特別研究報告

題目

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手法の提案

指導教員

井上 克郎 教授

報告者

藤原 裕士

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 29 年度 特別研究報告

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手法の提案

藤原 裕士

内容梗概

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．コード片を再利用することは，既存ソフトウェアを利用する方法の 1 つであり，信頼性やコストの面で優れている．そのため，新規ソフトウェアを開発する段階において，必要なコード片を特定することは重要である．しかし，既存ソフトウェアの数は増え，規模はますます大きくなっているため，手作業で既存ソフトウェアを管理し，その中から自分にとって必要なコード片を抽出することは困難である．この理由から，既存ソフトウェアの中から指定したコード片を自動で検索するための様々な手法が提案されている．

佐々木らはクラウド上のコード検索エンジンを用いてコード片をオープンソースソフトウェアの中から検索する手法を提案し，検出精度の観点からその有用性が確認されている．実際に公開されているコード検索エンジンを使用するので，検索する段階での信頼性は十分であり，検索結果をコードクローン検出ツールである CCFinder を用いてフィルタリングするため，検索結果として不適切なコード片は出現しにくい．しかし，クラウドサービスを利用しているため，サービス提供の状況次第ではツールが使用できなくなる可能性がある．また，フィルタリングの際に，元のコード片と似てはいても，少し異なる部分が存在した場合，CCFinder のフィルタリングによって検索結果から削除される恐れがある．

そこで本研究では，ニューラルネットワークを用いて，学習させたソースコードと類似したコードブロックを検索する手法を提案する．ここではコードブロックを，関数と，関数内部の if，for，while 文等の中括弧で囲まれた部分と定義する．

本手法は，まずソースコードに対して構文解析を行い，様々な種類のコードクローンを生成した後，元ソースコード及びコードクローン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，クローンセット毎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ニューラルネットワークのモデルを作成する．

実際に検索を行う際は，作成したモデルへ，学習データと同様の方法でコードブロックから生成された特徴ベクトルを与える．すると，学習させたコードブロックと与えた特徴ベクトルに対応するコードブロックが類似している場合は，学習コードブロックが属するクロー

ンセットに対応するラベルが出力される．そのため，出力されたラベルを確認することで，入力として与えたコードブロックが類似しているクローンセットを検索することができる．

主な用語

コード検索
コードクローン
ニューラルネットワーク
BoW
doc2vec

目 次

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．コード片を再利用することは，既存ソフトウェアを利用する方法の 1 つであり，信頼性やコストの面で優れている．そのため，新規ソフトウェアを開発する段階において，必要なコード片を特定することは重要である．しかし，既存ソフトウェアの数は増え，規模はますます大きくなっているため，手作業で既存ソフトウェアを管理し，その中から自分にとって必要なコード片を抽出することは困難であるため，既存ソフトウェアの中から指定したコード片を自動で検索するための様々な手法が提案されている．

Ichi Tracker[1] は，佐々木らによって提案された，入力として与えたコード片をオープンソースソフトウェアの中から検索するためのツールである．佐々木らの手法では，コード片に含まれる識別子として利用されている単語の数を測定し，クラウド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，コードクローン検出ツールである CCFinder[2] を適用してフィルタリングを行うことによって，類似したコード片を検索する．この手法は，評価実験によって，検索精度の観点から有用性が確認されている．しかし，コード検索エンジンは Google Code Search などのクラウドサービスを利用しているため，サービス提供の状況次第ではツールが使用できなくなる可能性がある．また，CCFinder を用いて検索結果をフィルタリングするため，元のコード片に類似している有用なコード片でも，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．

そこで，本研究では，様々な種類のコードクローンから生成された特徴ベクトルを学習データとして機械学習を行うことで，学習させたソースコードと類似したコードブロックを検索する手法を提案する．ここではコードブロックを，関数と，関数内部の if，for，while 文等の中括弧で囲まれた部分と定義する．

本手法では，まずソースコードに対して構文解析を行い，様々な種類のコードクローンを生成した後，元ソースコード及びコードクローン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，クローンセット毎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ニューラルネットワークのモデルを作成する．

実際に検索を行う際は，作成したモデルへ，学習データと同様の方法でコードブロックから生成された特徴ベクトルを与える．すると，学習させたコードブロックと与えた特徴ベクトルに対応するコードブロックが類似している場合は，学習コードブロックが属するクローンセットに対応するラベルが出力される．そのため，出力されたラベルを確認することで，入力として与えたコードブロックが類似しているクローンセットを検索することができる．

評価実験では，Ichi Tracker でも用いられている BoW と，ディープラーニングを使用し

て文章をベクトル化する技術である doc2vec の 2 種類のベクトル化の手法を使用して類似コードブロックの検索精度を比較した．また，学習データに含めるコードクローンの数を変化させることで，学習データの量と検索精度の間にどのような関係が生じるかを調べた．

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．最後に，5 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，コードクローン，ミューテーション，文書のベクトル化，ニューラルネットワーク，コード検索の既存研究である佐々木らの Ichi Tracker とその問題点について述べる．

2.1　コードクローン

コードクローンとは，ソースコード中に含まれる，互いに一致または類似したコード片のことであり，一般的に，コードクローンの存在はソフトウェアの保守を困難にすると言われている [3]．コードクローンの主な発生要因は，既存のソースコードのコピーアンドペーストによる再利用であり，他の発生要因としては，定型処理による発生，コード自動生成ツールによる発生，偶然の一致による発生等も挙げられる [4]．また，互いにコードクローンになるコード片の対をクローンペアと呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合をクローンセットと呼ぶ．

Roy らはコードクローンの定義として，コードクローン間の違いの度合いに基づき以下の 4 つのタイプに分類している [5]．

タイプ 1

空白やタブの有無，コーディングスタイル，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 の違いに加えて，変数名などのユーザー定義名，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，変更などが行われているコードクローン．

タイプ 4

類似した処理を実行するが，構文上の実装が異なるコードクローン

タイプ 4 のコードクローンとして，以下のものが挙げられる．

- 条件分岐処理や繰り返し処理などの制御構造の実装が異なる．
- 中間媒介変数の利用の有無が存在している．
- 文の並び替えが発生している．

2.2 コードクローン検出

ソースコードの規模はますます大きくなっているため，ソースコード中に含まれるコードクローンの量は膨大なものとなり，手作業でそれらを管理することは困難である．この理由から，コードクローンを自動的に検出することを目的とした様々なコードクローン検出手法が提案されている [3, 6]．

既存のコードクローン検出技術は，コードクローンの検出単位によって，大まかに以下の 5 つに分類される [3]．

行単位の検出

ソースコードを行単位で比較することにより重複コードを特定する．

Johnson や Ducasse らは，各行の空白やタブを取り除いた後，すべての行を比較することで重複行を検出する手法を提案している [7, 8, 9]．この手法ではタイプ 1 のコードクローンのみ検出可能だが，処理が単純なため，高速であり，不特定多数のプログラミング言語に対して適用できるという利点がある．

字句単位の検出

ソースコードを字句の列に変換し，閾値以上の長さで連続して一致している字句の部分列を検出する．

字句単位検出の代表的なツールとして，神谷らが開発した CCFinder[2] がある．CCFinder 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の列に変換する．このとき，変数名や関数名などのユーザー定義名を特殊文字トークンに変換する．そして，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する．このツールはタイプ 2 までのコードクローンが検出可能である．

