Vol.2019-SE-202 No.11
2019/7/13

コンテキストと構文の情報を用いたニューラルネットによる変数名予測

松田 浩幸[1,a)]　紫藤 佑介[2]　小林 靖明[1]　山本 章博[1]　宮本 篤志[3]　松村 忠幸[3]

概要：ソースコード内の変数の名前は，その変数の役割や機能を的確に表現したものを与えることが理想である．変数に適切な名前が与えられていなかった場合，コードの可読性は著しく低くなってしまう．そこで，本論文では，変数の直前と直後に出現するトークン列からなるコンテキストとソースコードの抽象構文木から得られるパスの情報を活用して，適切な変数名を予測する処方を提案する．提案手法をJavaScript のソースコードで構成されるデータセットを用いて実験したところ，既存手法よりも上回る精度で予測が可能であることを確認した．また，いくつかの例において，既存の手法では予測が困難な構文に依存して決定されるような変数名も正しく予測できることを確認した．

1. はじめに

ソースコード中の識別子は，コードの可読性を決定する最も重要な要素のひとつである [7], [15]．適切に名前が付けられた変数や関数は，それらがコードの中で何の役割を果たすのかを端的に説明するため，ドキュメントがない場合にもコードの解釈を助けることができる．しかしながら，適切な識別子を割り当てることは必ずしも自明な作業ではなく，ソフトウェア開発の規模が大きくなるにつれて，識別子の統一化などの問題から，非常に困難となる．また，デコンパイル (decompile) されたコードや難読化 (obfuscation) されたコードは，そもそも元のコードの識別子情報が失われているため，なんらかの理由でそれらのソースコードを読む場合には，理解することは非常に難しい．

このような背景から，適切な識別子を予測する研究には多くのニーズがあり，近年のニューラルネットの台頭とともに非常に盛んな研究分野のひとつとなっている．

まず，識別子名の予測のひとつとして，関数/メソッド名の予測がある．これが与えられたメソッドについて，処理に応じた適切な識別子を割り当てる問題である．この問題は，コードの分散表現を学習し，それに基づいて関数/メソッド名を予測する手法 [1], [2] や，関数/メソッドの内容を説明するドキュメントを生成する手法が提案されている．これは要約したい関数のソースコードを，自然言語に翻訳しながら要約をするという機械翻訳のタスクとして見ることができ，ニューラル機械翻訳の流行とともに，ニューラルネットを用いた要約手法が多く提案されている [10], [11], [23].

もうひとつの重要な識別子の予測問題としては，変数名の予測がある．関数/メソッド名の予測とは異なり，何がその変数名を決定する因子となりうるかが自明でないため，関数/メソッド名予測と同様の手法をそのまま利用することはできない．これらに対し，Bavishi ら [3] は変数が出現する周りのトークン列 (コンテキストと呼ぶ) に着目し，それらを Long Short-Term Memory (LSTM) [9] を用いて学習させる手法を提案した．ここで，LSTM は系列を学習するために提案されたニューラルネットワークの一種である．ソースコードの自然さの仮説 [8][*1]に基づくと，自然言語で頻繁に用いられているようなコンテキストを活用することは，変数を予測するというタスクにおいても有効であると期待できる．実際に，彼らの実験においても，既存の JavaScript 向けの脱難読化システム [17], [22] にわずかに及ばないものの，変数名予測においてニューラルネットの可能性が示されている．

一方で，コンテキストをのみでは学習が困難なタイプの変数名もまだ残されている．例えば，`for` ループについて考えてみると，一般的にループカウンタとして `i` という変数を用いることが多いが，その `for` ループにネストした `for` ループの場合には `j` を変数として用いられる (ソースコード 1)．このようなコード片では，変数 `j` をコンテキストの情報からネストしたループカウンタの変数であること

[1] 京都大学情報学研究科
[2] 株式会社メルカリ
[3] （株）日立製作所基礎研究センタ
[a)] matsuda-hiroyuki@iip.ist.i.kyoto-u.ac.jp

[*1] 人間が記述するソースコードは，自然言語を用いた文章などと同様に，統計的モデルを用いて予測が可能とする仮説．

を予測できるかどうかは必ずしも明らかではない．

ソースコード 1　ネストしたループ．

```
for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
        ...
    }
}
```

