情報処理学会論文誌 Vol.58 No.4 769–779 (Apr. 2017)

回帰結合ニューラルネットワークを利用したAPI推薦手法

山本 哲男[1,a)]

受付日 2016年8月8日, 採録日 2017年1月10日

概要：ソースコードを記述していく際，開発者は，効率良くプログラムを作成するために既存のソースコードの再利用やライブラリを活用して開発を行う．そこで，本研究では，既存のソースコードに記述されているメソッド呼び出し文の順序に着目し，メソッド呼び出し文を補完する手法について提案する．本手法では，回帰結合ニューラルネットワーク（recurrent neural network）を利用し，次に現れるであろうメソッド呼び出し文を予測する．さらに，提案する手法を実装し，10 プロジェクトのオープンソースソフトウェアを用いて補完候補の精度を計測した．また，回帰結合ニューラルネットワークの様々なパラメータが実験結果にどのように影響するかを調査し，補完候補の精度がどのように変化するかについても実験した．実験の結果，典型的なサンプルソースコードの補完においては，38%の精度で補完候補の 1 位に必要なメソッド呼び出し文が現れることが確認できた．

キーワード：コード推薦，リカレントニューラルネットワーク，深層学習

An API Suggestion Using Recurrent Neural Networks

Tetsuo Yamamoto[1,a)]

Received: August 8, 2016, Accepted: January 10, 2017

Abstract: Developers reuse existing source code or use libraries to develop effectively. In this study, we focus on the order of method invocation statements in existing source code and propose to suggest method invocation statements. This paper shows an approach to suggest method invocation statements using recurrent neural network. We have implemented the approach and conducted experiments to measure an accuracy with 10 open source software projects. We have investigated various prameters of recurrent neural network. Our evaluation has shown that our approach is 38% accuracy in API code suggestion, it can correctly suggest the API with top 1 candidate.

Keywords: code suggestion, recurrent neural netowork, deep learing

1. はじめに

近年，多くのライブラリやフレームワークが開発されており，開発者は効率良くプログラムを作成するために，それらのライブラリやフレームワークを活用して開発を行うことが頻繁にある．たとえば，Android アプリケーションにおいて，Android API を利用する割合は，多いアプリケーションで 42%にもなるという報告がある [17]．そこで，多くの統合開発環境では，作成中のソースコードに対してコード推薦の機能を提供し，調べる手間や入力の手間を省く機能が存在する．これらの機能を用いると，ソースコード中の変数やクラス名等に対して，それらのメソッド一覧を提示することができる．

図 1 は Andorid アプリにおける画面を作成する典型的なソースコードである．`View`，`TextView`，`Button` クラス等を利用し，`onClickListener` クラスを継承した匿名クラスをリスナーにセットすることでボタンの処理を記述している．しかし，API の順序やそもそも必要なクラス名やメソッド名が分からないと，自分で細かな処理を書く必要や既存のクラスを調べる時間が必要になり，手間がかかることになる．

そこで，すでに記述されたソースコードや開発者が記述

[1] 日本大学工学部情報工学科
College of Engineering, Nihon University, Koriyama, Fukushima 963–8642, Japan
[a)] tetsuo@cs.ce.nihon-u.ac.jp

```
1  public View onCreateView(LayoutInflater inflater,
2   ViewGroup container, Bundle savedInstanceState) {
3    View v=inflater.inflate(R.layout.fragment_dialog,
4                            container, false);
5    View tv=v.findViewById(R.id.text);
6    ((TextView)tv).setText("Dialog");
7    // Watch for button clicks.
8    Button button=(Button)v.findViewById(R.id.show);
9    button.setOnClickListener(new OnClickListener() {
10     public void onClick(View v) {
11       // When button is clicked, call up to owning
12       // activity.
13       ((FragmentDialog)getActivity()).showDialog();
14     }
15   });
16   return v;
17 }
```

図 1 Java ソースコード片
Fig. 1 A Java code snippet.

中のソースコードから情報を集めて解析し，適切な API やソースコードを推薦する仕組みに関する研究が多く存在する [1], [4], [13], [14], [20]．これらは，既存のソースコードの情報を利用することで，新規開発する開発者の求めているコード片や API を推薦する．[13] では API の呼び出し順を情報として利用し推薦する．我々は [20] で 2 つの API 呼び出し間の関係を情報として利用して，メソッド呼び出し文自体の推薦を行う手法を提案してきた．

本研究では，ニューラルネットワークを利用し，次に現れるであろうメソッド呼び出し文を予測し，候補を表示する手法を提案する．あるクラスのメソッド一覧が列挙される際に，適切と思われる順に並べ替えたメソッド一覧を開発者に提示する．

ニューラルネットワークには回帰結合ニューラルネットワーク（recurrent neural netowork）を利用し，既存のソースコードを学習させる．さらに，提案する手法を実装し，10 プロジェクトのオープンソースソフトウェアを用いて実験を行った．回帰結合ニューラルネットワークの様々なパラメータが実験結果にどのように影響するかを調査した．実験の結果，典型的なサンプルソースコードの補完においては，38%の精度で補完候補の 1 位に必要なメソッド呼び出し文が現れることが確認できた．

以降，2 章で提案手法で利用する言語モデルについて説明し，3 章で提案手法を説明する．4 章では実装したツールを用いて行った実験について述べる．さらに，5 章では，関連研究について触れ，最後に 6 章で本稿をまとめる．

2. 言語モデル

2.1 ニューラルネットワーク言語モデル

自然言語処理の分野において，単語が文章中に現れる過程を確率過程と見なし，ある単語がある位置に出現する確率がどの程度か計算し，単語の出現を予測することが行われている．このモデルは言語モデルと呼ばれる．

一般的に言語モデルは以下のように表せる．ある文書において，j 番目に出現する単語を w_j と表すと，1 番目から $j-1$ 番目まで連続し出現する単語の列は以下のように表す．

$$w_1^{j-1} = w_1, w_2, w_3, \ldots, w_{j-1}$$

なお，w_1^{j-1} の次に w_j が出現する条件付き確率を $P(w_j|w_1^{j-1})$ とする．いま，ある文書 s の単語の数が T とすると，その文書が生成される確率は以下ように表せる．

$$P(w_1^T) = \prod_{j=1}^{T} P(w_j|w_1^{j-1})$$

言語モデルにおいて，単語列の出現確率を予測するためにニューラルネットワークを用いた手法に Bengi らによるニューラルネットワーク言語モデル [3]（以下 NNLM）がある．