抽象構文木を用いた検出

ソースコードを構文解析することにより抽象構文木を作成し，同形の部分木を検出する．

抽象構文木を用いた検出手法の代表的なツールとして，Jiang らが開発した DECKARD[10] がある．DECKARD では，抽象構文木の各部分木を特徴ベクトルに変換する．そして，LSH（Locality-Sensitive Hashing）[11] を用い，特徴ベクトル間の類似度を求めることによってコードクローンの検出を行う．このツールはタイプ 3 までのコードクローンが検出可能である．

プログラム依存グラフを用いた検出

ソースコードを意味解析することにより，ソースコードの要素の依存関係を抽出し，要素をノード，依存関係を有向辺としたプログラム依存グラフを作成し，同形の部分グラフを検出する．

Komondoor らは，ソースコード中の文をプログラム依存グラフのノードとする検出手法を提案している [12]．この手法では，ソースコード中の文を種類ごとに分類し，同種の文の対に対して，フォワードプログラムスライスとバックワードプログラムスライスの両方を用いて，同一のグラフ構造が作成されるかを検査する．このとき，閾値以上の大きさの同一構造グラフが構築された場合，そのグラフに含まれる文をコードクローンとみなす．この手法ではタイプ 4 までのコードクローンが検出可能である．

メトリクスなど，その他の技術を用いた検出

プログラムをファイル，クラス，メソッドなど，何らかのモジュール単位に分割し，各モジュールに対してメトリクスを計算し，その値の一致または近似をもって，モジュール単位のコードクローンを検出する．

横井らは，コードブロックをモジュール単位とし，モジュールの特徴ベクトルから類似度を計測することでコードクローンを検出する手法を提案している [13]．この手法では，ソースコードに対して構文解析を行い，ソースコードからコードブロックを抽出し，情報検索技術の 1 つである TF-IDF 法 [14] を用いてコードブロックを特徴ベクトルに変換し，特徴ベクトル間のコサイン類似度が閾値以上となったコードブロックの組をコードクローンとして検出する．この手法ではタイプ 4 までのコードクローンが検出可能である．

2.3 ミューテーション

ミューテーションとは，機械的にソースコードを変更することによって，ソースコードに不具合を埋め込むことであり，ソースコードのテストケースを評価するためなどに使用される [15]．Roy らは，ミューテーションを用いることでコードクローン検出ツールを評価するフレームワークを作成し，13 のミューテーションオペレータを定義している [16]．

mCW

空白の数を変更することで，タイプ 1 のコードクローンを生成する．

mCC

コメントを変更することで，タイプ 1 のコードクローンを生成する．

mCF

コーディングスタイルを変更することで，タイプ 1 のコードクローンを生成する．

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ことで，タイプ 2 のコードクローンを生成する．

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ことで，タイプ 2 のコードクローンを生成する．

mRPE

変数単体の式を別の式に置き換えることで，タイプ 2 のコードクローンを生成する．

mSIL

ある文にわずかな挿入を行うことで，タイプ 3 のコードクローンを生成する．

mSDL

ある文の一部を削除することで，タイプ 3 のコードクローンを生成する．

mILs

いくつかの文を挿入することで，タイプ 3 のコードクローンを生成する．

mDLs

いくつかの文を削除することで，タイプ 3 のコードクローンを生成する．

mMLs

いくつかの文を修正することで，タイプ 3 のコードクローンを生成する．

mRDS

いくつかの宣言文を並べ替えることで，タイプ 4 のコードクローンを生成する．

mROS

いくつかの宣言文以外の文を並べ替えることで，タイプ 4 のコードクローンを生成する．

mCR

制御構造を別のものに置き換えることで，タイプ 4 のコードクローンを生成する．

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図 1 に示す．9 行目の一部を削除することで，タイプ 3 のコードクローンを生成している．

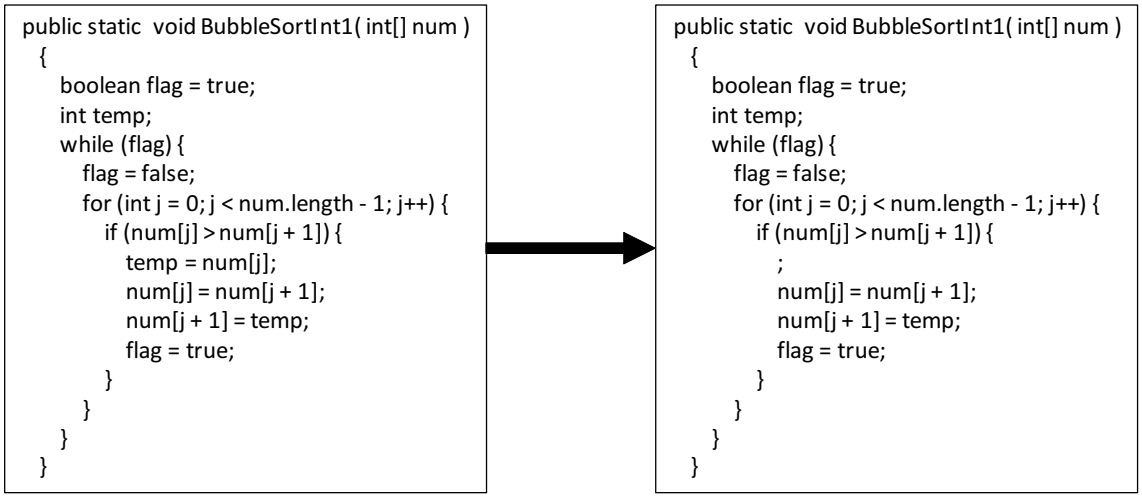


図 1: ミューテーションの例 (mSDL)

2.4 文書のベクトル化

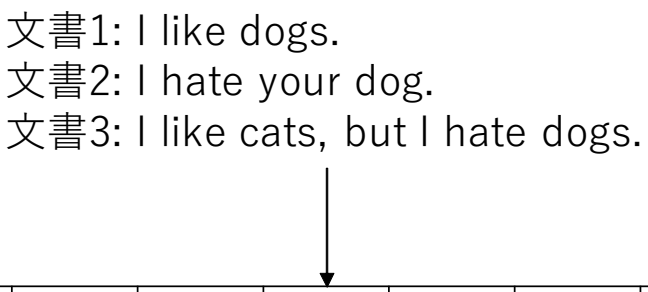
自然言語処理の分野において，文書の特徴を調べたり，複数の文書の類似度を求めたりする際に，文書を何らかの方法によってベクトルで表現することがある．

2.4.1 BoW

BoW(Bag of Words) とは，文書中の各単語の数を数えることで複数の文書を比較する手法である．比較したい文書をすべて単語に分割し，各文書中にどの単語がいくつ存在するかを数え，各単語の出現回数を特徴ベクトルの各要素とすることで文書をベクトルに変換する．よって，特徴ベクトルの次元数は，全文書に存在する全単語の種類数となる．BoW の例を図 2 に示す．

2.4.2 doc2vec

doc2vec とは，Le と Mikolov によって 2014 年に提案された，文書の分散表現をディープラーニングを用いて学習する手法である [17]．学習データとなる文書を与えると，まず，その文書に存在する各単語の分散表現をディープラーニングを用いて学習し，その学習結果をもとに，文書の分散表現を計算し，ベクトルに変換する．出力されるベクトルの次元数は固定長であり指定可能である．一般に，次元数を多くすればベクトルの表現能力が高くなるが，計算時間が増える．



	I	like	dogs	hate	your	dog	cats	but
文書1	1	1	1	0	0	0	0	0
文書2	1	0	0	1	1	1	0	0
文書3	2	1	1	1	0	0	1	1

