東京大学
情報理工学系研究科 創造情報学専攻
修士論文

プログラミング言語の文法獲得に向けた Skip-Gram モデルによる非終端記号の獲得

Acquisition of Nonterminal Symbols using Skip-Gram Model for Inferring the Grammar of a Programming Language

松永 智將
Tomomasa Matsunaga

指導教員　千葉 滋 教授

2019 年 1 月

概要

本研究では、Skip-gram モデルによるトークン列のベクトル表現を用いて非終端記号を獲得する手法を提案する。提案手法では、トークン列をそのベクトル表現を用いてクラスタリングすることで非終端記号を得る。ソフトウェア解析では、解析に利用できるデータが充実してきたことから、既存のデータから解析対象の構造を獲得して解析を行うようになってきている。そこで、本研究ではソースコードの持つ構造を分布仮説に基づいた文法として獲得することを目指す。文法を獲得するための非終端記号の決定にトークン列のクラスタリングを応用する。トークン列のベクトル表現を得るために、左右の文脈を区別した Skip-gram モデルを使用する。また、小規模な文法からランダムに生成したデータセットを用いて本手法の検証を行なった。既存のソースコード群から新たな構造を文法として獲得することで、コーディングスタイル検査器やログ解析などへの応用が期待できる。

Abstract

In this paper, we suggest a method for acquiring nonterminal symbols with vector representations of tokens. The vector representations are generated by using the skip-gram model. We regard nonterminal symbols as a cluster by using the vector representations. In software analysis, structures of analysis object were obtained from existing data because we have enough data for learning. In our research, we try to acquire a grammar on the basis of the distributional semantics as a structure from source code. We use the modified skip-gram model which distinguish left side context and right side context for obtaining vector representations of tokens. We obtained a grammar from source code generated from a simple grammar. Our approach can be applied to a coding style checker, log analysis tool and so on.

目次

第 1 章

はじめに

1.1　プログラミング言語の文法推定に向けた非終端記号の獲得

本論文ではプログラミング言語の文法をソースコード群から獲得することを目指し、そのために必要となる非終端記号の獲得手法を提案する。ソフトウェア開発の現場では、ソースコードの大規模化や複雑化が大きな問題となる。そのため、開発者はソフトウェアの静的解析器等を用いることでソースコードの品質の担保や開発効率の向上を試みている。これらの解析ツールの需要に伴って、ソースコードのバグ検出や型情報の提案など、ソフトウェア開発の質の向上を目指した多くのタスクが研究として取り組まれている。ソースコードの解析では、ソースコードが持つ構造をどのように獲得するかが課題となる。本研究では、このような構造を文法として獲得することを目指す。プログラミング言語の文法獲得は、ソースコードの解析を行うような多くのアプリケーションに応用が期待できる。

ソースコードの解析ツールの例として、コーディングスタイル検査器がある。コーディングスタイルは、ソースコードの可読性を保証するために定められるソースコードの記述方法を示したものである。ソフトウェア開発者は、このコーディングスタイルに従ってソースコードを記述することが望ましい。コーディングスタイルは多く数多く存在し、プログラミング言語ごとにコミュニティや言語開発者によってコーディングスタイルガイドラインが公開されている。また、開発チームや企業毎に独自のコーディングスタイルを設けることもある。ソースコードがコーディングスタイルに従っていることを人力で確認するのはコストがかかり、見落としも起こるため、コーディングスタイル検査器が利用されている。独自のコーディングスタイルを定める場合、ユーザはコーディングスタイル検査器の設定ファイルを記述することで検査するスタイルを変更する必要がある。また、コーディングスタイル検査器の開発者はプログラミング言語やコーディングスタイルごとに検査器を実装する必要がある。このような問題を解決するために、既存のソースコードからコーディングスタイルを獲得する手法が必要であると考えられる。コーディングスタイルに対応する構造を文法として獲得することができれば、プログラミング言語の文法獲得はコーディングスタイル検査器にも応用可能であるといえる。

近年では、GPU 性能の向上に伴って深層学習をはじめとする機械学習手法を用いた研究・開発が活発になっている。機械学習の手法は自然言語処理分野や画像処理分野などの幅広い分野