NNLM では，各単語 w_j を，出現したその単語の索引のみが 1 で残りの要素が 0 である N 次元のベクトルで表現する．N は全文書の語彙数を表し，このようなベクトル表現を 1-of-N 表現と呼ぶ．たとえば，文書が “I have a pen and a eraser.” だっとすると，語彙数 N は 6 になる．次に，各単語に通し番号を付ける．I を 1 番，have を 2 番，a を 3 番，pen を 4 番，and を 5 番，eraser を 6 番とする（番号自体に意味はなく，どの順番に番号をつけても構わない）．すると，I の 1-of-N 表現は $(1,0,0,0,0,0)^{\mathrm{T}}$ となり，have の 1-of-N 表現は $(0,1,0,0,0,0)^{\mathrm{T}}$ となる．

そして，単語列 w_{j-n+1}^{j-1} が与えられているときの，単語 w_j が出現する条件付き確率を出力するニューラルネットワークを学習する．ここで，n は学習する単語列の単語数である．たとえば，n を 3 とした場合，I という単語と have という単語の次には a という単語が出現するとして学習させる．次に，have と a の次に pen という単語が，a pen の次には and という風に，すべての組合せについて学習させていく．

2.2 回帰結合ニューラルネットワーク言語モデル

Bengi らの手法では，固定長 n の学習しかできない．そこで，Mikolov らは，回帰結合ニューラルネットワークを用いた言語モデル（以下 RNNLM）[11], [12] を提案している．回帰結合ニューラルネットワーク（以下 RNN）は通常のニューラルネットワークと異なり，時系列が考慮されており，系列データを処理するのに適している．また，系列の長さに制限はない．系列データは，テキストの場合は単語の並びと考えることができるため，単語の数が T とすると以下のように表せる．

$$w_1^T = w_1, w_2, w_3, \ldots, w_T$$

ここで，単語の並びをインデックス $t = 1, 2, 3, \ldots$ で表し，以降 t を時刻と呼ぶことにする．

図 2 に RNNLM における RNN のアーキテクチャを示す．図 2 に示すように，ニューラルネットワークの入力層を w，隠れ層を s，出力層を y とする．ある時刻 t におけるネットワークへの入力層を $w(t)$，出力層を $y(t)$，隠れ層（ネットワークの状態）を $s(t)$ で表す．$\mathbf{w}(t)$ は時刻 t の単語を 1-of-N 表現したベクトルである（N は語彙数）．隠れ層のユニット数を M とすると，行列 $\mathbf{U}$ は $N \times M$ 行列であり，単語を潜在空間へ写像する行列と考えることができる．$\mathbf{s}(t)$ は以下の式で表す．ここで，行列 $\mathbf{W}$ は時刻 $t-1$ の隠れ層からの重みとなる．

$$s(t) = f(\mathbf{U}w(t) + \mathbf{W}s(t-1))$$

$y(t)$ は以下のように表す．行列 $\mathbf{V}$ は隠れ層から出力層へつなぐための重みであり，$M \times N$ 行列となる．

$$y(t) = g(\mathbf{V}s(t))$$

ここで，$f(z)$ は以下の sigmoid 関数を表し，

$$f(z) = \frac{1}{1 + e^{-z}}$$

$g(z_m)$ は以下の softmax 関数を表している．

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$

RNN では学習する際，確率的勾配降下法が利用される．確率的勾配降下法としては，Backpropagation Through Time（BPTT）法 [19] を利用し計算されることが多い．BPTT 法では，各時刻の RNN を横に並べ，隠れ層の帰還路を，連続時刻での隠れ層のユニットの結合として表現する．結合したネットワークは，循環のない順伝播型ネットワークとなり，誤差逆伝播による勾配の計算が可能となる．計算量を減らすために，過去の履歴の数に制限をつけた Truncated BPTT を後に述べる本手法では採用する．さらに，長期依存を学習させるために，隠れ層に Long Short-Term Memory（LSTM）を利用する．また，隠れ層を 2 層，3 層と増やすネットワークも考えられる．

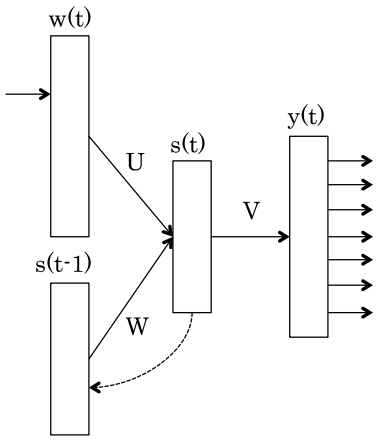

図 2 RNN のアーキテクチャ
Fig. 2 An architecture of RNN.

3. 手法

本章では，我々が提案するメソッド呼び出し文補完手法について説明する．本手法は学習フェーズと補完フェーズの 2 つのフェーズから構成されている．学習フェーズでは，補完の元となる情報をモデルに保存する．モデルは RNNLM を既存のソースコードのメソッド呼び出し文の並びに適用したものになる．具体的にはメソッド呼び出し文を単語と見なした文書を作成し，RNNLM で学習させる．補完フェーズでは，補完要求をするエディタ等の問合せに応答し，補完候補を出力する．エディタ等で現在入力中のソースコードを解析し，それらの情報からモデルを用いて補完候補を取得し，開発者に提示する．

以降，最初に全体の流れの説明をし，次に，学習フェーズで利用する手法，補完フェーズで利用する手法の順に説明する．なお，本手法で説明するソースコードは Java 言語で記述されているものとする．

3.1 ソースコード補完手法の概要

本手法を利用したソースコード補完の流れを**図 3** に示す．処理の流れは学習フェーズと，作成しているソースコードに対する補完フェーズに分けられる．本手法の最も重要な考えは，既存のソースコードの中のあるメソッド呼び出し文の並びから次に存在するであろうメソッド呼び出し文の割合を計算し，ソースコード補完をすることである．

そのために，事前に既存のソースコードを学習させ，次に存在するであろうメソッド呼び出し文の割合を計算しておく必要がある．学習フェーズでは，まず最初にソースコードを解析しすべての呼び出し文を抽出する．その後，メソッド単位で呼び出し文を順番に並べる．この際，後の学習でメソッド呼び出し文を単語として処理するために，メソッド呼び出し文を変換ルールに従った文字列に変換

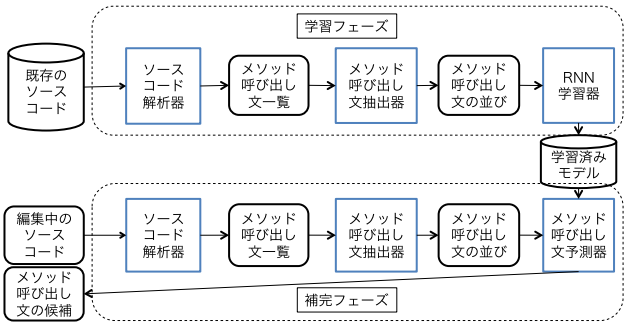

図 3 本手法の流れ
Fig. 3 Flow of proposed method.