図 2: BoW

2.5 ニューラルネットワーク

ニューラルネットワークとは，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 [18]．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り，重みの値を調整することによって，ニューラルネットワークを特定の機能を実行するように訓練することができる．ニューラルネットワークは，特定の入力を与えると何らかの出力を返す．その出力が目標値に一致するようになるまで，出力値と目標値を比較しながら，ネットワークの重みを調整することでニューラルネットワークを訓練する手法が一般的である．順伝播型ニューラルネットワークの模式図を図 3 に示す．順伝播型ニューラルネットワークは，歴史上最初に考案されたニューラルネットワークモデルである．

2.6 Ichi Tracker

この節では，Ichi Tracker[1] について説明する．

Ichi Tracker は，佐々木らによって提案された，オープンソースソフトウェアからコード片を再利用するため，入力として与えたコード片をオープンソースソフトウェアの中から検索するためのツールである．

2.6.1 Ichi Tracker のコード検索手法

Ichi Tracker のコード検索手法は以下の 8 つのステップに分けられる．図 4 は Ichi Tracker の概要を表している．

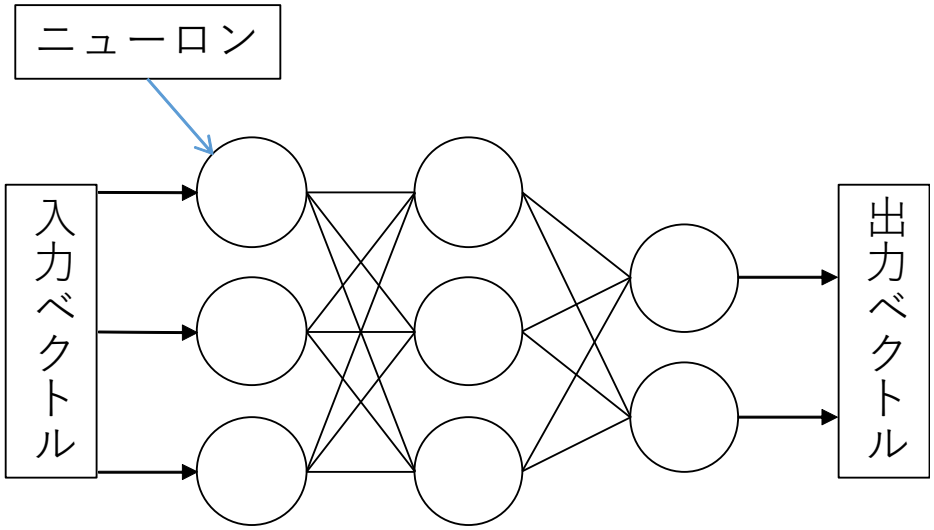


図 3: ニューラルネットワークの例

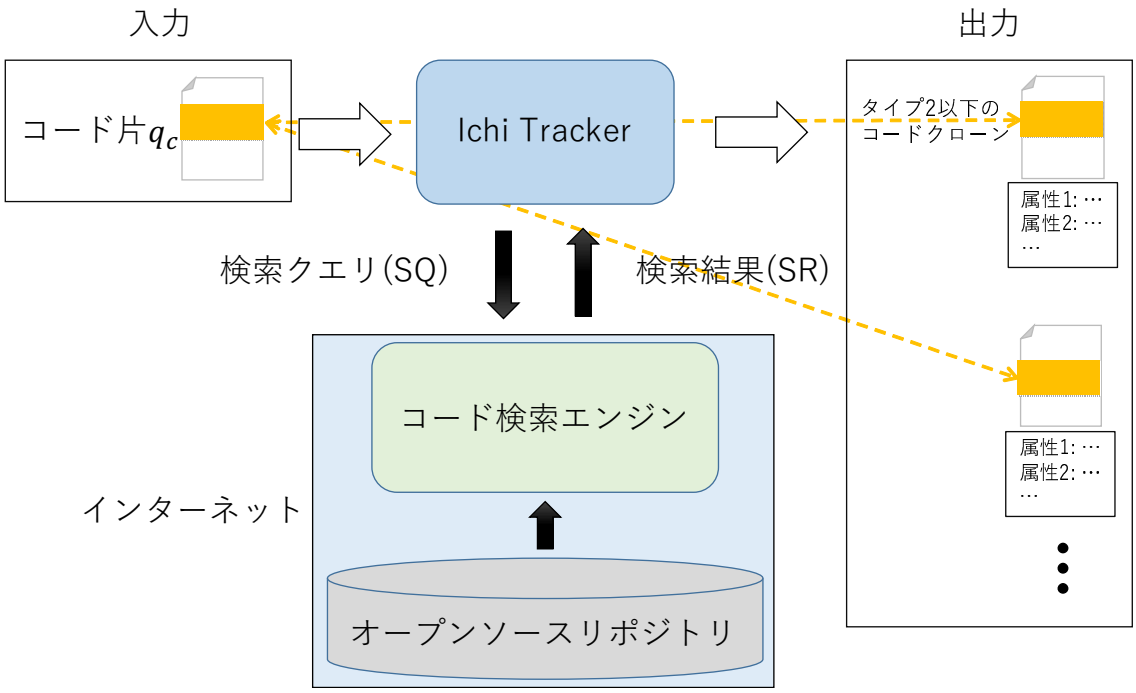


図 4: Ichi Tracker の概要

STEP1: ワードの抽出

入力されたコード片 q_c をトークンに分割し，ワードを抽出する．ここでワードとは変数や関数などにつけられた識別子名を構成する単語である．

STEP2: キーワードの選択

それぞれのワードが入力として与えられているコード片の中に登場する数を数え，5文字以上のワードをピックアップする．

STEP3: クエリ生成

STEP2 でピックアップしたキーワードから，登場した数の多い順に n 個のワードを選択し，コード検索エンジンに入力として渡すためのクエリ SQ を生成する．最初は $n = 1$ とする．コード検索エンジンとして，SPARS/R，Google Code Search，Koders を利用する．

STEP4: 結果の確認

コード検索エンジンから返される結果は，ヘッダーリスト SR である．もしその長さが m_{max} より長ければ，$n \leftarrow n + 1$ として STEP3 に戻る．Ichi Tracker では $m_{max} = 50$ と設定している．

STEP5: 検索結果の分析

得られたヘッダーリストを用いて，ファイルをダウンロードする．ダウンロードしたファイルには html の情報など，ソースコード以外の文が含まれているので，純粋なソースコード部分のみを抽出する．

STEP6: コードクローン検出ツールを用いた検索結果のフィルタリング

入力として与えたソースコード q_c と，STEP5 で得られたソースコード $sr_1, sr_2, \ldots$ のそれぞれがコードクローンかどうかを CCFinder[2] を用いて確認する．もし q_c と sr_i がコードクローンでなかった場合は，検索結果から sr_i を削除する．

STEP7: 結果の出力

STEP6 を通して残ったソースコードと，その属性を組み合わせて，システムの出力とする．

2.6.2 Ichi Tracker の問題点

Ichi Tracker に対して，以下の 2 つの問題点があげられる．

1 つ目は，コード検索エンジンにインターネット上のクラウドサービスを利用している点である．このため，インターネット環境がない状況下や，コード検索エンジンのサービスが

終了してしまった場合には，Ichi Tracker を使用することができない．

2 つ目は，タイプ 3・4 のコードクローンを検索できないという点である．Ichi Tracker では，フィルタリングの際に CCFinder を利用しているが，CCFinder はタイプ 2 までのコードクローンしか検出できないので，タイプ 3・4 のコードクローンを検索結果から削除してしまう．そのため，入力コード片に存在するバグ等が原因となって，検索結果の出力数が大幅に減少する可能性がある．

