Vol.2019-NL-242 No.3
2019/10/25

未知語と低頻度語に対応したソースコードにおける関数名の自動生成

藤田 正悟　上垣外 英剛　高村 大也　奥村 学

概要：ソースコード中には低頻度語や，特定の関数にのみ出現する単語が存在するが，既存の関数名の自動生成手法では一定の頻度以上で出現する単語のみを対象としているため，これらの事例に適切に対処することが難しい．そこで本研究では，ソースコードにおける関数名の自動生成において，未知語や低頻度語に対応するための単語の置き換え手法とコピー機構を用いる手法を提案する．GitHub で star 数上位のリポジトリを用いて作成された 89,393 件のソースコードデータセットを用いて評価を行った結果，提案手法は従来手法に対し F1 において 1.64 の改善を示した．

1. はじめに

現在，GitHub[*1]に代表されるように，複数人によるソースコードの共有が盛んに行われている．その一方で，ソースコード中の関数名は実装者により自由に命名されることから，必ずしも実際の処理に関連するもの，あるいは実際の処理を連想させるものが使用されているとは限らない．このため，関数名と実際の処理が乖離することによる可読性の低下は，複数人によるソースコードの共有を行う際に頻繁に生じる問題の一つとなっている．例えば，ソースコード 1 は，内部では配列 `this.elements` の中で，id が引数と一致しているオブジェクトのインデックスを返り値としているが，関数名だけを見るとさもオブジェクトの持つなんらかの値を返り値として持つように感じてしまう．

このような問題を解決するために，現在，ソースコードから関数名を自動生成したり自動推定したりする研究が行われている．例えば，ソースコード 1 の例であれば，`indexOfTarget` といったより内部の動作にあった関数名を生成する．特に，Uri らが提案した code2seq [3] という手法は，自動翻訳などで使用されるニューラル系列変換モデル（seq2seq）を用いることにより，現在最高精度を達成している．しかし，code2seq では，訓練データで出現回数が一定以上の単語しか出力できないため，未知語や低頻度語を部分文字列として含む関数名の推定が困難である．図 1 はソースコード中の関数名とその頻度を示している．縦軸が出現頻度であり，横軸が頻度の順位である．この結果は，ソースコードにおいて関数名はジップの法則に従い，関数名には低頻度あるいは未知の文字列が大量に含まれていることを示唆している．

この問題を解決するために，本論文では，訓練データの語彙に含まれていない未知語や低頻度語を文字列として含むような関数名についても，よ

```
int getTargetValue(int target_id) {
    int i=0;
    for (Object elem: this.elements) {
        if (elem.id == target_id) {
            return i;
        }
        i++;
    }
    return -1;
}
```

ソースコード 1　不適切なソースコードの例

[*1] https://github.com

ⓒ 2019 Information Processing Society of Japan

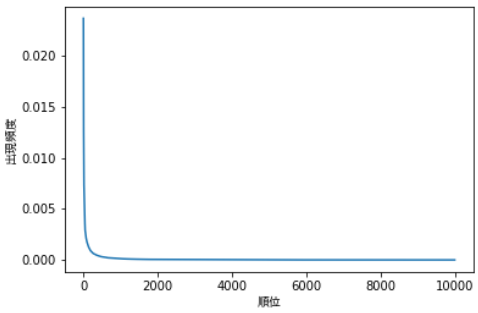

図 1 頻度上位 10000 語の出現頻度

り正しく生成できる方法を提案する．提案手法では，関数名を成す文字列の一部を汎化タグで置き換える仕組みと，階層的コピー機構により精度向上を実現している．

提案手法は Uri らが公開しているコーパス Java-small[*2]において，従来手法である code2seq に対し，F1 において 1.64 の性能向上を実現し，関数名の推定において未知語や低頻度語へ対処することの重要性を示した．

2. 関連研究

2.1 ソースコードを入力とした研究

近年，ソースコードの要素を予測する手法が発展している．Hindle ら [6] は，$n-1$ 個前までのトークンを元に次のトークンを予測する n-gram モデルを使用し，ソースコードの類似性や，入力ソースコードの次に出現するトークンの予測が可能なことを示した．

Raychev ら [9] は，“BigCode”と呼ばれる大規模なソースコードコーパスを使用して，ソースコードの要素を推測する手法を提案した．Javascript のソースコードの変数名を全てランダムなアルファベットに置き換えたものを入力として，それらを適切な変数名に変換するモデル JSNice[*3]を公開した．

