評価は以下の 4 ステップで行った．

STEP1 OpenSSL の過去 200 種類以上のバージョンに含まれ，バージョン間で類似コードブロック $f_n(n = 1, 2, \cdots, N)$ が存在する関数 f を選択し，ミューテーションを用いて類似コードブロックを大量に生成し，そこからコードブロックを抽出した後，OpenSSL のバージョン 1.0.0 以前のソースコードを学習させた doc2vec のモデルを用いて特徴ベクトルに変換し，ラベルとして 1 を付与する．

STEP2 STEP1 で作成した特徴ベクトルの中から a 個だけポジティブデータとして学習用データセットに加え，FreeBSD から抽出したコードブロックからランダムに 30000 個を STEP1 と同じ基準で特徴ベクトルに変換し，ネガティブデータとして学習データセットに加えた後，機械学習モデル M_a を作成する．a の値は，$a = 1, 100, 1000, 2000, 3000, 4000, 5000, 7000, 10000$ と変化させ，合計 9 種類のモデルを作成する．

STEP3 STEP1 で選択した関数の過去バージョンに存在する類似コードブロック $f_n(n = 1, 2, \cdots, N)$ を，STEP1 と同じ基準で特徴ベクトル $\vec{f_n}(n = 1, 2, \cdots, N)$ に変換し，作成した 9 種類のモデル $M_1, M_{100}, \cdots, M_{10000}$ それぞれに入力として与える．

STEP4 モデルが出力したベクトルから，STEP2 で学習させたポジティブデータ群と STEP3 で入力した特徴ベクトルが類似コードブロックセットに属している (ラベルが 1 である) とモデル M_n が判定する確率 $P_{M_a}(\vec{f_n}, 1)$ を求め，データ数 a と確率 $P_{M_a}(\vec{f_n}, 1)$ の関係を調べる．

4.7.2 実験結果

4.7.1 節に基づいた実験の結果を表 4 に示す．また，表 4 を折れ線グラフで表現したものを図 12 に示す．

表 4 中の $average_a$ と min_a は以下の式で与えられ，モデル M_a が特徴ベクトル $\vec{f_n}(n = 1, 2, \cdots, N)$ に対して算出する確率の平均値と最小値を表している．

$$average_a = \frac{1}{N} \sum_{n=1}^{N} P_{M_a}(\vec{f_n}) \quad (1)$$

$$min_a = \min_{n=1,2,\cdots,N} \{P_{M_a}(\vec{f_n})\} \quad (2)$$

学習データ数 a が大きいほど $average_a$ と min_a は増加した．

4.7.3 考察

元ソースコードに対してミューテーションを行って学習データを増やす手法は，ディープラーニングにおける入門

表 4 ポジティブデータ数と判定確率

学習データ数 a	$average_a$	min_a
1	0.000	0.000
100	0.000	0.000
1000	0.000	0.000
2000	0.973	0.173
3000	0.992	0.675
4000	0.989	0.731
5000	0.998	0.871
7000	0.999	0.955
10000	0.999	0.978

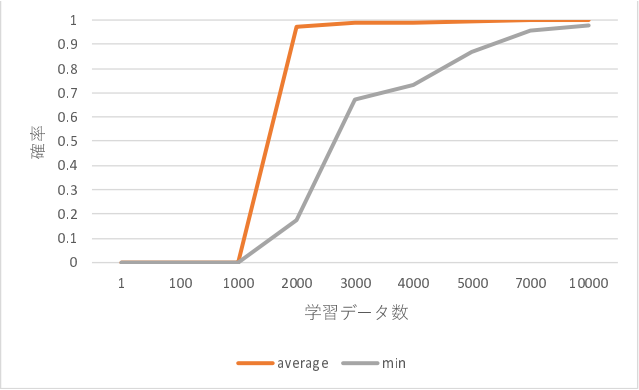

図 12 ポジティブデータ数と確率の関係

である，MNIST などの画像分類の問題から着想を得ている．画像分類の問題では，学習データを増やすために，元画像に対して拡大，縮小，移動などの操作を行い，元画像と少し異なる似た画像を新たに作成して学習データに加えていることがある．この発想をソースコードに適用し，元のソースコードから少し変更するために，ミューテーションを行った．

本節の実験においては，ポジティブデータの数を 2000 に設定すると，学習に使用したポジティブデータと，過去バージョンに存在する類似コードブロックのほとんどを，99%以上の確率で類似していると判定するモデルを作成することができ，ポジティブデータの数が 7000 程になると，過去バージョンに存在する全ての類似コードブロックに対して，90%以上の確率で類似していると判定するようになった．この実験から，ミューテーションを用いて学習データを増やすことは有効であると示すことができた．

5. 関連研究

コード検索とコードクローン検出は非常に近いテーマである．コード片を入力として与えるコード検索は，入力コード片のコードクローンが検索対象リポジトリ内に存在するかどうかを調べているといえるので，Ichi Tracker[1] のように，コード検索の際にコードクローン検出ツールを使用することも多い．神谷らが開発した CCFinder[2] は，トークン解析を行い，ユーザー定義名を特殊文字に変換した後，ソースコードをトークン列に変換し，閾値以上の長さで一致したトークン列をコードクローンとして検出している．