そこで本研究では，ニューラルネットワークを用いた機械学習モデルにより，オフラインの環境でもコード検索を実行でき，また，完全一致するコード片だけでなく，少し変更が加わったタイプ 3・4 のコードクローンも検索できる手法を提案した．

3　提案手法

本研究では，ニューラルネットワークを使用し，コードブロック単位で類似コード片を検索する手法を提案する．本手法は，ニューラルネットワークの学習を行う STEP A と，コードブロックの検索を行う STEP B の 2 段階に分かれている．なお，本手法では，順伝播型ニューラルネットワークを使用しており，以下の記述において，ニューラルネットワークとは，順伝播型ニューラルネットワークのことを指す．

まず，STEP A について説明する．STEP A では，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，それを用いてニューラルネットワークの機械学習モデルを作成する．STEP A は 5 つの手順から構成されている．STEP A の概要を図 5 に示す．

STEP A-1: コードブロック抽出とクローンセット構成

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，CCFinder や横井らのブロッククローン検出ツールを適用することで，クローンセット $S_i(1 \leq i \leq I)$ を構成する．クローンセット S_i には互いにコードクローンの関係にあるコードブロック ${b_i}^{(j)}(1 \leq j \leq J)$ が属する．また，ネガティブデータ (3.1.5 節) 生成用のソースコードを大量に用意し，コードブロック ${b_0}^{(l)}(1 \leq l \leq L)$ を抽出する．

STEP A-2: 学習データ用クローンブロックの作成

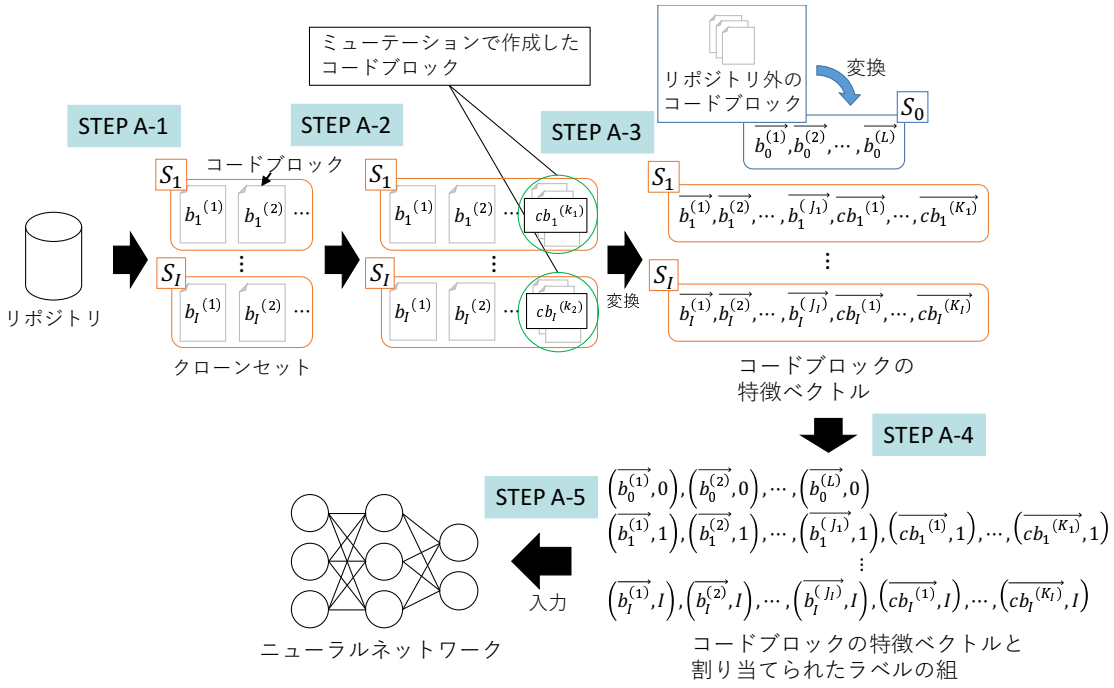


図 5: STEP A の概要図

クローンセット S_i に属するコードブロック ${b_i}^{(j)}$ に対するタイプ 2〜3 のクローンブロック ${cb_i}^{(k)}(1 \leq k \leq K)$ を，ミューテーションツールを用いて作成する．このとき，${cb_i}^{(k)}(1 \leq k \leq K)$ は，クローンセット S_i に属する．

STEP A-3: 学習データ用コードブロックのベクトル化

STEP A-1,2 で抽出，作成した各コードブロック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 3.3 節で説明する．

STEP A-4: ラベル割り当て

クローンセット 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 を割り当てる．また，コードブロック集合 ${b_0}^{(l)}$ はクローンセットではないが，データベース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 を割り当てる．

STEP A-5: 機械学習モデルの作成

特徴ベクトルと，割り当てたラベルを用いて，教師あり学習を行い，ニューラルネットワークの機械学習モデルを作成する．

次に，STEP B について説明する．STEP B では，STEP A で作成したニューラルネットワークを使用し，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 6 に示す．

STEP B-1: 検索コード片のベクトル化

検索したいコード片を構文解析し，STEP A-3 で用いたものと同じ手法で特徴ベクトルに変換する．

STEP B-2: ラベルの算出

STEP A-5 で作成した機械学習モデルへ，STEP B-1 で作成した特徴ベクトルを入力

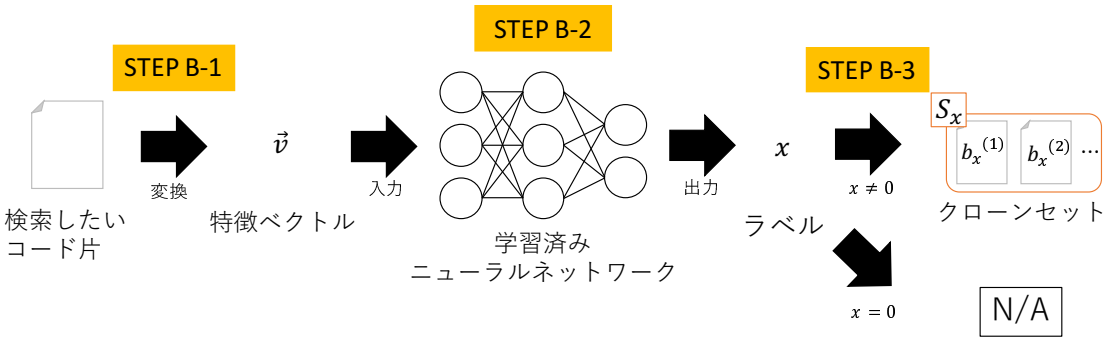


図 6: STEP B の概要図

として与えると，その特徴ベクトルがどのクローンセットに属するかの確率を示すベクトルが出力され，それを基にラベル $x(0 \leq x \leq I)$ を算出する．

STEP B-3: 類似コードブロックの出力

STEP B-2 で算出したラベル x に対応するクローンセット 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．

Ichi Tracker では検索とフィルタリングを分割して行っていたが，本手法では，検索とフィルタリングにあたる作業はニューラルネットワークの機械学習モデルを用いて行う．そのため，外部のコード検索エンジンに頼ることなく，ソースコードを管理することができる．また，Ichi Tracker ではフィルタリングの際に CCFinder を用いているので，タイプ 3 のコードクローンに該当する類似コード片を検索結果から削除してしまうが，本手法では，STEP A-2 でタイプ 2 や 3 のコードクローンを作成して学習データに含めているため，タイプ 3 のコードクローンに該当する類似コード片を検索することができる．