Allamanis ら [2] は，短い文と一行程度のソースコードのペアを学習させる手法を提案した．この手法ではソースコードを構文木に変換し，あるノードとその親ノードの種類の出現回数を元としてソースコードをベクトル化している．また，自然言語をベクトル表現する際には BoW と呼ばれる，出現した単語の種類と回数を元にベクトル化する手法が使用されていた．

Allamanis ら [1] は，自然言語処理技術でなく，グラフベース技術でソースコードを解析する手法を提案した．この手法は，ソースコードを抽象構文木に変換することで，入力をグラフとして扱うことに成功している．抽象構文木（Abstract Syntactic Tree; AST）とはソースコードをコンパイラで解析する過程で，プログラムの中間表現として扱われる，解析に必要な情報のみを取り出した木構造のデータ構造である．

Uri ら [4] は，ソースコードを入力とするタスクにおいて，抽象構文木に変換してからさらに，その木が持つ全ての葉から別の葉への経路をベクトル化したものを入力とすることを提案した．また，これまで提案されたソースコードの要素を予測するタスクとそれに使用されるモデル，入力となるソースコードの言語の比較を行った．

2.2 ソースコードを入力として関数名を推定，生成する研究

Uri ら [5] が提案した code2vec モデルは，ソースコードを抽象構文木に変換した後に，エンコーダでその木が持つ全ての葉から別の葉への経路をベクトル化する．さらに，ソースコード中の変数名や型名を入力として使用して，関数名の予測を行なっている．

Xu ら [11] は階層的 attention という手法を提案した．これは，ソースコードをブロック単位で階層構造に見立てて，下の構造から再帰的ニューラルネットを使用して重み付き和を取り，その情報を上の構造での入力とすることで重要な情報のみが最終的に残るように工夫されている．Xu らのタスクは，ソースコードの関数を入力として，関数名を出力するものであり，階層構造の重み付き和を使用している為今回の提案手法と関連が深い．

Uri ら [3] が提案した code2seq モデルは，今回の提案の元となった論文であり，上記の code2vec モデルの出力部分に再帰的ニューラルネットワークを用いることで，関数名の生成を行なっている．モデルの詳細は次節で述べる．

3. code2seq モデル

提案手法の元となった code2seq モデル [3] について述べる．これは関数のソースコードを入力と

[*2] https://github.com/tech-srl/code2seq#datasets
[*3] http://jsnice.org/

ⓒ 2019 Information Processing Society of Japan

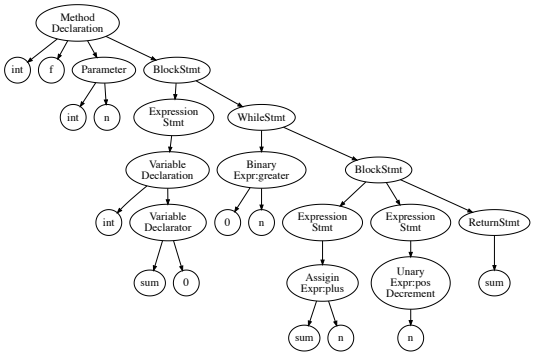

図 2　抽象構文木

```
int f(int n) {
    int sum = 0;
    while(n > 0){
        sum += n;
        n--;
    }
    return sum;
}
```

ソースコード 2　Java ソースコード

して関数名を生成するモデルである．この手法では，入力のソースコードを抽象構文木に変換したのち，エンコーダデコーダモデルで木構造から関数名を生成する．抽象構文木については以下で説明する．

3.1　抽象構文木

抽象構文木とはコンパイラによってソースコードを解析する際に得られる中間表現であり，解析に関係がある部分だけを抽出した木構造のデータ構造である．例えば，ソースコード 2 を入力とすると，図 2 のような抽象構文木が得られる．ここでは，JavaParser を使用して Java の抽象構文木を得ている．

3.2　code2seq モデル

本節では code2seq モデルの概要を紹介する．このモデルは時刻 t の出力をする際には図 3 のように動作する．

まず，ソースコードを抽象構文木に変換する．次に，葉から葉へ枝を辿ったときの抽象構文木の要素の系列を，「葉」→「内点の系列」→「葉」に分ける．この手法は，葉の系列と内点の系列を異なる方法でベクトル化する．