で利用されており、ソフトウェア開発者を支援するツールやソフトウェア解析ツールにも応用され始めている。これらのツールで機械学習が利用されている理由の一つとして、データセットとして活用できるソースコードの増加が考えられる。例として、GitHub[*1]や SourceForge[*2]などのプラットフォーム上に公開されるオープンソースソフトウェアのソースコードが学習に利用可能である。また、Stack Overflow[*3]などのソフトウェアエンジニア向けの Web サービスにより、自然言語の質問とコードスニペットの組をデータとして扱うことも可能になっている。同様に、プログラミング言語の文法推定においても、多くのソースコードをデータセットとして用いて学習を行うことが可能であると考えられる。

本論文では、プログラミング言語の文法を獲得するために必要となる非終端記号の決定手法を提案する。本手法では、非終端記号を互いに交換可能なトークン列の集合と捉えることで非終端記号を決定する。同じ非終端記号から生成可能な文字列同士は周囲に似たトークンが現れると考えられるため、周囲に現れるトークンが似たトークン列同士は互いに交換可能であると仮定する。本手法では、周辺情報が似たトークン列同士のベクトル表現の距離が近くなるように、出現し得る各トークン列にベクトル表現を与える。このようなトークン列のベクトル表現を求めるために、文脈を左右で分けて学習するような Skip-gram モデルを用いる。割り当てたベクトル表現を用いてトークン列をクラスタリングすることによって、互いに交換可能なトークン列の集合を決定する。この処理によって得られた非終端記号を用いて、プログラミング言語の文法の獲得と、獲得した文法を用いた構文解析を行う。本手法で獲得する生成規則には適用確率が与えられており、この適用確率の更新には Inside-outside アルゴリズムを使用する。提案手法については 3 章でより詳細に述べる。本研究の主な貢献は以下の二点である。

- 文脈を左右で区別するような Skip-gram モデルを用いたトークン列へのベクトル表現の割り当てとクラスタリングを用いた非終端記号の獲得手法を提案した。
- 提案手法によって獲得した文法を用いて構文解析を行い、単純な生成規則からなる文法に対して提案手法の有用性を示した。

1.2　本論文の構成

本論文の残りの部分について、構成内容を述べる。2 章では、本研究と関連研究の理解に必要となる前提知識についての説明と、本研究で取り組むプログラミング言語の文法推定について述べる。また、ソースコードに対して機械学習手法を適用した研究やプログラミング言語の文法推定に取り組んだ既存研究についても述べる。3 章では、提案手法の説明と提案手法に基づいて作成したシステムについての説明を行う。本システムは初期の文法を決定するまでの前半処理と、各文法の適用確率を更新する後半処理の二段階に分かれている。4 章では本手法の妥当性を検証するために行った実験の内容とその実験結果について述べる。実験のために作成

[*1] https://github.com
[*2] https://sourceforge.net
[*3] https://stackoverflow.com

したデータセットから、本システムを用いて文法の学習を行い、評価用のデータセットに対して構文解析を行うことで評価を行なった。ここで使用したデータセットは、単純な文法からその文法に従う文字列をランダムに生成することで作成している。適切な閾値を与えた場合、簡単なデータセットについては良い性能を示していることが確認できた。しかし、異なる意味で使用されるトークンが現れるようなデータセットに対しては良い結果が得られなかった。この結果についての考察も 4 章で行なっている。最後に、5 章で本研究のまとめと実験結果を踏まえた今後の課題について述べる。

第 2 章

研究背景

ソフトウェア開発における開発効率の向上などの観点から、ソースコードに対する静的解析は重要視されている。ソースコードの解析に関連する研究として、クローン検出、コーディングスタイルの検査や自然言語を用いたコード検索などの多様なタスクが取り組まれてきた。また、GitHub 等で公開されているソースコードを利用して、上記のタスクに機械学習手法を適用した研究も増えてきている。本章では、本研究に関連する前提知識の説明と本研究で取り組むプログラミング言語の文法推定について述べる。また、関連研究としてソースコードを対象として機械学習の手法を適用した研究例や既存の文法推定手法についても述べる。

2.1 前提知識

本節では、本研究の内容を説明するにあたって必要となる前提知識について述べる。本研究の目標であるプログラミング言語の文法の獲得と、本研究の提案手法を述べるにあたって必要となる前提知識を以下に示す。

- 文脈自由文法と構文解析手法について
- 確率的文脈自由文法と学習アルゴリズムについて
- Neural Network について
- Skip-gram モデルについて