そこで本研究では，変数のコンテキストからだけでなく，その変数の出現した位置がどのような構文に属しているかの情報を**抽象構文木のパス**の情報からも学習し，それらを組み合わせることで変数名を予測する手法を新たに提案する．具体的には，抽象構文木 (AST) においては，変数は葉として出現するため，その葉から根に向かって定義されるパスの情報を構文の系列情報として，LSTM を用いて学習する．このパスの系列情報においては，AST 内の内部頂点がその変数の出現する位置の構文情報を表現しているため，例えば「変数 j が 2 重ループの中に出現している」といった情報も取得することができる．この構文の系列情報も活用して変数名を予測するニューラルネットワークモデルを提案し，既存手法である Context2Name [3] との比較実験を行った．JavaScript のコード群からなるデータセット [16] を用いた実験の結果，Context2Name を上回る精度で変数名の予測を行うことができた．また，コンテキストのみからでは予測が困難であると思われる変数に関しても，正しく予測ができる例を観測することができた．

本稿の構成は以下の通りである．第 2 節において，本研究と関連のあるソフトウェアにおける識別子予測に関する研究について述べる．第 3 節では，提案手法のアイデアの元となった研究である Context2Name について紹介する．その次に，提案手法の説明を行い，第 5 節において Context2Name と提案手法の比較のための計算機実験について述べる．

2. ソフトウェアにおける識別子予測の背景

プログラムの難読化は，脆弱性への攻撃や剽窃などから守るために行われる処理である．このような目的のために施される難読化は，完全に攻撃などから守ることは困難であるが，実用の観点においては十分な場合も多く，数多くのケースで用いられている．特に，JavaScript においては，プログラムがクライアント側で実行されることが多いため，コードがクライアント側で実行可能かつ人間にとって理解が困難になるようにする目的で使用されている．また，帯域幅を不用意に使わないように，コードをできるだけ短くしてクライアント側へ送信するという目的のためにも難読化は利用されている．

一方で，コードの脱難読化は，リバースエンジニアリングやマルウェア分析などを目的として行われる．難読化がされたコードは，脆弱性への攻撃から守る助けになる反面，不正に書き換えられたコードをオリジナルの著者が判別できなくなるという問題点 [5], [18], [19], [20], [21] もある．

これらの脱難読化においては，尤もらしい識別子名を復元することが，コードの可読性の観点から重要となるため，変数名の予測に関する研究が取り組まれている [12], [17], [22]．また，同様の理由から，機械語からデコンパイルをしたコードに対して変数名を予測するタスクも取り組まれている [12], [13]．

変数名の予測に関する代表的な研究として，JSNice [17], JSNaughty [22], そして本研究との比較を行う Context2Name [3] が知られている．Context2Name については次節で説明する．

JSNice [17] は JavaScript の識別子名や変数の型注釈を予測するシステムである．この手法では，変数の依存関係を条件付き確率場 (conditional random field) としてグラフカルモデルとして表現する．JSNice では，変数の意味的な依存関係などを細かに分析しながら確率モデルとして定式化するため，高い精度で変数名の予測が可能である一方，プログラム言語依存の知識が問われるため，別の言語へ予測アルゴリズムを転用するのが容易ではない．

JSNaughty は，Vasilescu ら [22] が提案する統計的機械翻訳をベースとした Autonym と呼ばれるシステムと JSNice を並行して利用することで変数名の予測を行う．Autonym では，難読化されたコードを翻訳元の言語の文とみなし，脱難読化されたコードを翻訳先の言語 e の文とみなすことで，機械翻訳のアルゴリズムを適用して予測を行う．翻訳には，Moses [14] と呼ばれるオープンソースの機械翻訳のシステムを利用している．JSNaughty も高い精度で変数名の予測が可能である一方で，学習後のモデルに予測させたときの実行時間が非常に大きいという問題点がある．

そこで Bavishi らは，様々なプログラミング言語にも適用可能であり，さらに高速に予測のできる学習モデルとして Context2Name を提案した [3]．

3. 既存手法 (Context2Name)

本節では，機械学習を用いた変数名予測の既存研究として，Bavishi ら [3] によって提案された Context2Name について説明する．Context2Name は，変数の前後に出現するトークンの系列 (コンテキスト) と，ある変数に対してどのようなコンテキストが出現するかの系列を学習することで適切な変数名を予測する手法で，提案手法においてもベースとなる手法である．