葉のベクトルは，その葉に対応する文字列を成すサブワードのベクトルの総和と定義する：

$$encode\_token(w) = \sum_{s \in split(w)} e_s^{subtokens} \quad (1)$$

ここで，split は変数名をサブワードに分割する操作であり，$e_s^{subtokens}$ はサブワード s を埋め込み層でベクトル化したものである．葉と葉を結ぶ内点の系列は双方向 LSTM をエンコーダとして用いてベクトルにしている．すなわち，まず葉と葉を結ぶ内点の系列は以下の LSTM の更新式により，内部状態の系列 $\overrightarrow{h}_1, \overrightarrow{h}_2, \ldots, \overrightarrow{h}_l$ に変換される：

$$f_t = \sigma(W_f[\overrightarrow{h}_{t-1}; x_t] + b_f) \quad (2)$$
$$j_t = \sigma(W_j[\overrightarrow{h}_{t-1}; x_t] + b_j) \quad (3)$$
$$\hat{c}_t = \tanh(W_c[\overrightarrow{h}_{t-1}; x_t] + b_c) \quad (4)$$
$$c_t = f_t \otimes c_{t-1} + j_t \otimes \hat{c}_t \quad (5)$$
$$o_t = \sigma(W_o[\overrightarrow{h}_{t-1}; x_t] + b_o) \quad (6)$$
$$\overrightarrow{h}_t = o_t \otimes \tanh(c_t). \quad (7)$$

逆方向の $\overleftarrow{h}_t$ についても同様に導出する．ここで，W_f，W_j，W_c，W_o はそれぞれ重み行列であり，b_f，b_j，b_c，b_o はバイアス項を表す．また，σ は活性化関数であり，$\otimes$ はアダマール積の演算子である．この内部状態の系列を用い，系列 $v_1, v_2, \ldots, v_l$ は以下のようにベクトル化される：

$$encode\_path(v_1 \cdots v_l) = [\overrightarrow{h}_l; \overleftarrow{h}_1]. \quad (8)$$

これらを用いて，構文木の葉と葉を結ぶ要素の系列は，内点系列のベクトルと，2 つの葉のベクトルを連結したものとしてベクトル化される：

$$z = tanh(W_{in}[encode\_path(v_1 \cdots v_l); encode\_token(v_0); encode\_token(v_{l+1})]). \quad (9)$$

ここで，W_{in} は 3 つのベクトルを結合したものを線形変換する行列であり，v_0 と v_{l+1} は系列の両端の葉である．

系列集合に対して LuongAttention 機構を用いて LSTM のデコーダで関数名を生成する．ここで，デコーダの初期入力 s_0 は，

$$s_0 = \frac{1}{k} \sum_{l \in L} \sum_{l' \in L \setminus \{l\}} y_{l,l'} \quad (10)$$

ⓒ 2019 Information Processing Society of Japan

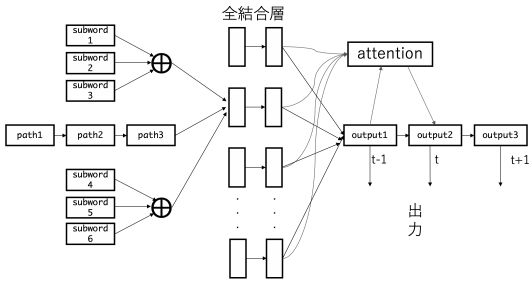

図 3 code2seq のモデル図

のように算出する．$k = min(|L| \times (|L| - 1), M)$ と定義する．M はハイパーパラメータであり，L は葉の要素集合である．y_{l_i,l_j} は，葉 l_i と葉 l_j を結ぶ経路をベクトル化したものである．

3.3 LuongAttention 機構

本節では，code2seq モデルで使用された手法の一つである LuongAttention 機構 [8] について説明する．これは，デコードしている時に入力のどの単語が重要かを重み付けするものである．デコーダに使用される時刻 t の LSTM の状態 s_t と，エンコーダの k 番目の出力 q_k を入力としている．$\alpha_t(k)$ は，

$$\alpha_t(k) = \frac{\exp(d_a^T \tanh(W_a[s_t; q_k]))}{\Sigma_{k'} \exp(d_a^T \tanh(W_a[s_t; q_{k'}]))} \quad (11)$$

と導出され，s 番目の入力情報に対する重要度を表している．最終的に $\Sigma_k \alpha_t(k) s_k$ を用いて出力を行うことになる．ここでの W_a は線形変換のための重み行列であり，d_a はモデルのパラメータのベクトルである．