はじめに、文脈自由文法とその構文解析手法について 2.1.1 節で説明をする。本研究で作成したシステムでは、確率的文脈自由文法と同じ形式で生成規則を獲得する。そのため、確率的文脈自由文法とその学習アルゴリズムである Inside-outside アルゴリズムについて 2.1.2 節で説明をする。提案手法では主に自然言語処理の単語のベクトル埋め込みに用いられている Skip-gram モデルを応用してトークン列のベクトル化を行う。そこで、Neural Network の基本的な構造について 2.1.3 について説明し、Skip-gram モデルについての概要を 2.1.4 節で説明する。最後に、本研究の実験で用いる評価指標として Accuracy、Precision、Recall、F1 score についての説明を 2.1.5 節で行う。

$$
\begin{array}{lcl}
S & \rightarrow & Stmt* \\
Eq & \rightarrow & \text{“}=\text{”} \\
Lp & \rightarrow & \text{“(”} \\
Rp & \rightarrow & \text{“)”} \\
Op & \rightarrow & \text{“}+\text{”} \\
Var & \rightarrow & [\text{“}a\text{”} - \text{“}z\text{”}]+ \\
Expr & \rightarrow & Var \\
 & | & Var \quad Op \quad Expr \\
 & | & Lp \quad Expr \quad Rp \\
Stmt & \rightarrow & Var \quad Eq \quad Expr
\end{array}
$$

図 2.1. EBNF で記述された文法例

```
a = b + c
foo = (bar + baz) + foo
```

図 2.2. 図 2.1 の文法から生成可能な文字列例

2.1.1 文脈自由文法と構文解析手法

形式文法は、非終端記号と終端記号と生成規則、開始記号から成る生成規則は $A \rightarrow abc$ のような形式で記述し、$\rightarrow$ の左辺の記号から右辺の記号列へ置換をすることができることを表す。終端記号とは生成規則による置換をそれ以上行うことができない文字を指す。また、生成規則によって置き換え可能な記号は非終端記号と呼ばれる。EBNF を用いて記述した生成規則の例を図 2.1 に示す。ここでは、ダブルクォート (“”) で囲ってある記号が終端記号を表している。図 2.1 の文法から生成可能な文字列の例を 2.2 に示す。ここで、開始記号は S とする。図 2.2 の文字列のような、開始記号から始めて生成規則による置換を繰り返すことによって生成できる文字列の集合を言語と呼ぶ。一つの言語を生成するような文法は複数考えられる。

形式文法の包含関係はチョムスキー階層として定義されている [1]。チョムスキー階層では、形式文法を以下の四つのタイプに分類する。各文法は図 2.3 のような包含関係を持っている。

タイプ 0 制限なし文法
タイプ 1 文脈依存文法
タイプ 2 文脈自由文法
タイプ 3 正規文法

タイプ 0 の制限なし文法は全ての形式文法を含み、次いでタイプ 1 の文脈依存文法、タイプ 2 の文脈自由文法、タイプ 3 の正規文法といった順に包含関係を持つ。これらの文法から生成できるような言語をそれぞれ帰納的可算言語、文脈依存言語、文脈自由言語、正規言語と

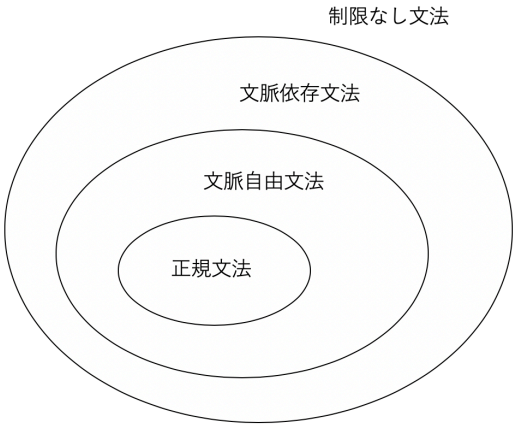

図 2.3. チョムスキー階層の包含関係

呼ぶ。多くのプログラミング言語の文法は文脈自由文法に含まれており、ソースコード中の各トークンは正規文法で表現できるのが一般的である。

形式文法が、1 個の非終端記号から 0 個以上の非終端記号と終端記号の列に変換できるような生成規則のみを持つとき、その文法は文脈自由文法と呼ばれる。形式的には、文脈自由文法は四つ組 $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, s)$ で表される。ここで、$\mathcal{N}$ は非終端記号の有限集合、$\mathcal{T}$ は終端記号の有限集合、$\mathcal{R}$ は生成規則の集合、s は開始記号である。生成規則は $u \to v$ の形式で表され、$u \in \mathcal{N}, v \in (\mathcal{N} \cup \mathcal{T})^*$ である。ここで $*$ はクリーネスターと呼ばれ、クリーネ閉包を作成する。