以降の節では，本手法で用いる用語の定義，コードブロックの抽出，特徴ベクトルの計算について説明する．なお，用語の定義，コードブロックの抽出については，横井らのブロッククローン検出ツールに準ずる．

3.1 用語の定義

3.1.1 コードブロック

プログラミング言語において，複数の命令文を一括りにまとめたものをコードブロックという．多くのプログラミング言語では，コードブロックは変数のスコープとしての意味を持つことがある．

本手法では，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

また，コードブロック単位のコードクローンを，クローンブロックと定義する．

3.1.2 ワード

本手法では以下の条件 3，4 のいずれかを満たす文字列をワードとして定義する．

条件 3 予約語

条件 4 識別子名を構成する単語

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

また，2 文字以下の識別子は，それらをまとめて同一のメタワードとして扱う．これにより，例えば，繰り返し文等でよく利用される i や j といった変数は，意味情報が込められていない変数として扱うことができる．さらに，条件分岐に用いられる if や while，繰り返しに用いられる for や while 等の予約語もワードとして扱う．なお，各ワードの大文字と小文字による区別はつけず，同一のワードとして扱う．

3.1.3 ニューラルネットワーク

現在までに様々な種類のニューラルネットワークが考案されてきたが，本手法におけるニューラルネットワークは，順伝播型ニューラルネットワークを指す．

3.1.4 ポジティブデータ

本手法では，条件 5 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 5 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，そのコードクローン

3.1.5 ネガティブデータ

本手法では，条件 6 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ネガティブデータとして定義する．

条件 6 リポジトリ内には存在せず，ポジティブデータすべてとコードクローンの関係にないため，特徴ベクトルにラベル 0 が付与されるコードブロック

3.2 コードブロックの抽出

本手法では構文解析を行い，コードブロックの抽出を行う．コードブロック抽出の手法は以下の 6 つのステップに分けられる．本手法では，構文解析に ANTLR（ver. 4.5.3）[1]を利用している．

[1] http://www.antlr.org/

STEP1 ソースコードに対して構文解析を行い，抽象構文木を生成する．

STEP2 抽象構文木から関数（3.1.1 章の条件 1 を満たすコードブロック）の部分木を取り出す．

STEP3 STEP2 で取り出した部分木を最も外側のコードブロックとして抽出する．

STEP4 STEP3 で取り出した部分木から，コードブロック（3.1.1 章の条件 2 を満たすコードブロック）の部分木を取り出す．

STEP5 STEP4 で取り出した部分木を入れ子関係にあるコードブロックとして抽出する．

STEP6 以降，深さ優先探索で抽象構文木からコードブロックを抽出する．

3.3 特徴ベクトルの計算

この節では，自然言語処理の分野において，文書をベクトル化する手法である BoW，doc2vec をどのようにして，自然言語ではない文書であるソースコードに対して適用したかについて説明する．

3.3.1 BoW

BoW を用いた特徴ベクトルの計算では，学習データ用コードブロックに現れる各ワードを，自然言語処理における単語とみなし，その数を数え，その値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．

また，検索したい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ワードが，そのコード片にいくつずつ現れているかを数え，その値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 7 に示す．図 7 中の Word_2 はメタワードであり，図 7 の例では，変数 j のことである．

3.3.2 doc2vec

本手法では，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．まず，学習データ用コードブロックを ANTLR を用いてトークン単位に分割する．そして，コードブロックを自然言語処理における文書，各トークンを単語とみなして学習させるため，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，ディープラーニングに