4. 提案手法

本節では提案手法を説明する．基本的なモデルの構造は前節で述べた code2seq モデルに従っている．従来手法には，学習データに存在しない未知語が入力に含まれている場合適切に扱えず，また未知語や低頻度語を出力することができない，という問題点があった．そこで，それを解決するための方法として，以下で述べるように，最頻単語を汎化タグに置き換える方法と，階層的コピー機構を用いる方法を提案する．

4.1 最頻単語の置き換え

本方法では，コーパス中のある関数について，その関数内で最も多く出現するサブワードを汎化タグに置換する．

最頻単語はソースコードの重要な情報を持っており，学習データ全体の 31.26%の関数名は最頻単語を含んでいる．ソースコードの中に未知語が含まれる場合，従来手法であれば全ての未知語を単一のトークン UNK に置換してしまう．しかし，最頻単語を汎化タグに置き換えることによって未知語であってもその単語が重要であるということがわかり，その単語を含む系列が関数名を生成する上で重要であるということを捉えることができる．

4.2 階層的コピー機構

まず，提案手法に使用するコピー機構 [10] について説明する．コピー機構はエンコーダ・デコーダモデルを発展させたもので，出力する際に入力に使用された単語をそのままコピーして使用することができる．時刻 t に，単語 w を出力とする確率 $p(w)$ は以下のように導出される:

$$p(w) = p_{gen} p_{voc}(w) + (1 - p_{gen}) p_{pos}(w) \quad (12)$$

$$p_{gen} = softmax(h_{ctx}) \quad (13)$$

$$h_{ctx} = W_h l_t + W_s s_t + W_x x_t + W_c g_t \quad (14)$$

$$p_{pos}(w) = \sum_{i:w_i=w} a_i^t \quad (15)$$

$$g_t = \sum_{k=0}^{t-1} a^k \quad (16)$$

時刻 t のエンコーダによって入力の情報を付加したベクトルを l_t，デコーダの LSTM の隠れ状態を s_t，デコーダへの入力を x_t，入力の i 番目の情報の重みを a_i^t，a_i^t を i 番目の要素としてもつベクトルを a^t としている．また時刻 t に，デコーダの出力として単語 w が出現する確率を $p_{voc}(w)$，$p_{voc}(w)$ の出力を利用する確率を p_{gen}，単語 w を入力からコピーする確率を $p_{pos}(w)$ としている．また，W_h，W_s，W_x，W_c は線形変換のための重み行列である．

入力された文字列に含まれる単語をコピーして使用する確率が $1 - p_{gen}$ であり，この確率は前述の attention による重み付けの情報を元に算出される．また，$t-1$ までの重み情報の総和 g_t を入力に使う機構は coverage 機構と呼ばれ，過去にコピーした単語を使うのを抑える効果がある．

code2seq モデルの場合は入力がただの文字列ではなく，系列の配列であるため従来のコピー機構を使用することができない．本研究のエンコー

ダでエンコードする情報は，[葉のサブワードの埋め込みベクトルの和; 内点の系列を LSTM でエンコードしたベクトル; 葉のサブワードの埋め込みベクトルの和] のベクトルである．ここで，コピーする可能性があるのは葉のサブワードであるので，ある葉から別の葉への経路に含まれるサブワードは 1 つ目の葉に含まれるサブワードと 2 つ目の葉に含まれるサブワードである．そのため，まず葉から葉への経路全体の重み a_i と，ある経路の中に含まれる，あるサブワードの重み $b_{i,j}$ を加味する必要がある．そこで階層的コピー機構という手法を用いる．$b_{i,j}$ は b_i の j 番目の値である．ある入力に含まれている単語 w をコピーする確率 $p_{copy}(w)$ は，

$$p_{copy}(w) = \sum_i \sum_{j:w=w_j} a_i^t b_{i,j}^t \tag{17}$$

のように表される．a_i^t は従来の luong attention の重み付けをそのまま使用しており，b_i^t は，

$$g_t = \sum_{k=0}^{t-1} a^k \tag{18}$$

$$b_i^t = softmax((W_h' l_t + W_s' s_t + W_x' x_t + W_c' g_t)^T E_{subword,i}) \tag{19}$$

のように計算される．また，$E_{subword,i}$ は i 番目の系列に含まれる全てのサブワードを埋め込み層にて出力した結果であり，a^t は時刻 t の a_i の系列である．ここでの W_h'，W_s'，W_x'，W_c' は線形変換のための重み行列である．