3.1 コンテキストの取得

まず，対象となるコード片をトークン列に変換し，それらから丸括弧とドットを除いて出現する順番に $t_1, t_2, ...$

とする．その中からローカル変数を抽出する[*2]．ある変数 v において $t_k = v$ であるとき，前後 q 個ずつのトークン $c : t_{k-q}, t_{k-q+1}, ..., t_{k-1}, t_{k+1}, ..., t_{k+q}$ を抽出し，これを v の**コンテキスト**として定義する．このとき，他の変数は変数という情報だけを持つように特別な ID トークンとして扱い，ある変数名を予測するためにこの他の変数は利用しない．これは，コード片を一度に脱難読化をする上で必要な仮定である．前後のトークンを取り出すとき，q 個の範囲がコードの範囲外になってしまう場合は，超えてしまった部分には特別な PAD トークンを代わりに挿入し，この PAD トークンも同時にコンテキストに含める．

同じ変数は同じスコープ内で複数回出現するため，一つの変数から複数のコンテキストが得られることとなる．複数回の出現の中で，最初から l 個目までのコンテキストを $c_1, c_2, ..., c_l$ とし，$l+1$ 回以上その変数が出現している場合には，$l+1$ 回目以降のコンテキストは全て切り捨てることとする．逆に，同じスコープ内での変数の出現回数が l を下回る場合には，下回った分だけ $2q$ 個の PAD トークンの系列を挿入することで，要素数を揃える．

最後に，取得したコンテキストは，後段のニューラルネットワークに用いるために，one-hot ベクトル表現に変換される．ここで変換される one-hot ベクトルとは，ある決められた範囲の語彙数の元でトークンと一致する成分のみが 1，それ以外が 0 となるような次元を語彙数とするベクトルのことである．

3.2 系列オートエンコーダによる分散表現の獲得

この節では，各変数のコンテキストからその分散表現を系列オートエンコーダ [4] によって獲得する過程について簡単に説明する．

3.1 節で得られたコンテキスト中のトークンの one-hot ベクトルは，要素のほとんどが 0 で次元数が非常に大きいベクトルである．このようなベクトルや行列をニューラルネットワークの入力にすると，パラメータが不必要に多くなり計算時間が大きくなったり，過学習が起こる可能性がある．そこで，そのベクトルの “特徴” を維持しつつ，低次元に射影したベクトル表現を得ることが必要になる．Context2Name ではオートエンコーダの一つであり，系列データに対して有効な**系列オートエンコーダ** (sequence autoencoder)[4] を活用している．系列オートエンコーダの概略を図 1 に示す．オートエンコーダは，次元を圧縮させたいベクトルの入力に対して，出力と入力が同じ値のベクトルに近づくように学習させる半教師ありニューラルネットワークモデルの 1 つである．できるだけ元の情報を保持しつつ，入力の次元数を圧縮したベクトルを隠れ層から得ることができる．

[*2] JavaScript の難読化においては，グローバル変数は難読化されないため，ここではローカル変数のみを対象とする．

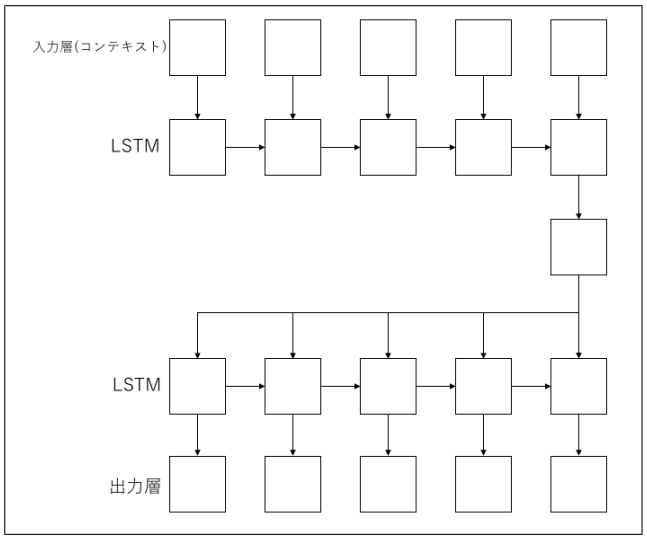

図 1 系列オートエンコーダの概略.