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

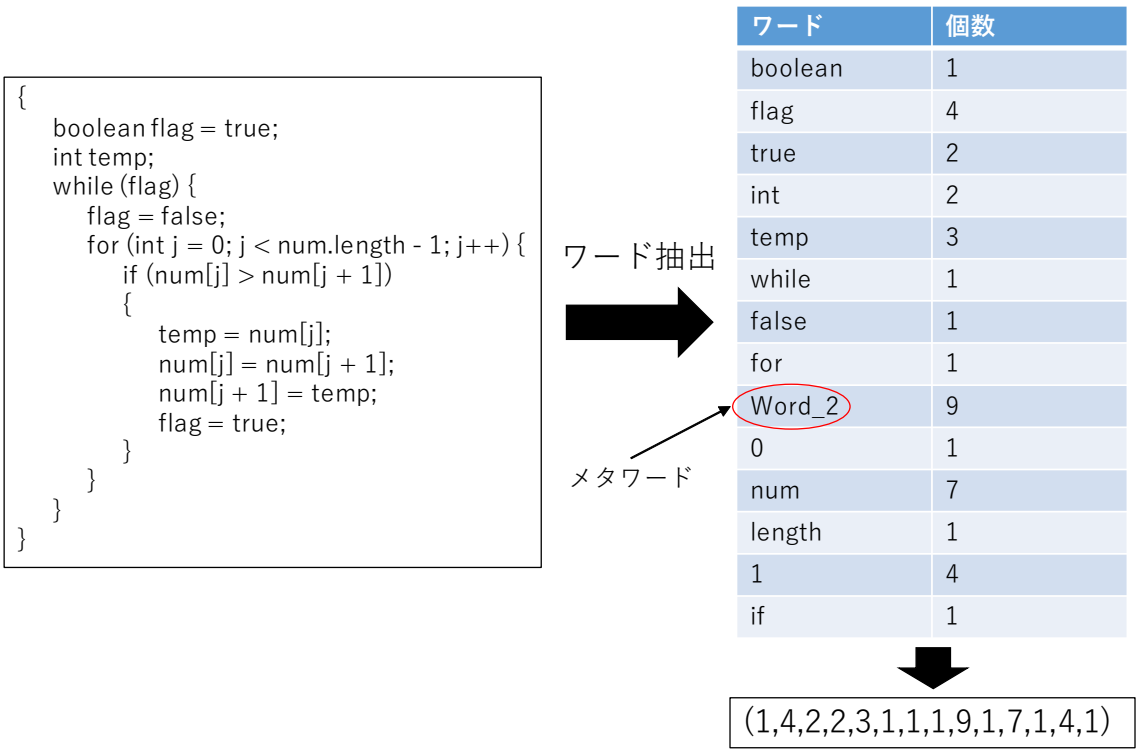


図 7: BoW をソースコードに適用した例

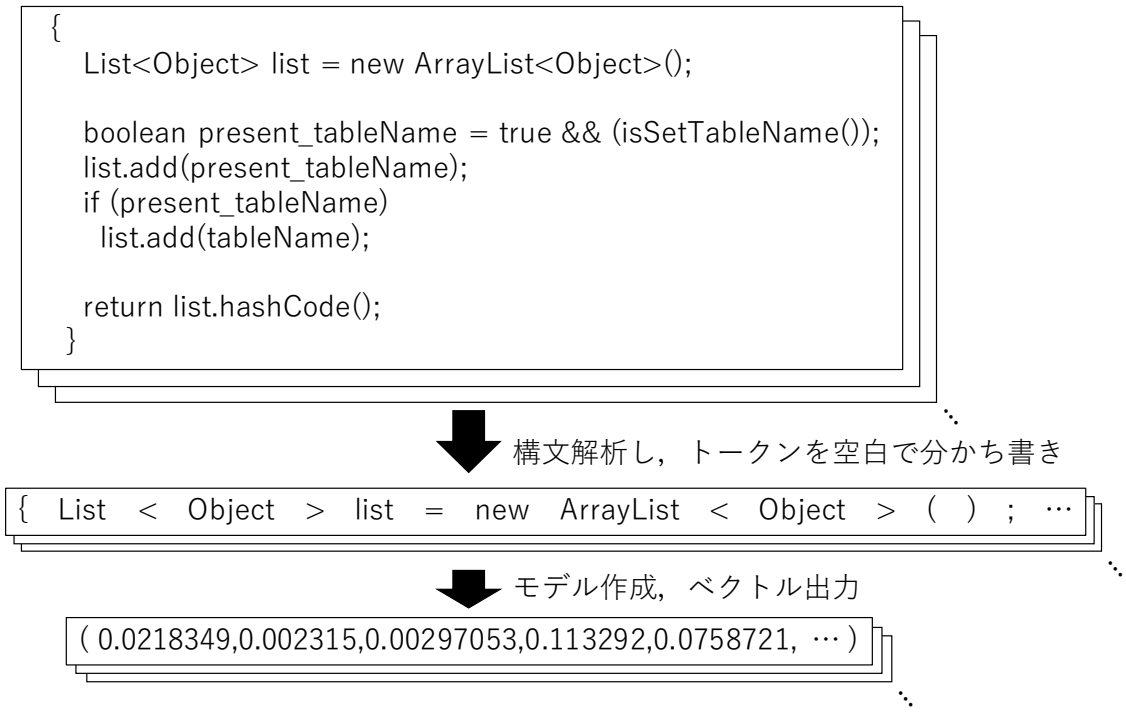


図 8: doc2vec をソースコードに適用した例

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生成する．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．ベクトルの次元数は一般的に 50〜300 あれば十分と言われており，今回は 300 次元に設定している．

doc2vec を用いたベクトル変換の例を図 8 に示す．

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，作成したベンチマークに対する検索精度の評価と，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を行った．

以降 4.1 節で，検索精度の評価実験について述べる．また，4.2 節で，ミューテーションの評価実験について述べる．

4.1　検索精度の評価

本節では，検索精度の評価実験について述べる．検索精度の評価実験では，対象プロジェクト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．本実験で検出対象としたプロジェクトは HBase[3]，OpenSSL[4]，FreeBSD[5]の 3 つである．

4.1.1　実験手順

検索精度の評価は以下の 3 ステップで行った．実験の概要を図 9 に示す．

1. 各プロジェクト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．
2. 学習用データセットを用いてニューラルネットワークの機械学習モデルを作成．
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．

4.1.2　データセットの作成方法

この節では，検索精度の評価実験におけるデータセットの作成手順をプロジェクトごとに説明する．

1. HBase
   (a) hbase.java に含まれており，かつプロジェクト内の他のソースコードにも多く使用されているメソッドを 7 つ選ぶ．これらはメソッド名が同一であり，タイプ 2 クローンであることを CCFinder を使用して確認した．

[3]https://hbase.apache.org/
[4]https://www.openssl.org/
[5]https://www.freebsd.org/

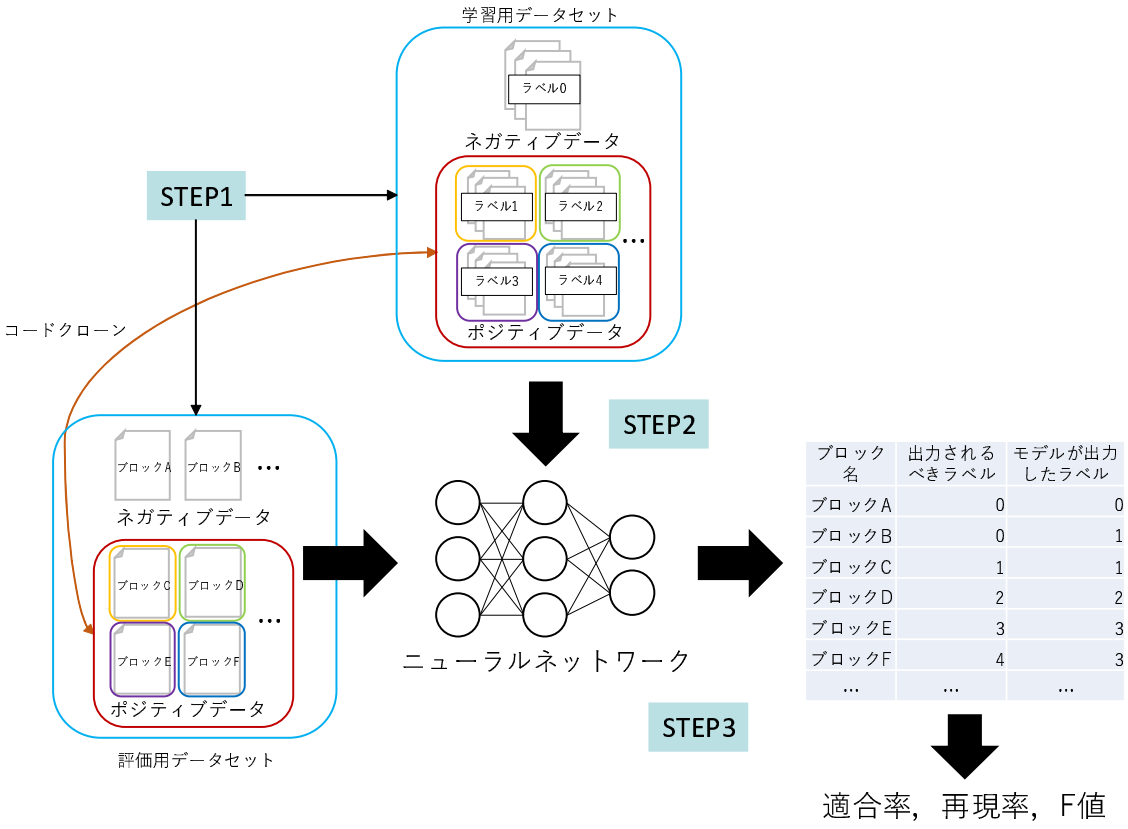


図 9: 計算精度検証実験の概要

(b) これらのメソッドをミューテーションによりコードクローンを生成してコードブロックを抽出した後，特徴ベクトルに変換し，ポジティブデータとして学習用データセットに加える．

(c) BigCloneBench[19] からランダムにコードブロックを 30000 個抽出し，7 つのメソッドとコードクローンの関係にないことを確認し，ネガティブデータとして学習用データセットに加える．

(d) 評価用データセットには，HBase プロジェクト内に存在する，hbase.java 以外のソースコードすべてから抽出したコードブロックを基にして生成した特徴ベクトルを加える．

2. OpenSSL

(a) Github から過去バージョンのプロジェクトをすべてダウンロードする．多くのバージョンに含まれている同一名称関数はコードクローンであり，タイプ 3 である．

(b) 200 種類以上の過去バージョンに含まれている同一名称関数の中から，目視で実際にタイプ 3 のコードクローンになっていると確認できた関数を 3 つ選択する．

(c) 選択した関数の中で，バージョン 1.0.0 以前に含まれているものをミューテートし，コードクローンを生成してコードブロックを抽出した後，特徴ベクトルに変換し，ポジティブデータとして学習用データセットに加える．

(d) FreeBSD からランダムにコードブロックを 30000 個選んで，3 つの関数とコードクローンの関係にないことを確認し，ネガティブデータとして学習用データセットに加える．

(e) 評価用データセットには，バージョン 1.0.0 以降のソースコードすべてから抽出したコードブロックを基にして生成した特徴ベクトルを加える．

3. FreeBSD

(a) オプションを処理するために getopt を用いている関数をランダムに 100 個選ぶ．これらの関数はオプション処理をするという共通点があるので，弱いタイプ 4 のコードクローンと言える..