する．さらに，メソッドの区切りを特殊な文字列として挿入する．学習フェーズの最後として，作成したメソッド呼び出し文の並びを文書と見なして，RNN 学習器に入力する．モデル構築するために，2.2 節で述べた RNNLM を利用する．

補完フェーズでは，開発者が補完させたいソースコード断片を基に処理を始める．ソースコード断片中から補完したい場所の直前のメソッド呼び出し文を抽出し，学習フェーズで利用した変換ルールに従い，文字列に変換する．そして，予測器と学習済みのモデルを利用してソースコード補完の候補を取得し，開発者に提示する．

3.2 ソースコード解析器

ソースコード解析器では，最初に，既存のソースコードを構文解析，意味解析し，ソースコード中のクラス内のメソッド一覧を取得する．その後，各メソッド中のメソッド呼び出し文とインスタンス生成文を抽出する．抽出する際，抽出する文の順序はソースコードの字面上の順番とし，制御文による実際の実行順序ではないものとする．

この際，メソッド呼び出し文のオブジェクトをクラスの完全限定名に変換する．変換時に，多様性は考慮しないものとする．クラスの完全限定名が自クラスだった場合はライブラリやフレームワークの API の利用ではないと判断し，無視するものとする．クラスに総称型が利用されていた場合は，総称型の名前を無視して変換する．メソッド名はそのまま利用するが，インスタンス生成文の場合はメソッド名として特殊な名前である<init>に置き換える．また，パラメータは考慮しないものとする．最後に，クラス名とメソッド名を#文字で結合した文字列を生成する．

たとえば，java.util.ArrayList<String>クラスの add メソッドを呼び出す文が存在した場合は，java.util.ArrayList#add と変換する．また，new java.util.ArrayList<String>() 文の場合は，java.util.ArrayList#<init>と変換する．

3.3 メソッド呼び出し文抽出器

補完したい API に必要なメソッド呼び出し文を 3.2 節で抽出したメソッド呼び出し文から抽出する．たとえば，JDK のみを対象にした場合は java で始まるクラス名を対象にすればよい．フレームワークやライブラリの種類を限定することで，精度の高い補完が期待できる．

抽出したメソッド呼び出し文の最後にメソッドの区切り単語を表す<eom>という文字列を付与する．ただし，メソッド呼び出し文が 1 つしか存在しないメソッドは除外する．これらのメソッドは単純な委譲メソッドである可能性もあり，メソッド呼び出し文の並びを予測するのには適さないと考えるためである．

図 1 を変換した例を**図 4** に示す．この例では，android で始めるクラス名しか呼び出しておらず，すべてのメソッド呼び出し文を抽出した．

```
1 android.view.LayoutInflater#inflate
2 android.view.View#findViewById
3 android.widget.TextView#setText
4 android.view.View#findViewById
5 android.widget.Button#setOnClickListener
6 android.view.View.OnClickListener#<init>
7 <eom>
```

図 4 メソッド呼び出し文の並び
Fig. 4 List of method call statements.

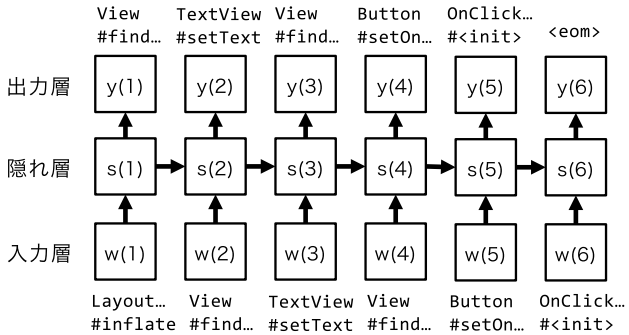

図 5 時刻で展開した RNNLM
Fig. 5 RNNLM unfolded in time.

最後に，すべてのメソッドについて上記の変換が完了すると，変換後の並びを 1 つのファイルにまとめる．このファイルを，<eom>で区切られた文書として扱い，RNNLM の入力とする．

3.4 RNN 学習器

2.2 節で説明した RNNLM を用いて学習を行う．まず，3.3 節で作成した文書ファイル中に存在する語彙数を数える．この際，変換されたメソッド呼び出し文の文字列を単語として扱う．語彙数が求まると，各単語の 1-of-N 表現が可能となる．そして，1 行目を時刻 1 の単語として扱い，2 行目を時刻 2 の単語として扱っていき，最終行まで順番に RNNLM で処理する．truncated BPTT の長さをどの程度にするかを考えたとき，固定長にするのではなく，<eom>が次の単語になった時点の長さで BPTT を実行する．さらに，BPTT を実行後には隠れ層である LSTM の状態を初期化する．Java のソースコードにおいてメソッドの並びは任意の場所でよく，記述する順番は考慮されない．そこで，<eom>が出現し，次のメソッドに移る際は，別の系列と判断し，内部状態を初期化する．

図 4 のメソッド呼び出し文の並びを例にして学習の過程を説明する．この例を入力として，RNNLM を時刻で展開して適用したネットワークは**図 5** のようになる．まず，1 行目の android.view.LayoutInflater#inflate を入力として計算し出力層から出力されるベクトルを求める．ここ

で，この出力されたベクトルを $y(1)$ とする．図 5 における一番左の縦 3 つの四角形がこの処理にあたる．次に，2 行目の android.view.View#findViewById を入力としてベクトル $y(2)$ を計算する．この際，前回（1 行目の処理）の隠れ層 s(1) から出力される結果も今回（2 行目の処理）の隠れ層 s(2) の入力に加え計算する．この処理を<eom>が出現するまで繰り返していくと，$y(1)$ から $y(6)$ の 6 つのベクトルが得られる．$y(1)$ は 2 行目の android.view.View#findViewById を表すベクトルであることが望ましいはずである．そこで，$y(1)$ と android.view.View#findViewById を 1-of-N 表現したベクトルとの誤差を求める．同様に，$y(2)$ と 3 行目の android.widget.TextView#setText との誤差を求め，最終的に $y(1)$ から $y(6)$ のすべての誤差を求め，それらの誤差をすべて累積した累積誤差を求める．最後に，確率的勾配降下法を用いて累積誤差が小さくなるようにニューラルネットワーク内の重みを計算する．この重みが学習済みモデルとなる．