Context2Name では，トークンの one-hot ベクトルの系列が入力となっているため，系列データを圧縮することができる扱えるように拡張した系列オートエンコーダを用いる． 系列オートエンコーダはエンコーダとデコーダと呼ばれる部分に分かれている．エンコーダとデコーダは，それぞれ LSTM により構成されている．エンコーダは入力のベクトルの系列を LSTM の最終時刻の出力により低次元の実数値ベクトルへと変換し，一方で，デコーダではエンコーダの出力する実数値ベクトルから元の大きさの系列へと同じく LSTM により復元する．そして，エンコーダによって変換されたベクトルをデコーダによって復元させたときに，入力と出力の間のクロスエントロピー誤差が小さくなるように学習する．

Context2Name では，前述のようにこの系列オートエンコーダの入力と出力にコンテキストの one-hot ベクトルの系列を利用する．入力と出力には変数の前後 q 個ずつの one-hot ベクトルを用いる．学習の結果得られたエンコーダを使って，前後のトークン列の one-hot ベクトルを次元数 E のベクトル ce へと圧縮し，圧縮されたコンテキストの系列 $ce_1, ce_2, ..., ce_l$ を後段のニューラルネットワークへの入力として使用する．

3.3 変数名予測

最後に，前節で求めたコンテキストを用いて変数名を予測する．変数名予測のためのニューラルネットワークの概略を図 2 に示す．前節で求めたコンテキストのベクトルは，各変数について l 個からなるベクトルの系列である．変数名を予測するためのニューラルネットワークでは，最初にコンテキストの系列 $ce_1, ce_2, ...ce_l$ を LSTM に入力して次元数 H のベクトルへと変換してから，出力層への入力とする．出力層は，ベクトルの各次元が各変数名に対応するようなベクトルとする．そして，出力のベクトルの次元の

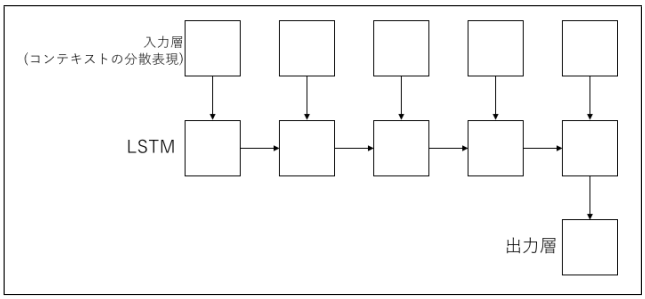

図 2 変数名予測のためのニューラルネットの概略.

中で最も大きい値となった次元に対応する変数名を，このニューラルネットが適切だと予測した変数名とする．

4. 提案手法

前節で説明した Context2Name は，変数のトークンの周辺から得られるコンテキストの情報のみを活用した手法であったが，本論文では，Context2Name のコンテキストに加えて，構文から得られる情報（具体的には抽象構文木から得られるパス方向の情報）も活用した新たな変数名予測の手法を提案する．提案手法は，主にコンテキストの抽出，抽象構文木のパスの取得，それらを組合わせたニューラルネットワークモデルといった 3 つのパートからなる．以下では，それぞれについて提案手法の詳細を説明する．

4.1 コンテキストの抽出

提案手法においても，Context2Name と同様の方法で変数の前後 q 個ずつのトークン列を抽出し，変数名予測のための情報として用いる．Context2Name では前後のトークン列を一連の系列とみなし，その分散表現を学習に利用していたが，この分散表現では系列の特徴を表現してはいるものの，どの位置で変数が使われているのかという情報がエンコードなどを経て失われてしまう可能性がある．そこで，提案手法では，変数の前のトークン列と後ろのトークン列を別の情報と考え，前後 q 個ずつのトークン列 $c_b : t_{k-q}, t_{k-q+1}, ..., t_{k-1}$ と $c_a : t_{k+1}, ..., t_{k+q-1}, t_{k+q}$ の 2 つを分離して変数名予測に利用することとした．このようにすることで，変数は前のトークン列の次に出現し，その後に後ろのトークン列が続くということを明確に表すことができるようになる．例えば，この操作をソースコード 2 中の 1 行目に出現している変数 `array` に対して行うと，以下のようになる．

- 変数 `array` の前のコンテキスト: {“PAD”, “PAD”, “PAD”, “`function`”, “`sort`”}
- 変数 `array` の後のコンテキスト: {“{”, “`for`”, “`let`”, “ID”, “`=`”}

同じ変数は同じスコープ内で複数回出現するため，1 つの変数に対して複数のコンテキスト c_b と c_a が得られる．複数回の出現の中で，最初から l 個目までのコンテキスト

ソースコード 2 バブルソートの例.

```
1  function sort(array){
2      for(let i = 0; i < array.length; i++){
3          for(let j = array.length - 1; j > i ; j
               --){
4              if(array[j] < array[j - 1]){
5                  let tmp = array[j];
6                  array[j] = array[j - 1];
7                  array[j - 1] = tmp;
8              }
9          }
10     }
11     return array;
12 }
```