(b) 選んだ関数をミューテートしてコードクローンを生成し，コードブロックを抽出した後，特徴ベクトルに変換し，ポジティブデータとして学習用データセットに加える．

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	
プロジェクト名	バージョン	ポジティブ	ネガティブ	ポジティブ	ネガティブ
Hbase	2.0	28,822	30,000	740	12,688
OpenSSL	0.9.1∼1.1.0	36,772	30,000	281	99,719
FreeBSD	11.1.0	27,852	30,000	747	8,177

(c) OpenSSL 内の getopt を使用していないコードブロック 30000 個をランダムに選び，特徴ベクトルに変換し，ネガティブデータとして学習用データセットに加える．

(d) 評価用データセットには，(a) で選ばなかった getopt 使用関数を含むソースコードから抽出したコードブロックと，FreeBSD のプロジェクトに含まれる getopt を用いていないソースコードから抽出したコードブロックを基にして生成した特徴ベクトルを加える．

各プロジェクト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．

4.1.3 検索精度の指標の定義

本実験では，検索精度の指標として，適合率，再現率，F 値の 3 つの指標を用いて評価を行った．3 つの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．

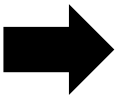
再現率

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．

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 10: 検索精度指標計算例

計算例を図 10 に示す．この例では，ブロック B〜F の 5 つに対して，モデルは何らかの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，図 10 の通りになる．

4.1.4 実験結果

4.1.1 節に基づいて行った実験から算出された適合率，再現率，F 値を表 2 に示す．

Hbase と OpenSSL から作成したデータセットにおいて，再現率が 1.000 になるという結果が出た．この結果から，コードブロックを事前に学習させると，そのコードブロックに対するタイプ 2 のコードクローンとタイプ 3 のコードクローンは，極めて高い確率で検索できることが分かった．

検索に成功した例を図 11，12 に示す．左側のコードブロックが学習データであり，モデルは右側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

表 2: 検索精度の評価

プロジェクト名	BoW			Doc2Vec		
	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

HBase - Hbase.java - hashCode

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

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

return list.hashCode();
```

検索クエリ

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

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

return list.hashCode();
```

図 11: 検索成功例 (タイプ 2 クローン)

OpenSSL- apps.c – chopup_args

```
{
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);
}
```

検索クエリ

```
{
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);
}
```

図 12: 検索成功例 (タイプ 3 クローン)

れが左側のコードブロックと類似していると判定している．図 11 はタイプ 2，図 12 はタイプ 3 のコードクローンなので，想定通り，与えられたコードブロックから，学習済みの類似コードブロックを検索することに成功している．

検索に失敗した例を図 13 に示す．これらはコードクローンではないが，右側と左側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．4.1.5 節でも述べるが，ArrayList 型のオブジェクトを生成する文の存在によってモデルが誤判定したと考えられる．他にも，入力として与えた際に，左側のソースコードを検索結果として返されたコードブロックが存在し，それらのコードブロックのほとんどに ArrayList 型のオブジェクトを生成する文が存在していることを確認した．

HBase - Hbase.java - hashCode	検索クエリ

List<Object> list = new ArrayList<Object>();
 	
boolean present_tableName = true && (isSetTableName());
list.add(present_tableName);
if (present_tableName)
  list.add(tableName);
 	
return list.hashCode();
	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);

図 13: 検索失敗例

4.1.5 考察

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，タイプ 3 までのコードクローンを学習させると，ほぼ確実に検索にヒットするということを示している．また，適合率が少し低いという結果になった．これは，検索の際に学習させたコードブロックとコードクローンの関係にないコードブロックに対しても，何らかの検索結果を返してしまう場合があることを示している．

適合率が低くなる原因として，学習データと検索クエリ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，という現象がある．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，そのモデルに検索クエリとして，`List<Object> list = new Arraylist<Object>();` など，List 型のオブジェクトを作成するような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，全体的にはコードクローンではなくても，似ているとモデルが判定しやすくなる．

適合率をさらに高める方法として，上記の現象による確率の上昇分を考慮してラベルを算出する方法が考えられる．また，学習させたコードブロックのクローン以外がモデルに入力された場合に，それらがラベル 0 であると判定される確率を高めることで適合率の上昇を狙うならば，リポジトリ外からネガティブデータとして学習に使用するコードブロックの数や種類を増やす方法が考えられる．

FreeBSD から作成したデータセットは，他の 2 種類に比べると精度が悪かった．このデータセットで学習させようとしたものが，タイプ 4 のコードクローンだったため，コードブロック単位で見ると類似した部分の割合が少なかったので，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手法とは相性が悪かったと考えられる．

4.2 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．この評価実験では，学習用データセットに加えるポジティブデータの数と，作成したモデルがそのポジティブデータのタイプ 3 クローンを入力として与えられたときに返す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．

4.2.1 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．実験の概要を図 14 に示す．

STEP1 OpenSSL の過去 200 種類以上のバージョンに含まれ，バージョン間でコードクローン $f_n(n = 1, 2, \cdots, N)$ が存在する関数 f を選択し，ミューテーションを用いてコードクローンを大量に生成し，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，ニューラルネットワークの機械学習モデル 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コードクローン $f_n(n = 1, 2, \cdots, N)$ 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同じクローン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．

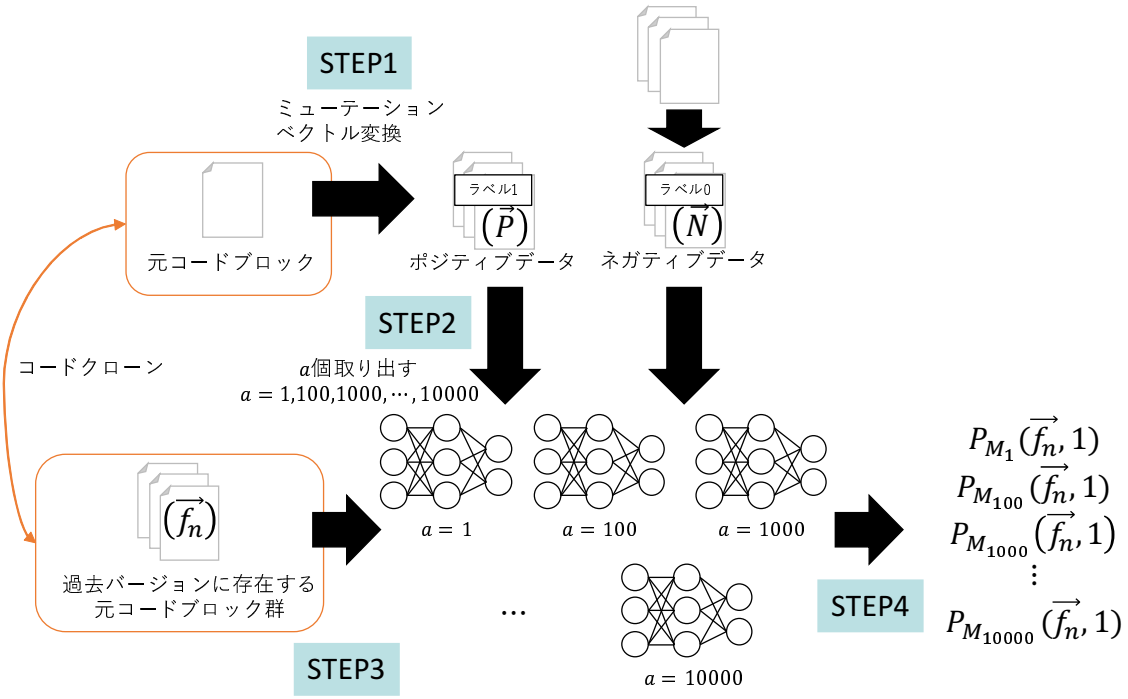


図 14: ミューテーション評価実験概要

4.2.2 実験結果

4.2.1 節に基づいた実験の結果を表 3 に示す．また，表 3 を折れ線グラフで表現したものを図 15 に示す．

表 3 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．

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

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

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

4.2.3 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である，MNIST[20] 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