最終的な単語 w の出現確率 $p(w)$ は式 (13) と式 (17) から，

$$p(w) = p_{gen} p_{voc}(w) + (1 - p_{gen}) p_{copy}(w) \tag{20}$$

のように表すことができる．階層的コピー機構を示したのが図 4 である．従来手法では学習データ中の関数名に含まれていない単語を出力することができなかった．しかし，この階層的コピー機構をデコーダに加えることで，訓練データに含まれていなくても，入力ソースコードに含まれている単語を関数名の出力に使用することができる．学習データの 71.42%は入力されたソースコードに含まれるサブワードのいずれかが関数名にも含まれているため，入力されたソースコードからそのままサブワードを使用することは有効であると考えられる．

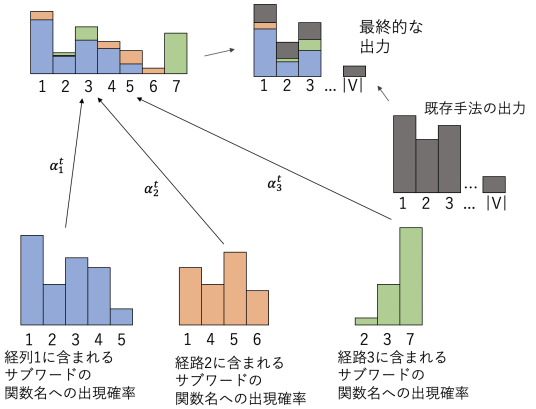

図 4　階層的コピー機構．$|V|$ は語彙サイズを表す．

5. 実験

実験には，code2seq の著者が公開しているコーパス Java-small[*4]を使用した．これは，学習用データとして 89,393 件，開発データとして 5,269 件，テスト用データとして 1,875 件，のソースコードからなるコーパスである．一つのソースコード中には複数の関数が含まれうるため，学習用データには 691,974 個，開発データには 23,844 個，テスト用データには 57,088 個の関数が含まれる．我々のここでの目的はソースコードから関数名を生成することである．

評価指標である精度，適合率，F1 について説明する．まず，あるソースコードに対して生成した関数名と，正解関数名の両方に含まれているサブワードの数を true positive，生成した関数名に含まれているが正解関数名に含まれていないサブワードの数を false positive，正解関数名に含まれているが生成した関数名に含まれていないサブワードの数を false negative とした．これらの値を用いて精度，適合率，F1 を計算する．

また分類正解率は，生成した関数名が正解関数名と完全に一致していた事例の数をテスト事例の数で割ったものである．実験は 3 回実行して，それらの試行のスコアの平均をとった．

比較として，3.2 節で紹介した code2seq モデルの論文で報告されているスコアと，著者が公開している実装を利用して再現実験した結果のスコアを使用した．最頻単語の置き換えは 4.1 節で紹介した手法であり，階層的コピー機構は 4.2 節で紹介した手法である．また，階層的コピー機構+最頻単語の置き換えは，最頻単語の置き換えと階層

[*4] https://github.com/tech-srl/code2seq#datasets

ⓒ 2019 Information Processing Society of Japan

```
protected void f(long cPtr, boolean
    cMemoryOwn) {
    if (!destroyed)
        destroy();
    super.reset(swigCPtr = cPtr,
        cMemoryOwn);
}
```

ソースコード 3　出力例 (reset)

```
private void f (
    Queue<?> q1, Queue<?> q2) {
    assertEquals(q1, q2);
    assertEquals(
    "Hash codes are not equal",
    q1.hashCode(), q2.hashCode());
}
```

ソースコード 4　出力例 (assertEqualsAndHash)

```
public void f () {
    Mesh.clearAllMeshes(app);
    Texture.clearAllTextures(app);
    Cubemap.clearAllCubemaps(app);
    ShaderProgram.
        clearAllShaderPrograms(app);
    FrameBuffer.clearAllFrameBuffers(
        app);

    logManagedCachesStatus();
}
```

ソースコード 5　出力例 (clearManagedCaches)

的コピー機構を併用する手法である．実験結果を表 1 に示す．各指標において最も高いスコアを太字表記にした．

Koehn [7] が提案した，出力対を対象とするブートストラップ法を用いて code2seq モデルと，階層的コピー機構と最頻単語の置き換えを併用したモデルの出力を 1 万回サンプリングしたところ，p 値が 0.001 を下回った．そのため，未知語や低頻度語を考慮した提案手法は既存手法と比較して有意差があると言える．