C_b,C_a を変数名の予測に用いる．このとき，Context2Name と同様に，PAD トークンの挿入などによって要素数を揃える．

最後に，後述のニューラルネットワークの入力として用いるめに，それぞれをベクトルへと変換する．そのために，取得したコンテキスト中のトークンを全て固有の番号に変換する．あらかじめ各トークンと一対一対応の固有の番号の組み合わせの辞書を作成しておく．辞書は，全コンテキスト中のトークンを頻度順で上から V_{inp} 個分だけ作成し，辞書から漏れてしまった出現数の少ないトークンは特別な UNK トークンとして扱う．ID トークン，UNK トークン，PAD トークンにもそれぞれ固有の番号を用意しておく．そして，各トークンをこの辞書を用いて固有の番号へと変換する．これにより，各コンテキストは固有の番号が要素となるようなベクトルとなる．

以上の操作を訓練データの全ソースコードの，全変数に対して実施することとする．

4.2 抽象構文木のパスの取得

変数名を予測するにあたって，変数がどのような構文構造の中で出現しているのかという情報を利用することを考える．そのために，抽象構文木のパスを取得する必要性とその手順について説明する．例として，ソースコード 2 中の 3 行目の変数 `j` の中で 1 番目に出現している変数について考える．まず，JavaScript の構文解析器である Esprima [6] を使用して構文解析を行い，その抽象構文木をグラフで可視化すると，以下の図 3 のような抽象構文木が得られる．

各ノードは JavaScript の文と式の型に対応しており，葉ノードは各トークンに対応している．赤い枠に囲まれているものが変数 `j` のノードであり，この `j` は初めて宣言されたものであるため，VariableDeclarator の子ノードとなっている．VariableDeclarator のノードから上方向へとパスを辿っていくと，VariableDeclarator，VariableDeclaration，ForStatement となっており，`for` 文の中に `j` が存在していることが分かる．また，さらに上へと辿っていくと，

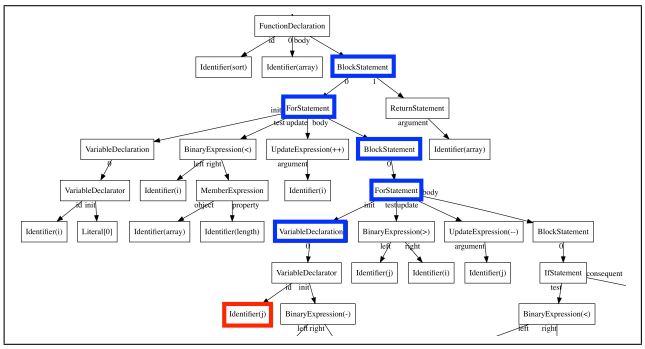

図 3　ソースコード 2 の抽象構文木の一部.

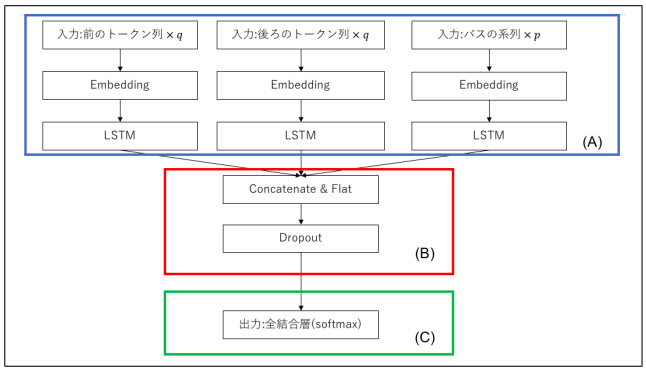

図 4　提案手法のニューラルネットワーク概観.