学習させる際のパラメーターとしては，隠れ層の数とそれぞれの隠れ層のユニット数 M，さらに，同じ文書を何回学習させるかの数である epoch 数があげられる．

3.5 メソッド呼び出し文予測器

学習済みのモデルを利用し，次に出現するであろうメソッド呼び出し文の候補を取得する．エディタ等で記述している編集中のソースコードから，補完したい箇所を囲むメソッドと特定し，そのメソッド内のメソッド呼び出し文をすべて抽出する．抽出には学習フェーズで利用したソースコード解析器を利用する．そして，抽出したメソッド呼び出し文を 3.2 節で説明した変換ルールと同様のルールに従い文字列に変換する．この際，メソッド内のソースコードが完成していないため，<eom>は付与しない．

変換した文字列を入力として，学習済みのモデルから次に出現するであろう単語（メソッド呼び出し文）の確率を求め，その確率順に整列をする．1 行ずつ単語（変換後のメソッド呼び出し文）を RNN に入力していき，最終行の単語を入力した際に出力される単語の出現確率を補完候補として利用する．その後，候補を出現確率の大きい順に並び替え，候補の上位を開発者に提示する．ただし，モデルに単語を入力していく際，語彙として登録されていないメソッド呼び出し文が出現した場合は無視する．

図 1 の 7 行目までしか入力されていない場合を例として，予測器の動作を説明する．7 行目までに含まれるメソッド呼び出し文は図 4 の 1 行目から 3 行目に相当する．事前に多くの Android のライブラリを用いたソースコードから学習済みモデルは構築されているとする．そして，学習済みモデルに保存されている重みをニューラルネットワークの各重みにセットする．まず，1 つ目の android.view.LayoutInflater#inflate をニューラルネットワークの入力として計算を始める．次に，2 つ目の android.view.View#findViewById を入力して計算し，最後に，3 つ目の android.widget.TextView#setText を入力して計算する．3 つ目の入力に対応した出力層の出力ベクトル $y(3)$ が次に出現するであろうメソッド呼び出し文の確率となっている．もちろん，2 つ目の中間層の入力には，1 つ目の中間層の出力を入力している．つまり，図 5 の左半分と同様の計算することになる．

出力層には 2.2 節でも述べたように，softmax 関数を用いて出力ベクトルの値を得ている．ベクトルの次元数は語彙数（総メソッド呼び出し文数）であり，softmax 関数を適用しているためベクトルの各要素の合計値は 1 となる．そのため，1-of-N 表現した際に付けた通し番号とベクトルの要素番号を一致させることで，各要素の値を各呼び出し文の出現確率と見なすことができる．理想的な結果な場合，$y(3)$ は次に出現する android.view.View#findViewById の 1-of-N 表現（android.view.View#findViewById の要素が 1，他の残りの要素は 0）になる．

4. 実験

提案手法が実用に耐えうるかを評価するために実験を行った．本章では，以下の実験を実施した．実用的な速度で実現可能かを検証するためのパフォーマンス評価について記述し，その後，本手法の補完候補精度に関する実験結果について示す．

- モデル構築のパフォーマンス実験
- Top-k の精度．補完候補上位何位に正解文が出現するかの計測
- モデルのパラメータ変化による精度の変化の確認
- プロジェクトの違いによる精度の変化の確認
- 他手法との比較

4.1 実験対象ソフトウェア

表 1 に実験対象ソフトウェアの情報を示す．本実験では，Android アプリケーションの作成に焦点を当てた．10 個のオープンソースソフトウェアを対象とした．Android API の補完を対象にするため，実験のメソッド呼び出し文抽出器においては，android で始まるクラス名のみを抽出候補とした．

表 1 の android-23 [*1] は，Android SDK（API 23）に含まれるサンプルソースコードを表す．それ以外のプロジェクト（Android-Universal-Image-Loader [*2]，MPAn-

[*1] http://developer.android.com/intl/ja/sdk/index.html#downloads
[*2] https://github.com/nostra13/Android-Universal-Image-Loader

表 1 実験対象プロジェクト
Table 1 Target projects.

プロジェクト名	ファイル数	LOC	メソッド数
android-23	1,531	212,670	3,058
Android-Universal -Image-Loader	88	13,857	64
MPAndroidChart	219	40,406	252
SlidingMenu	33	4,932	67
ViewPagerIndicator	42	4,059	36
butterknife	91	10,035	26
fresco	601	83,821	279
glide	432	52,869	272
iosched	378	62,604	631
picasso	76	13,293	99

droidChart[*3]，SlidingMenu[*4]，ViewPagerIndicator[*5]，butterknife[*6]，fresco[*7]，glide[*8]，iosched[*9]，picasso[*10]）は，GitHub に登録されている Android アプリケーションの中で Java ファイルを含むプロジェクトを選択した．選択した基準は，プロジェクトに付けられたスターの数が多いものとした．

表 1 のファイル数は，プロジェクト内の Java ファイルの総数であり，LOC は総行数を表す．メソッド数は，RNN 学習器に学習させるメソッドの数である．メソッド呼び出し文抽出器において，特定の呼び出し文（先頭が `android`）だけを抽出し，さらに，メソッド呼び出し文が 1 個以下のメソッドは除外されるため，実際のメソッド数よりは少なくなる．

4.2 学習フェーズのパフォーマンス評価

学習フェーズで作成するモデルの構築に利用に要した時間と構築したモデルのサイズについて調査した．

表 1 のすべてのプロジェクトを対象に学習させた．10 プロジェクトの総ファイル数は 3,491 ファイルになり，その中の計 4,784 メソッドを対象とした．すべてのメソッドを変換ルールに基づき変換した後の文書ファイルには 29,463 単語（メソッド呼び出し文）存在し，語彙数にすると 3,019 となった．