表 3: 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

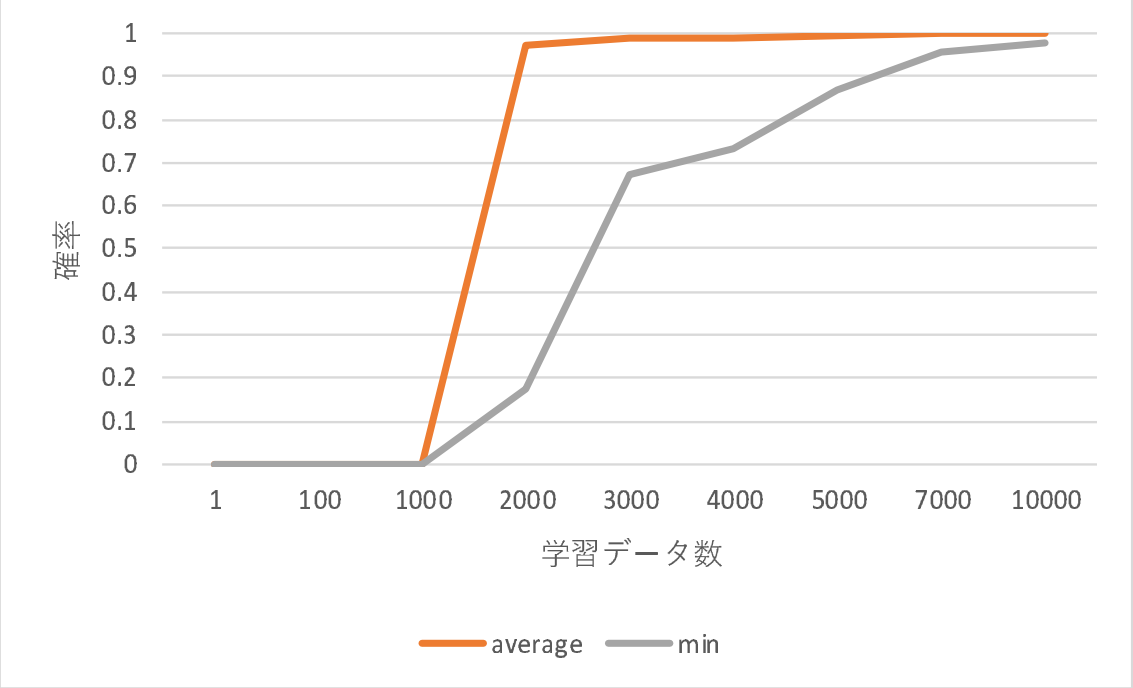


図 15: ポジティブデータ数と確率の関係

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そのポジティブデータとコードクローンの関係にあるコードブロックのほとんどを，99%以上の確率で似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コードクローンに対して，90%以上の確率で似ていると判定できるようになった．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　まとめと今後の課題

本研究では，ニューラルネットワークの機械学習モデルを用いて，類似コードブロックを検索する手法を提案した．本手法では，ソースコードに対して構文解析を行い，ミューテーションオペレータを適用してコードクローンを生成した後，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，各クローン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ニューラルネットワークの機械学習モデルを作成する．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 つのオープンソースソフトウェアに対し，ニューラルネットワークを用いた機械学習モデルによるコード検索を実際に行い，本手法の検索精度について調べた．その結果，Ichi Tracker では検索結果に出現しないタイプ 3 のコードクローンを非常に高い精度で検索できることがわかった．

また，学習に使用するポジティブデータの数を変更させたモデルを作成してその学習状況を観察した．その結果，学習データを増やすと，学習モデルが特徴ベクトルを正しく認識する確率が増加するという傾向を確認することができ，ミューテーションによる学習データ増加の有効性を示すことができた．

今後の課題として以下の点を挙げることができる．

- 本手法は現在，C 言語か Java で書かれたソースコードのみに対応している．そのため，他の言語にも対応させる必要がある．
- 検索速度の評価を行う必要がある．本手法は事前準備に多大な時間を要するため，実際に検索を行う際の処理速度がどの程度で，事前準備のコストに見合うものかどうかを評価する必要がある．
- 学習させるクローン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，本手法の有用性を評価する必要がある．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授には，研究の各段階において，多くのご指導を賜りました．井上 教授から多く賜った適切な御指導により，本論文を完成させることができました．井上 教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授には，多くの御助言を賜りました．特に，Clang の背景知識について教えてくださったことは，本研究に使用したミューテーションツールの実装に大きく作用しております．松下 准教授に心より深く感謝いたします．

名古屋大学 大学院情報学研究科附属組込みシステム研究センター 吉田 則裕 准教授には，研究に関する直接の御指導を賜りました．常に適切な御指導及び御助言を頂いたことにより，本論文を完成させることができました．吉田 准教授に心より深く感謝いたします．

奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学研究室 崔 恩瀞 助教には，研究や本論文の構成に関する多くの御助言を頂きました．崔 恩瀞 助教に心より深く感謝いたします．

大阪大学大学院情報科学研究科 神田 哲也 特任助教には，研究に関する貴重なご意見を賜りました．神田 哲也 特任助教に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 横井 一輝 氏には，ブロッククローン検出ツールを提供していただくなど，本研究の実装段階でご助力いただきました．心より深く感謝いたします．

最後に，私を支えてくださった，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井上研究室の皆様に，心より深く感謝いたします．

参考文献

[1] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go?-integrated code history tracker for open source systems. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 331–341. IEEE Press, 2012.

[2] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 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.

[3] 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.

[4] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pp. 368–377, 1998.

[5] 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.

[6] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[7] J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, Vol. 94, pp. 120–126, 1994.

[8] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pp. 109–118. IEEE, 1999.

[9] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software: Evolution and Process*, Vol. 18, No. 1, pp. 37–58, 2006.

[10] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105, 2007.

[11] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM symposium on Theory of computing*, pp. 604–613, 1998.

[12] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pp. 40–56. Springer, 2001.

[13] 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 (ソフトウェアサイエンス). 電子情報通信学会技術研究報告= IEICE technical report: 信学技報, Vol. 117, No. 136, pp. 109–116, 2017.

[14] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval: The concepts and technology behind search*. Addison-Wesley, 2011.

[15] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, Vol. 11, No. 4, pp. 34–41, 1978.

[16] Chanchal K Roy and James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pp. 157–166. IEEE, 2009.

[17] Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*, 2016.

[18] Howard Demuth, Mark Beale, and Martin Hagan. *Neural network toolbox*. Mathworks, 1994.

[19] 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 *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 476–480. IEEE, 2014.

[20] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, Vol. 29, No. 6, pp. 141–142, 2012.