BlockStatement，ForStatement となっており，`for` 文の中に `for` 文が存在していることが分かる．このように，抽象構文木のパスを利用すると，どのような構文構造の中で出現しているのかが分かる．そのため，葉から根方向へのパスの情報を活用することは，`j` のような変数名を推測する上での重要な要素になると考えられる．

次に，提案手法での抽象構文木のパスの取得手順について説明する．ローカル変数それぞれについて，その変数のノードから抽象構文木のパスを上へと辿っていき，p 個まで文と式の情報を取得する．このとき，VariableDeclarator に関しては VariableDeclaration の子ノードとして必ず出現するため，VariableDeclarator の情報は必要がないと判断し，無視することとした．同様の理由で SwitchCase と Property も無視している．また，MemberExpression に関しては，配列の要素を取得する式とそれ以外では，変数の使われ方がかなり異なると判断しその 2 つを別の式として扱った．

コンテキストと同様に，パスの情報も，後述のニューラルネットワークの入力として用いるめに，それぞれをベクトルに変換する必要がある．まず，訓練データから取得できた全種類の文と式の型のそれぞれと番号を対応させた辞書を作る．そして，各変数で取得した p 個の文と式を番号に変換させる．生成できた p 個の番号は系列としてニューラルネットワークの入力として用いる．

4.3　ニューラルネットのモデル

本節では，前後のトークン列と抽象構文木のパスを学習して，実際に変数名を予測するためのニューラルネットワークの構成について述べる．今回提案手法で用いるネットワークの概形図を図 4 に示す．(A)，(B)，(C) のそれぞれの部分を以下に説明する．

(A) まず，ネットワークの入力は 3 つある．1 つ目と 2 つ目はそれぞれ予測する変数の前のトークン列と後ろのトークン列を辞書により番号に変換したものに対応し，それぞれ次元数 q のベクトルである．前後のトークン列のベクトルはそれぞれ Embedding という層で，番号から次元数 e_1 の密なベクトルへと変換される．次に，トークン列の分散表現を得るために，Embedding 層で得られた密なベクトルの系列を LSTM へと入力し，次元数 e_2 のベクトルへと変換する．3 つ目の入力は抽象構文木のパスを変数のノードから辿っていったときの p 個のノードに対応した番号によるベクトルである．トークン列と同様に，Embedding で系列を ID から次元数 e_1 の密なベクトルへと変換させ，LSTM によって次元数 e_2 のベクトルへと変換する．

(B) A で得られた 3 つのベクトルを Concatenate 層で横並びに連結させることで，1 つの変数の前後のトークン情報と抽象構文木のパスの情報を 1 つにまとめた次元数 $3e_2$ のベクトル C が得られる．同一スコープの同じ変数は複数回出現するため，その回数分のベクトル C が l 個得られることとなる．これを $C_1, C_2, ..., C_l$ とする．$C_1, C_2, ..., C_l$ を横並びにした次元数 $3le_2$ のベクトルを Dropout 層 (ドロップアウト率は 0.1) へとつなげる．Dropout 層は過学習を防ぐために挿入される．

(C) 出力層はベクトルの各次元が各変数名の番号に対応するようなベクトルとする．そのために，あらかじめ各変数名と一対一対応の固有の番号の組み合わせの辞書を作成しておく必要がある．辞書は，訓練データの正解の変数名を頻度順で上から V_{inp} 個分だけ作成する．出力層の活性化関数には softmax 関数を利用する．softmax 関数の性質により，各次元の値は対応した変数名が元々の変数名である確率とみなすことができる．確率が最大の次元に対応した変数名を，予測した変数名とする．

5.　計算機実験

5.1　データセット

データセットには Context2Name でも使用されている 150k JavaScript Dataset [16] を用いた．これは，約 15 万個の JavaScript のソースコードが集められたデータセットであり，機械学習のデータとして用いるためにあらかじめ約 10 万個の訓練データと約 5 万個の評価データに分けられている．Context2Name と同条件の実験によって正解率を評価するために，それぞれのデータの中で重複しているデータや，訓練データと評価データの互いの間で重複しているデータ，131,072 文字以上の大きすぎるファイルなどは

除去した．この結果訓練データと評価データのソースコードはそれぞれ 64,750 個と 33,229 個となった．検証を行うために評価データの 20%を分離し，これを検証データとした．訓練データと検証データ，評価データから変数を抽出した結果，それぞれ 2,551,065 個，255,591 個，1,022,366 個の変数を抽出することができた．これらの変数を用いて学習や検証，評価を行った．

5.2 実験環境

実験環境は次の通りである．

- CPU : Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz
- OS : Ubuntu 16.04 LTS
- メモリ : 64GB
- GPU : GeForce GTX 1080Ti

5.3 Context2Name の実験と結果

Context2Name を Python 3.6.5，Keras 2.2.4 によって実装し，5.1 節で示したデータセットを用いて実験を行った．パラメータに関しては，Context2Name の論文にある数値と同じ値にした．出力の語彙数は 60,000 としており，この語彙の中に当てはまった評価データの変数名は，1,022,366 個のうち 897,506 個であった．出力の語彙に当てはまらないデータも含めた全体の正解率は約 49.0%だった．