本実験では，RNN 学習器は Chainer[*11]を利用して作成し，文書ファイルを NVIDIA GeFroce GTX 970 4 GB memory GPU で学習させた．学習にかかった時間は 1 分 28 秒であり，学習済みモデルのサイズは約 26 MB になった．隠れ層を 2 層とし，ユニット数 M を 512 としたパラメータでネットワークを構築し，epoch 数を 1 としてモデル構築を行った．

実験結果から，実用上問題ない時間で構築できる．精度を上げるためには，訓練用のソースコードを増やす必要性があるが，モデル構築は訓練用のソースコードに変更がなければ最初に一度構築すればよく，問題ないと考える．

4.3 有効性評価

手法の有効性を評価するために，以下の手順で実験を実施する．まず，表 1 の 10 個のプロジェクトを訓練用のプロジェクトと評価用のプロジェクトの 2 種類に分類する．分類後，訓練用のプロジェクトを用いてモデルを構築する．そして，構築したモデルを利用して，評価用のプロジェクトのソースコードが補完可能かを評価する．

次に評価方法について説明する．評価用のプロジェクトのソースコードのすべてのメソッドを，ソースコード解析器とメソッド呼び出し文抽出器を用いて，メソッド呼び出し文の並びに変換する．あるメソッドのメソッド呼び出し文の並びが，$a_1, a_2, \ldots, a_k$, `<eom>` だったとする．まず，a_1 まで入力されていたとして，a_2 が補完候補の何位に出現するか計測する．同様に，a_2 まで入力されていたとして，a_3 が補完候補の何位に出現するか計測する．最終的に a_{k-1} まで計測を実施する．k 個の呼び出し文が並んでいたときは，$k-1$ 回の試行を実施することになる．もちろん，a_{k-1} までの入力があったときは，a_1 から a_{k-1} のすべてが入力されていると考え，順番にモデルに入力する．その後，最終的な出力を a_k の候補と見なして計測する．そして，この計測を評価用プロジェクトのすべてのメソッドについて行う．今回の実験では，補完候補のメソッド呼び出し文の取得する際，クラス名は分かっているものとして計測を行う．これは，通常の統合開発環境と同様の補完と似ている．

図 1 と図 4 を例として考える．図 1 をソースコード解析器とメソッド呼び出し文抽出器を用いて変換した呼び出し文の並びが図 4 である．まず最初に，図 4 の 1 行目である `android.view.LayoutInflater#inflate` と `android.view.View` 型の変数までが入力されたと考え，補完候補を取得する．取得結果を**表 2** に示す．さらに，`android.view.LayoutInflater#inflate` と `android.view.View#findViewById` の 2 行と `android.widget.TextView` 型の変数までが入力されたと考え，補完候補を取得する．取得結果を**表 3** に示す．図 4 のメソッドの場合は 6 個のメソッド呼び出し文が抽出されている．そのため，5 行目まで入力された場合の計測まで実施し，計 5 回実施することになる．なお，取得結果は，4.2 節で実験した学習モデルを用いている．

4.3.1 Top-k の精度

android-23 を評価用プロジェクトに，それ以外の 9 つのプロジェクトを訓練用プロジェクトとして計測した結果を

[*3] https://github.com/PhilJay/MPAndroidChart
[*4] https://github.com/jfeinstein10/SlidingMenu
[*5] https://github.com/JakeWharton/ViewPagerIndicator
[*6] https://github.com/JakeWharton/butterknife
[*7] https://github.com/facebook/fresco
[*8] https://github.com/bumptech/glide
[*9] https://github.com/google/iosched
[*10] https://github.com/square/picasso
[*11] http://chainer.org/

表 2 android.view.LayoutInflater#inflate の後の呼び出し文候補
Table 2 Method calls following android.view.LayoutInflater#inflate.

順位	呼び出し文の候補	出現確率
1	android.view.View#findViewById	0.007
2	android.view.View#getBottom	0.004
3	android.view.View#getViewTreeObserver	0.003
4	android.view.View#getHeight	0.003
5	android.view.View#getTop	0.003
6	android.view.View#setTag	0.002
7	android.view.View#setVisibility	0.002
8	android.view.View#setFocusable	0.002
9	android.view.View#setOnClickListener	0.002
10	android.view.View#getLeft	0.002
11	android.view.View#getMeasuredHeight	0.002
12	android.view.View#requestLayout	0.002
13	android.view.View#setLayoutParams	0.002
14	android.view.View#getVisibility	0.001
15	android.view.View#getTag	0.001
16	android.view.View#offsetLeftAndRight	0.001
17	android.view.View#<init>	0.001
18	android.view.View#getLayoutParams	0.001
19	android.view.View#setClickable	0.001
20	android.view.View#getPaddingBottom	0.001

表 3 android.view.View#findViewById の後の呼び出し文候補
Table 3 Method calls following android.view.View#findViewById.

順位	呼び出し文の候補	出現確率
1	android.widget.TextView#setText	0.002
2	android.widget.TextView#<init>	0.002
3	android.widget.TextView#setVisibility	0.001
4	android.widget.TextView#setTypeface	0.001
5	android.widget.TextView#setTextSize	0.000
6	android.widget.TextView#setMovementMethod	0.000
7	android.widget.TextView#setBackgroundResource	0.000
8	android.widget.TextView#setPadding	0.000
9	android.widget.TextView#setTag	0.000
10	android.widget.TextView#setOnClickListener	0.000
11	android.widget.TextView#append	0.000
12	android.widget.TextView#setTextColor	0.000
13	android.widget.TextView#setAllCaps	0.000
14	android.widget.TextView#getText	0.000
15	android.widget.TextView#setGravity	0.000
16	android.widget.TextView#setContentDescription	0.000
17	android.widget.TextView#setLayoutParams	0.000
18	android.widget.TextView#setId	0.000
19	android.widget.TextView#announceForAccessibility	0.000

表 4 と**図 6** に示す．隠れ層を 2 層にし，それぞれのユニット数 M を 512 としたネットワークを作成し，epoch 数を 1 回から 30 回までの 30 通りの学習モデルを作成して計測を行った．表と図の横軸は epoch 数である．Top-k は補完候補の k 番目までに候補が含まれている割合を示している．Top-1 の epoch 数が 2 のときに 38%という割合になっている．android-23 のソースコード中に 8,482 カ所補完する箇所が存在し，3,249 カ所で補完候補に 1 位に正解候補が現れたことを示している．

3 割以上の確率で 1 位に候補が出現することが分かる．また 5 位以内を考えると，7 割になることが分かる．補完候補をすべての箇所で計測したことを考えると，高い精度で補完表示されていると考えられる．