我々の研究グループでは，コンポーネントやファイル単位での再利用を特定する研究を行っている [10], [11], [12]．これら研究は，コンポーネントに含まれるクラスのシグネチャ（クラス名，メソッドのシグネチャ，フィールド名など）やコンポーネントに含まれるファイル間の Jaccard 係数，ファイル間の最長共通部分列を利用してコンポーネントやファイル単位の再利用を特定している．本研究は，FFNN に基づく機械学習を用いることで，コンポーネントやファイルよりも細粒度であるコード片レベルの検索を行っている．

White ら [13] は，RNN（再帰型ニューラルネットワーク）とオートエンコーダを使用して抽象構文木のベクトル化を行い，各構文木ベクトルの l2 ノルムを計測することによって，コードクローン検出を行う手法を提案している．Gu ら [14] は，機械翻訳にも使用されているディープラーニングのモデルである，RNN Encoder-Decoder モデルを使用し，自然言語をクエリとして与えることで API の使用順序例を生成する，“API Learning”を提案している．本研究は，これら研究と異なり類似コードブロックの検索を目的としている．また，本研究は RNN に比べて単純な構造のネットワークである FFNN を使用している．

6. まとめと今後の課題

本研究では，FFNN を用いることによる，類似コードブロックの検索を試みた．本検索手法では，ミューテーションで元ソースコードと構文的に類似したソースコードを作成した後，コードブロックを抽出する．そして，抽出したコードブロックを特徴ベクトルに変換し，各類似コードブロックセットに対応するラベルを付与し，教師あり学習を実行することで FFNN モデルを作成する．このモデルに，検索したいコード片の特徴ベクトルを入力することで，そのコード片と類似しているコードブロックのラベルがモデルから出力され，類似コードブロックを検索することができる．

評価実験では，3 つの OSS に対してコード検索を行い，構文的に差異があるコードブロックを非常に高い精度で検索できることが確認できた．

今後の課題として以下の点を挙げることができる．

- CCFinder を利用した Ichi Tracker と本手法を比較し，Ichi Tracker では検索できない類似コードブロックが本手法ではどの程度検索できるようになったのかを評価する必要がある．
- 適合率を高める必要がある．そのために，様々な構造の FFNN について実験し，ある特定の文の存在に検索結果が影響されすぎないような FFNN の構造を見つける必要がある．
- 本研究では，実装が容易であり，学習にかかる時間が比較的短いことから，FFNN を使用している．そのため，RNN 等の他のネットワークを使用して同様に実験を行いたいと考えている．
- 検索速度の評価を行う必要がある．本検索手法は事前準備に多大な時間を要するため，検索の速度が，学習にかかるコストに見合うものかどうかを評価する必要がある．
- 学習させる類似コードブロックセットの数をさらに増やし，より大規模なリポジトリを管理することができ

るかどうかを実験し，本検索手法の有用性を評価する必要がある．

- 検索精度の評価において，コードブロックの類似度だけでなく，各 OSS のコード片の特徴が検索精度に影響している可能性がある．そのため，同じ OSS 内に存在する様々な類似度をもつ類似コードブロックを学習させ，類似度と検索精度の関係をさらに正確に評価する必要がある．

謝辞 本研究は，JSPS 科研費 JP25220003，JP18H04094，JP16K16034 の助成を受けた．

参考文献

[1] Inoue, K., Sasaki, Y., Xia, P. and Manabe, Y.: Where does this code come from and where does it go?-integrated code history tracker for open source systems, *Proc. of ICSE 2012*, pp. 331–341 (2012).
[2] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670 (2002).
[3] Roy, C. K. and Cordy, J. R.: A mutation/injection-based automatic framework for evaluating code clone detection tools, *Proc. of ICSTW 2009*, pp. 157–166 (2009).
[4] Demuth, H., Beale, M. and Hagan, M.: *Neural network toolbox 6 User's Guide*, Mathworks (1994).
[5] Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M. and Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations, *IEEE Trans. Software Eng.*, Vol. 31, No. 3, pp. 213–225 (2005).
[6] Rush, D. and Bulsara, A.: Source Code Search Engine (2007). US Patent App. 11/663,417.
[7] 横井一輝，崔 恩瀞，吉田則裕，井上克郎：情報検索技術に基づくブロッククローン検出，電子情報通信学会技術研究報告，Vol. 117, No. 136, pp. 109–116 (2017).
[8] Lau, J. H. and Baldwin, T.: An empirical evaluation of doc2vec with practical insights into document embedding generation, *arXiv preprint arXiv:1607.05368* (2016).
[9] Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K. and Mia, M. M.: Towards a big data curated benchmark of inter-project code clones, *Proc. of ICSME 2014*, pp. 476–480 (2014).
[10] Ishio, T., Kula, R. G., Kanda, T., Germán, D. M. and Inoue, K.: Software ingredients: detection of third-party component reuse in Java software release, *Proc. of MSR 2016*, pp. 339–350 (2016).
[11] Ishio, T., Sakaguchi, Y., Ito, K. and Inoue, K.: Source file set search for clone-and-own reuse analysis, *Proc. of MSR 2017*, pp. 257–268 (2017).
[12] Kawamitsu, N., Ishio, T., Kanda, T., Kula, R. G., Roover, C. D. and Inoue, K.: Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity, *Proc. of SCAM 2014*, pp. 305–314 (2014).
[13] White, M., Tufano, M., Vendome, C. and Poshyvanyk, D.: Deep learning code fragments for code clone detection, *Proc. of ASE 2016*, pp. 87–98 (2016).
[14] Gu, X., Zhang, H., Zhang, D. and Kim, S.: Deep API learning, *Proc. of FSE 2016*, pp. 631–642 (2016).