5.4 提案手法の実験と結果

提案手法を Context2Name と同様に Python 3.6.5，Keras 2.2.4 によって実装し，5.1 節で示したデータセットを用いて実験を行った．訓練データから文と式の情報を抽出したところ，VariableDeclarator と SwitchCase，Property を除いて以下の表 1 と表 2 の 28 種類の情報が抽出できた．

表 1 式の種類

ArrayExpression	ObjectExpression	SequenceExpression
FunctionExpression	MemberExpression	CallExpression
NewExpression	UpdateExpression	UnaryExpression
BinaryExpression	LogicalExpression	ConditionalExpression
AssignmentExpression		

表 2 文の種類

Do-WhileStatement	ExpressionStatement	ForStatement
ForInStatement	FunctionDeclaration	IfStatement
LabeledStatement	ReturnStatement	SwitchStatement
WhileStatement	ThrowStatement	TryStatement
CatchClause	VariableDeclaration	BlockStatement

パラメータに関しては，表 3 の通りにし，入力と出力の語彙数は Context2Name と同じ値に設定した．

また，ニューラルネットワークの損失関数はクロスエントロピー誤差とし，最適化アルゴリズムには Adam を使用した．学習にはバリデーションデータ用いてバリデーションを行い，正解率が最も高くなったエポック時点でのモデルを最良のモデルとした．

表 3 ハイパーパラメータの値

q	p	l	V_{inp}	V_{out}	e_1	e_2
5	5	5	4096	60000	128	128

このような条件の下で実験を行なった結果，出力の語彙に当てはまらないデータも含めた全体の正解率は約 50.9% だった．

5.5 変数名の予測例

図 2 のソースコードの 2 つの変数 `i`，`j` に対して実際に復元を行なった．学習後の Context2Name と提案手法のモデルを用いて変数名を予測した結果，以下の表 4 のようになった．

表 4 Context2Name と提案手法での変数名の予測.

正解の変数名	Context2Name	提案手法
`i`	`i`	`i`
`j`	`m`	`j`

提案手法と Context2Name の両方で，一つ目の `for` ループの変数である `i` に関しては正しく `i` と予測できた．しかし，`j` に関しては，Context2Name では正しく予測することはできなかったが，提案手法では抽象構文木のパスを利用した意図通り，`j` へと正しく復元することに成功した．

他にもいくつかの例で，文法情報が予測の助けになっているであろう例が観測できた．

ソースコード 3 のような配列の要素を調べるような手続きを含むコード片では，Context2Name では，`key` 変数を `i` や全く関係のない変数名を予測したが，提案手法では正しく `key` と予測した．これは，`key` という変数が単純に配列のインデックスで使用されただけでなく，メソッドの引数として出現したという情報が予測に対して重要な因子となる．提案手法では，`key` は FunctionExpression のパラメータとして位置するため，このような情報を陽に利用できたのだと考えられる．また，コンテキストだけでは，配列アクセスを行っている行が `for` ループ内として出現していないという情報が考慮できていないが，本手法では，抽象構文木のパス上に ForStatement が存在しないことから考慮できたのだと考えられる．

その他の例として，JavaScript の Web フレームワークとして普及している Express[*3]では，ソースコード 4 のような記述が頻繁に用いられる．

テストデータにおいては，この例と同様のパターンが数多く出現しているが，Context2Name ではうまく予想ができず，提案手法では予測ができているパターンが観測でき

[*3] `https://expressjs.com`

ソースコード 3　メソッド内で配列の要素のチェックを含むコード片.

```
function(array, key) {
    ...
    if (key === "xxx" || array[key] == null) {
      return;
    }
    ...
```

ソースコード 4　Express でのミドルウェア関数の例.

```
app.use(function (req, res, next) {
  console.log('...');
  next();
});
```

た．Context2Name では，精度向上のため丸括弧が取り除かれていることと，変数名が全て ID に置き換わっていることを考慮すると，next はコンテキストから推測するのが困難である．しかしながら，提案手法では，next には CallExpression であるという文法情報が付加されているため，予測が可能であったと考えられる．