4.3.2 epoch 数と精度

表 4 と図 6 の epoch 数と精度の関係を見ると，epoch 数を増加させても精度は変わらない，もしくは下がる傾向にある．訓練させすぎることで訓練用データに過剰に適合し，評価プロジェクトから外れる傾向にあることが分かる．訓練データ自体に同じような API の並びが多数あると思われることから，学習モデルを構築する際に epoch 数を多くする必要はないと考えられる．

4.3.3 隠れ層のパラメータ

epoch 数を 1 として，隠れ層の層の数を 1 層，2 層，3 層に変化させた場合に精度にどの程度差が出るか実験した．各層のユニット数 M はすべて 512 としたネットワークを作成した．結果を**表 5** と**表 6** と**表 7** にそれぞれ示す．層を増やすごとにわずかながら，精度が向上している．しかしながら，精度の差はわずかな差である．層を増やしていくと層の数に比例してモデルの構築時間が増加するため，むやみに増やす必要はないと考える．

次に隠れ層のユニット数を変化させた場合に精度が変わるかどうかを計測した．隠れ層を 2 層として，ユニット数を 1,024 とした場合の結果を**表 8** に示す．こちらもユニット数が 512 の場合と比べて大きな変化は見られなかった．

隠れ層を増やしたりユニット数を増やすと学習するための時間が余計に必要になる．そのため，1 層また 2 層，ユニット数も 512 で十分だと考えられる．

4.3.4 学習モデルの違いによる精度の変化

訓練プロジェクトと評価プロジェクトを変更することで，どのように精度が変わるかを測定した．結果を**表 9** に示す．評価プロジェクトを 1 つにし，残り 9 個のプロジェクトを訓練プロジェクトにした．10 個のプロジェクトを用いているため，10 個の組合せが存在する．android-23 を評価プロジェクトにした場合が最も精度が高い結果となった．最も低い評価プロジェクトは glide となった．

android-23 は典型的なサンプルソースコード集であり，訓練プロジェクトに多数 API の並びが記述されているためだと考える．また，glide は画像をロードするためのライ

表 4 android-23 を評価した場合の精度一覧
Table 4 A summary of the accuracy of andorid-23.

横軸はエポック数（1〜30）

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Top-1	38%	38%	38%	35%	33%	34%	33%	33%	34%	32%	33%	31%	33%	31%	32%
Top-5	75%	74%	73%	72%	72%	71%	71%	71%	72%	70%	72%	70%	71%	70%	70%
Top-10	88%	89%	88%	88%	88%	86%	87%	86%	87%	88%	88%	86%	87%	86%	85%
Top-20	97%	97%	96%	96%	96%	96%	96%	96%	96%	96%	96%	96%	95%	95%	94%
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Top-1	31%	31%	31%	31%	32%	31%	32%	33%	31%	30%	31%	31%	31%	30%	30%
Top-5	69%	69%	70%	70%	69%	69%	68%	69%	68%	67%	70%	69%	68%	68%	68%
Top-10	85%	85%	85%	85%	85%	84%	85%	86%	84%	85%	85%	85%	84%	85%	85%
Top-20	95%	95%	95%	95%	95%	94%	95%	96%	95%	95%	95%	95%	94%	95%	95%

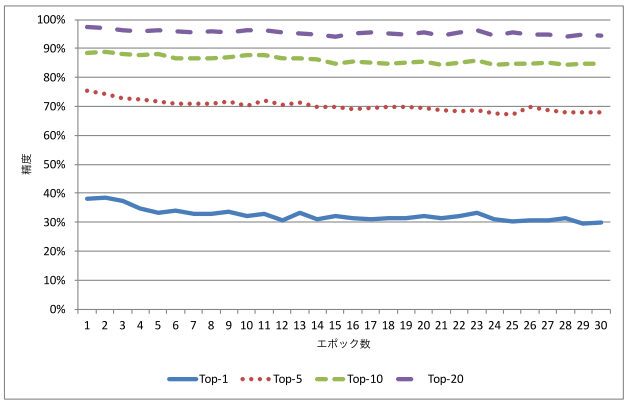

図 6 android-23 を評価した場合の精度
Fig. 6 A line graph of the accuracy of andorid-23.

表 5 隠れ層が 1 層の結果
Table 5 Results of 1 hidden layer.

（表中の「数」は 8,482 カ所中に何カ所で適切な結果が得られたかを表す数）

	数	精度
Top-1	3,012	35%
Top-5	6,062	71%
Top-10	7,360	86%
Top-20	8,129	95%

表 6 隠れ層が 2 層の結果
Table 6 Results of 2 hidden layers.

（表中の「数」は 8,482 カ所中に何カ所で適切な結果が得られたかを表す数）

	数	精度
Top-1	3,242	38%
Top-5	6,398	75%
Top-10	7,494	88%
Top-20	8,255	97%

ブラリで特殊な API の並びが存在したため，訓練プロジェクトに似たような API の並びがなく，精度が低くなったと考える．

表 7 隠れ層が 3 層の結果
Table 7 Results of 3 hidden layers.

（表中の「数」は 8,482 カ所中に何カ所で適切な結果が得られたかを表す数）

	数	精度
Top-1	3,284	38%
Top-5	6,528	76%
Top-10	7,573	89%
Top-20	8,232	97%

表 8 隠れ層が 2 層・ユニット数が 1,024 の結果
Table 8 Results of 2 hidden layers and 1,024 Units.

（表中の「数」は 8,482 カ所中に何カ所で適切な結果が得られたかを表す数）

	数	精度
Top-1	3,219	37%
Top-5	6,386	75%
Top-10	7,506	88%
Top-20	8,257	97%

4.3.5 他手法との比較

Robbes らはソースコードの変更履歴からメソッド名の推薦を実現するためにいくつかのアルゴリズムを提案している [15]．提案手法を評価するためにメソッド名の最初の n 文字を入力した時点での Top-k 精度を計測している．先頭の 2 文字を入力した時点で，最も精度が良かったアルゴリズムの Top-1 精度は約 60%で，Top-10 精度は約 88%であった．

精度を比較するために，本手法もメソッドの先頭 2 文字を入力した時点での精度を計測した．まず，4.3 節と同様の手法で補完候補を取得する．その後，その補完候補の中から先頭 2 文字と一致する結果のみを抽出したものを最終的な補完候補とした．**表 10** に実験結果を示す．ViewPageIndicator プロジェクトの Top-1 精度は 50%をきっているが，Top-10 精度は 90%となり，Robbes らの手法より精度が高い．また，多くのプロジェクトで Top-1 精度，

表 9 プロジェクトごとの比較
Table 9 A comparison among projects.