傾向として，最頻単語の置き換え及び，階層的コピー機構がそれぞれ効果的であることがわかった．それぞれ再現率が 2.61 ポイント，5.15 ポイント上昇しているのは，既存手法では取ってくることのできないような単語を出力に加えられていると考えられる．最終的な F1 もそれぞれ 1.64 ポイント，4.27 ポイント上昇しており，明らかに既存手法が出力できていなかった重要な情報を出力できるようになったということを示していると考えられる．

また，先に述べた最頻単語の置き換えと階層的コピー機構を併用して実験を行った．最頻単語はエンコーダとデコーダ両方に効果があり，コピー機構はデコーダでのみ作用するので併用することでさらなる精度の向上が見込まれる．このモデルは，F1 の上昇は大きくないものの分類正解率は階層的コピー機構単体のものと比較して 2.14 ポイント向上している．これは，階層的コピー機構はデコーダでしか機能しないのに対して最頻単語の置き換えはエンコーダ部分で重要な情報をエンコードするのに貢献するからだと考えられる．

出力例を紹介する．ソースコード 3 は，状態を初期化する関数である．正しい関数名が reset であるのに対し，出力も reset であった．reset のような頻出する関数名は出力できていた．ソースコード 4 は，2 つの文字列が等しいかどうか，及び 2 つの文字列のハッシュ値が等しいかどうかを判別する関数である．正しい関数名が assertEqualsAndHash であるのに対し，出力は asserHashCodeEqual であった．この出力は，2 つの文字数が等しいか判別していることが分かりにくいという問題点があった．ソースコード 5 は，扱っているデータを削除する関数である．正しい関数名が clearManagedCaches であるのに対し，出力は clearAll であった．ソースコードから読み取れる情報を考えると出力は妥当なものに思われる．

6. 結論と今後の課題

本論文ではソースコードを入力としてその関数名を出力する手法として，訓練データの語彙に含まれていない未知語や低頻度語を，入力ソースコード中から動的に抽出して出力に利用する方法を 2 つ提案した．結論として，最頻単語の置き換えと階層的コピー機構は，どちらも未知語や低頻度語を考慮することに貢献することがわかった．

今後の課題は大きく分けて 2 つある．1 つ目は，サブワードの重みづけ $b_{i,j}$ である．提案手法では訓練データのみを使用しているが，これには Glove などの学習済み埋め込みベクトルを利用で

ⓒ 2019 Information Processing Society of Japan

表 1　実験結果

手法	精度	再現率	F1	分類正解率
code2seq の報告値	50.64	37.40	43.02	-
code2seq の検証値	49.44	38.02	42.81	16.20
最頻単語の置き換え	49.14	40.63	44.45	17.32
階層的コピー機構	51.77	**43.17**	47.08	19.13
階層的コピー機構+最頻単語の置き換え	**53.11**	43.01	**47.52**	**21.27**

きる可能性があると考えられる．

2 つ目は，置き換える単語の選び方である．本論文の手法では，単純に最も多く出現した単語という選び方をしたが，これには問題がある．例えば，`indexOfTarget` という関数名を生成したい場合を考える．target は他の単語に置き換えやすい単語であるが，of を別の単語に置き換えて適切な関数名になる場合は考えにくい．そのため，置き換えやすい単語のみを選択して汎化タグに置換するべきである．また，本研究では汎化タグに置き換えたのは 1 単語だけだったが，置き換えやすい単語を選ぶ方法があるのであれば，複数の単語を異なる汎化タグで置き換えることで，より効果的に未知語に対応できる可能性がある．

参考文献

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of International Conference on Learning Representations*, 2018.

[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, Vol. 37 of *Proceedings of Machine Learning Research*, pp. 2123–2132, Lille, France, 2015. PMLR.

[3] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *Proceedings of International Conference on Learning Representations*, 2019.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In Jeffrey S. Foster and Dan Grossman, editors, *PLDI*, pp. 404–419. ACM, 2018.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. In *Proceedings ACM Program. Lang.*, Vol. 3, pp. 40:1–40:29, New York, NY, USA, January 2019. ACM.

[6] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[7] Philipp Koehn. Statistical significance tests for machine translation evaluation. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pp. 388–395, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[8] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[9] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *SIGPLAN Notices*, Vol. 50, No. 1, pp. 111–124, January 2015.

[10] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[11] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. Method name suggestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, pp. 10–21, New York, NY, USA, 2019. ACM.

ⓒ 2019 Information Processing Society of Japan