これらのような結果から，変数を復元する上で抽象構文木のパスの情報は一定の効果を発揮していることが推測される．

6. おわりに

本研究では，ニューラルネットワークを用いたプログラム中の変数名の予測について取り組んだ．本手法では，Bavish ら [3] らの Context2Name をベースとして，それらの抽象構文木から得られるパスの情報を取り込むようにニューらネットを学習させた．その結果，予測精度についてはわずかに向上することが確認でき，さらには，コンテキストの情報からでは予測が難しいと思われる，構文に依存した変数名の予測が可能になった例をいくつか観測することができた．

今後の課題としては，変数名を予測するためには，コンテキストと変数出現する構文情報以外の情報で変数名を決定しうるものを考慮した学習が考えられる．特に，変数の依存関係などをグラフとして表現し，それらを学習に用いることは有用であると考えられる，また，変数の型情報も変数名の予測において重要であると予想される．JavaScript ではプログラム上で型情報が得られるとは限らないため，型注釈を自動的に付与するようなシステムを用いて，予め型情報を補足するという方法も有用であると予想できる．

参考文献

[1] M. Allamanis, H. Peng, C. Sutton: A convolutional attention network for extreme summarization of source code. In Proceedings of ICML '16, pp. 2091–2100 (2016)
[2] U. Alon, M. Zilberstein, O. Levy, E. Yahav: code2vec: learning distributed representations of code. In Proceedings of POPL '19, pp. 40:1–40:29 (2019)
[3] R. Bavishi, M. Pradel, K. Sen: Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. arXiv:1809.05193 (2018)
[4] A. M. Dai, Q. V. Le: Semi-supervised sequence learning. In Proceedings of NIPS '15, pp. 3079–3087 (2015)
[5] L. Durfina, J. Kroustek, P. Zemek: PsybOt malware: A step-by-step decompilation case study. In Proceedings of WCRE '13, pp. 449–456 (2013)
[6] Esprima. http://esprima.org/ (2019 年 6 月 4 日)
[7] E. M. Gellenbeck, C. R. Cook: An investigation of procedure and variable names as beacons during program comprehension. Technical Report, Oregon State University (1991)
[8] A. Hindle, E. T. Barr, M. Gabel, Z. Su, P. Devanbu: On the naturalness of software. Communications of the ACM 59(5), 122–131 (2016)
[9] S. Hochreiter, J. Schmidhuber: Long short-term memory. Neural Computation 9(8), 1735–1780 (1997)
[10] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin: Deep code comment generation. In Proceedings of ICPC '18, pp. 200–210 (2018)
[11] S. Iyer, I. Konstas, A. Cheung, L. Zettelemoyer: Summarizing source code using a neural attention model. In Proceedings of ACL '16, pp. 2073–2083 (2016)
[12] A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, B. Vasilescu: Meaningful variable names for decompiled code: a machine translation approach. In Proceedings of ICPC '18, pp. 20–30 (2018)
[13] D. S. Katz, J. Ruchti, E. Schulte: Using recurrent neural networks for decompilation. In Proceedings of SANER '18, pp. 346–356 (2018)
[14] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, E. Herbst: Moses: Open Source Toolkit for Statistical Machine Translation. In Proceedings of ACL '07, pp. 177-180 (2007)
[15] D. Lawrie, C. Morrell, H. Feild, D. Binkley: What's in a name? A study of identifiers. In Proceedings of ICPC '06, pp. 3–12 (2006)
[16] V. Raychev, P. Bielik, M. Vechev, A. Krause: Learning Programs from Noisy Data. In: Proceedings of POPL '16, pp. 761–774 (2016)
[17] V. Raychev, M. Vechev, A. Krayse: Predicting Program Properties from "Big Code". In Proceedings of POPL '15, pp. 111–123 (2015)
[18] E. J. Schwartz, J. Lee, M. Woo, D. Brumley: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In Proceedings of USENIXSEC '13, pp. 353–368 (2013)
[19] K. Yakidan, S. Dechand, E. Gerhards-Padilla, M. Smith: Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In Proceedings of SP '16, pp. 158–177 (2016)
[20] K. Yakidan, S. Eschweiler, E. Gerhards-Padilla, M. Smith: No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformation. In Proceedings of NDSS '15 (2015)
[21] M. J. van Emmerik: Static single assignment for decompilation. Ph.D. dissertation, University of Queensland (2007)
[22] B. Vasilescu, C. Casalnuovo, P. Devanbu: Recovering

Clear, Natural Identifiers from Obfuscated JS Names. In Proceedings of ESEC/FSE '17, pp. 683–693 (2017)

[23] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, P. S. Yu: Improving automatic source code summarization via deep reinforcement learning. In Proceedings of ASE '18, pp. 397–407 (2018)