評価プロジェクト	Total	Top-1	Top-5	Top-10	Top-20	Top-1 accuracy	Top-5 accuracy	Top-10 accuracy	Top-20 accuracy
Android-Universal-Image-Loader	178	59	143	154	170	33%	80%	87%	96%
MPAndroidChart	1,181	393	804	1,011	1,112	33%	68%	86%	94%
SlidingMenu	210	61	140	178	193	29%	67%	85%	92%
ViewPagerIndicator	184	40	89	128	178	22%	48%	70%	97%
android-23	8,482	3,242	6,398	7,494	8,255	38%	75%	88%	97%
butterknife	47	14	21	28	29	30%	45%	60%	62%
fresco	751	150	443	604	682	20%	59%	80%	91%
glide	646	129	494	572	613	20%	76%	89%	95%
iosched	2,700	856	1,883	2,282	2,568	32%	70%	85%	95%
picasso	189	46	115	154	177	24%	61%	81%	94%

表 10 メソッド名の先頭 2 文字を入力したときの精度
Table 10 The accuracy of the first 2 letters of method name.

評価プロジェクト	Top-1 accuracy	Top-10 accuracy
Android-Universal-Image-Loader	60%	94%
MPAndroidChart	63%	94%
SlidingMenu	52%	95%
ViewPagerIndicator	40%	90%
android-23	68%	97%
butterknife	89%	100%
fresco	61%	97%
glide	61%	97%
iosched	56%	96%
picasso	58%	96%

Top-10 精度ともに高い結果となった．さらに，butterknife プロジェクトが最も高いプロジェクトであり，Top-1 精度は 89%となっている．実験の対象としているソースコード群は異なるが，既存のメソッドの推薦手法に比べ精度が高い結果が得られている．

4.4 考察

本手法は Mikolov らが提案した RNNLM を利用したコード推薦手法である．ただし，自然言語における段落の扱いを変更して利用している．通常，自然言語の文や段落の順序には意味がある．そこで，RNNLM では区切りを表す特殊文字を利用して文間を区切って学習させている．一方，クラス内のメソッドの記述順序には決まりがない．そのため，メソッド単位でニューラルネットワークの重みを計算するように変更を加えている．

本実験で最も高い Top-1 精度は 38%であった．一般的に，Top-1 の精度を 100%にするのは困難である．同じような処理を記述する場合でも，組織や開発者でソースコードの書き方が変わるためである．たとえば，API の処理の流れとして，open - read - close という処理と open - read - read - close という 2 種類のソースコードがあった場合を考える．ともに，open - read まで記述した場合，推薦してほしい API は，それぞれ，close と read になる．そのため，この 2 つのソースコードがともに評価用プロジェクトに含まれていた場合，Top-1 精度は 100%にはならない．そのため，Top-5 や Top-10 といった複数の候補の中に必要な API があるかどうかを考えることも重要だと考える．表 4 を見ると，Top-5 で 70%，Top-10 で 80%程度の精度があり，推薦候補の中に必要な API が高い確率で含まれていると考えることできる．

また，本手法はメソッド内に記述されたメソッド呼び出し文の順序をそのまま利用するため，メソッド内に複数の処理が混在している場合は精度が悪化する場合がある．今後は，メソッド内の制御構造やオブジェクトに着目した API の呼び出し順序を考慮し，学習モデルを構築することで精度が向上すると考える．

4.5 妥当性への脅威

本研究で利用している RNNLM は自然言語処理の分野ですでに提案されており，いくつかの実験を通して評価されているモデルである．そのため，RNNLM 自体は信頼できるモデルであると考えられる．自然言語における単語の並びをメソッド呼び出し文の並びに置き換えて学習させた結果，他のコード推薦手法に比べて高い精度で推薦できる結果となった．ただし，RNNLM の各種パラメータの値によっては，この結果が大きく変わる可能性がある．

本実験では Andorid アプリケーションを対象として実験を実施した．そのため，他のドメインのアプリケーションを対象とした場合は精度が異なる傾向になる可能性がある．しかし，本手法はある特定のドメインの API に特化した手法ではなく，一般的に適用できる手法である．ただし，学習済みモデルを作成する際，どのドメインの API 利用例を多く含むソースコードから構築したかによって，推薦精度が変わる可能性がある．学習済みモデルに存在しな

いAPIを用いたソースコードを新規に記述する場合は，推薦候補に適切なAPIが含まれない場合も存在する．そのため，学習済みモデルを構築する際は，新規に開発するソースコード内で利用するであろうAPIを網羅する学習済みモデルを構築することが望まれる．

5. 関連研究

APIの情報を既存のソースコードから抽出し，役立たせる研究は数多く存在する．Prospector [8] やXsnippet [16] は，あるメソッドの返り値を利用して，さらにメソッドを呼び出すといった連鎖がある場合に，その情報をデータベースに入れておきコードアシストをするツールである．PARSEWeb [18] は，既存のコードサーチエンジンを利用し，関係するソースコードを取得し，提示するツールである．これらのツールは，あるクラスから別のクラスの情報を取得するときに，どのようなメソッド呼び出しの連鎖が必要かを提示する．一方，我々のツールはあるメソッド呼び出し群があった場合に，次にくるメソッド呼び出しとして，どのような文が適切かを予測するツールであり，使用用途が異なる．また，Michailらは，アソシエーション分析を利用して，APIの再利用パターンを抽出している [9], [10]．Strathcona [6], [7] は，ソースコードの構造に基づき，コード例を推薦してくれるツールである．ただし，コード例を提示するシステムであり，文単位での推薦はしてくれない．

APIに着目しコード検索を行う手法としてMishneらの手法 [13] がある．この手法では，APIの実行順に着目し，そのAPIの並びをクエリーとしてコード検索を行う．コード検索のための手法であるため，そのままでは補完に適さない．

既存のソースコードを利用したコード補完手法にBruchら [4] の手法がある．この手法は，あらかじめフレームワーク等でオーバライドして記述するメソッド内でよくつかわれるメソッド一覧を既存のソースコードから作成しておく．そして，そのフレームワークを利用してオーバライドするメソッドを開発する際に，そのメソッド内で最適なメソッドを提示してくれるものである．

コード補完や省略語の補完に隠れマルコフモデル（HMM）を採用した研究も多くなされている．Hanら [5] はHMMを利用して，省略語の入力を補完する手法を提案している．省略語を入力して，その文字列を基に補完する手法であり，APIを補完する手法ではない．また，Nguenら [14] は，HMMを利用して，Andoridアプリの APIを補完する手法を提案している．さらに，ASTLan [1] は，ASTベースの言語モデルを提案している．ASTを扱うため，制御構造を考慮しており，より精度の高い補完を実現している．CSCC [2] は，コンテキストをAPI推薦手法を提案している．クラス名や，メソッド名や，キーワードをコンテキストと考え，補完のための情報として取得している．

また，[20] で2つのAPI呼び出し間の関係を情報として利用して，メソッド呼び出し文自体の推薦を行う手法がある．既存のソースコード内にAPI呼び出しのペアがどの程度存在するかといった情報をソースコードコーパスとして保存し，そのコーパスを元にコード補完をする．この手法だと，コーパス内に存在するペア以外は補完できないという問題が存在する．一方，本手法は既存のソースコード内に記述されているメソッド呼び出し文であれば候補に現れる可能性がある．

6. まとめと今後の課題

本研究では，回帰結合ニューラルネットワークに着目し，メソッド呼び出し文を補完する手法について提案した．事前に，既存のソースコードのメソッド呼び出し文の並びを学習させ，ソースコードのAPI補完に利用する．

そして，提案する手法を実装し，10プロジェクトのオープンソースソフトウェアを用いて実験を行った．さらに，回帰結合ニューラルネットワークの様々なパラメータが実験結果にどのように影響するかを調査した．実験の結果，典型的なサンプルソースコードの補完においては，38%の精度で補完候補の1位に必要なメソッド呼び出し文が現れることが確認できた．

今後は，様々な種類のソースコードに対して学習させ，手法が有効であるか確認することがあげられる．また，APIの流れを補完候補として提示することもあげられる．一文だけではなく，今後必要となるであろうすべてのメソッド呼び出し文が分かれば，さらなる生産性の向上につながると考える．

謝辞 本研究はJSPS科研費 15K00108の助成を受けたものである．

参考文献

[1] Nguyen, A.T. and Nguyen, T.N.: Graph-Based Statistical Language Model for Code, *Proc. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp.858–868 (2015).

[2] Asaduzzaman, M., Roy, C.K., Schneider, K.A. and Hou, D.: CSCC: Simple, Efficient, Context Sensitive Code Completion, *Proc. 2014 IEEE International Conference on Software Maintenance and Evolution*, pp.71–80 (2014).

[3] Bengio, Y., Ducharme, R., Vincent, P. and Janvin, C.: A Neural Probabilistic Language Model, *The Journal of Machine Learning Research*, Vol.3, pp.1137–1155 (2003).

[4] Bruch, M., Monperrus, M. and Mezini, M.: Learning from examples to improve code completion systems, *Proc. 7th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.213–222 (2009).

[5] Han, S.H.S., Wallace, D. and Miller, R.: Code Completion from Abbreviated Input, *Proc. 24th IEEE/ACM*

International Conference on Automated Software Engineering, pp.332–343 (2009).
[6] Holmes, R. and Murphy, G.C.: Using structural context to recommend source code examples, *Proc. 27th International Conference on Software Engineering*, pp.117–125 (2005).
[7] Holmes, R., Walker, R. and Murphy, G.: Approximate Structural Context Matching: An Approach to Recommend Relevant Examples, *IEEE Trans. Software Engineering*, Vol.32, No.12, pp.952–970 (2006).
[8] Mandelin, D., Xu, L., Bodik, R. and Kimelman, D.: Jungloid mining: Helping to navigate the API jungle, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.48–61 (2005).
[9] Michail, A.: Data mining library reuse patterns using generalized association rules, *Proc. 22nd International Conference on Software Engineering*, pp.167–176 (2000).
[10] Michail, A.: Browsing and searching source code of applications written using a GUI framework, *Proc. 24th International Conference on Software Engineering*, pp.327–337 (2002).
[11] Mikolov, T., Karafiat, M., Burget, L., Cernocky, J. and Khudanpur, S.: Recurrent Neural Network based Language Model, *Interspeech 2010*, pp.1045–1048 (2010).
[12] Mikolov, T. and Kombrink, S.: Extensions of recurrent neural network language model, *ICASSP*, pp.5528–5531 (2011).
[13] Mishne, A., Shoham, S. and Yahav, E.: Typestate-based semantic code search over partial programs, *ACM SIGPLAN Notices*, Vol.47, No.10, pp.997–1016 (2012).
[14] Nguyen, T.T., Pham, H.V., Vu, P.M. and Nguyen, T.T.: Recommending API Usages for Mobile Apps with Hidden Markov Model, *Proc. 2015 IEEE/ACM International Conference on Automated Software Engineering*, pp.795–800 (2015).
[15] Robbes, R. and Lanza, M.: Improving code completion with program history, *Automated Software Engineering*, Vol.17, No.2, pp.181–212 (2010).
[16] Sahavechaphan, N. and Claypool, K.: XSnippet: Mining For sample code, *Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp.413–430 (2006).
[17] Syer, M.D., Nagappan, M., Hassan, A.E. and Adams, B.: Revisiting Prior Empirical Findings For Mobile Apps: An Empirical Case Study on the 15 Most Popular Open-Source Android Apps, *CASCON: Conference of the Center for Advanced Studies on Collaborative Research*, pp.283–297 (2013).
[18] Thummalapenta, S. and Xie, T.: Parseweb: A programmer assistant for reusing open source code on the web, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp.204–213 (2007).
[19] Werbos, P.J.: Backpropagation Through Time: What It Does and How to Do It, *Proc. IEEE*, Vol.78, No.10, pp.1550–1560 (1990).
[20] 山本哲男：制御構造を考慮したソースコードコーパスに基づくメソッド呼び出し文補完手法，情報処理学会論文誌，Vol.56, No.2, pp.682–691 (2015).

山本 哲男 （正会員）

平成 9 年大阪大学基礎工学部情報工学科卒業．平成 14 年同大学大学院博士後期課程修了．同年科学技術振興事業団計算科学技術研究員．平成 16 年立命館大学情報理工学部情報システム学科講師．平成 20 年同学科准教授．平成 23 年日本大学工学部情報工学科准教授．博士（工学）．ソフトウェア開発支援環境の研究に従事．電子情報通信学会，IEEE-CS 各会員．