文脈自由文法の標準形の一つとしてチョムスキー標準形がある。全ての文脈自由文法は、チョムスキー標準形に変換可能であることが知られている。チョムスキー標準形の文法は、以下のような形式のみで記述された文法である。

$$\begin{aligned} A &\to B\ \ C \\ A &\to \alpha \\ S &\to \epsilon \end{aligned}$$

ここで、A, B, C は非終端記号を表し、α は終端記号を表す。また、S は開始記号で ϵ は空文字列を表す。

チョムスキー標準形の形式で表された文法を用いて構文解析を行う手法として Cocke-Younger-Kasami(CYK) 構文解析 [2, 3] が知られている。CYK 構文解析は動的計画法に基づく効率的な構文解析手法であり、短い区間から順に表を埋めることによって構文解析を行う。入力文字列の長さを n としたとき、$\mathcal{O}(n^3)$ で判定を行うことができる。

2.1.2　確率的文脈自由文法と学習アルゴリズム

文脈自由文法の各生成規則に、その生成規則の適用確率が付属するような文法を確率文脈自由文法と呼ぶ。確率的文脈自由文法は主に自然言語処理などの分野で用いられている。自然言

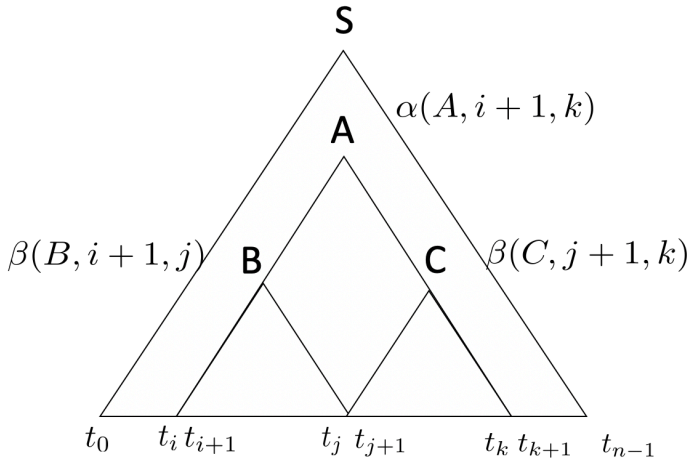

図 2.4. 単語列に対する Inside-outside アルゴリズムの適用

語処理では文脈自由文法の曖昧性を解決するために用いられる。各生成規則の適用確率の学習にはツリーバンクといった木構造形式のコーパスを用いた教師あり学習が行われる。構文木を教師として用いない場合には、EM アルゴリズム [4] が学習に使用される。EM アルゴリズムによって愚直に計算を行おうとすると構文木の種類が多くなり、計算が困難であるため、通常は EM アルゴリズムの効率的な手法として Inside-outside アルゴリズム [5] が用いられる。

Inside-outside アルゴリズムは以下の手順で行われる。

1. 各生成規則の適用率に適当な初期値を割り振る
2. 現在の適用確率を用いて内側確率と外側確率を計算する (内側確率・外側確率は後述)
3. 生成規則の適用確率の期待値を計算する
4. 各生成規則の適用確率の更新を行う
5. 2 に処理を戻す

内側確率 $\beta(A, i, j)$ とは非終端記号 A から単語列 $t_i, \ldots, t_j$ を導出する確率である。また、外側確率 $\alpha(B, i, j)$ とは開始記号から $t_0, \ldots, t_{i-1}, B, t_{j+1}, \ldots, t_{n-1}$ を導出する確率である。ここでは全体の単語数を n としている。図 2.4 のような場合、生成規則 $A_{(i+1,k)} \rightarrow B_{(i+1,j)}C_{(j+1,k)}$ の適用回数の期待値は式 2.1 で表される。

$$\frac{1}{Z} \times P(A \rightarrow BC) \times \beta(B, i+1, j) \times \beta(C, j+1, k) \times \alpha(A, i+1, k) \tag{2.1}$$

ここで、$Z = \alpha(S, 0, n-1)$ である。また、$P(A \rightarrow BC)$ は生成規則 $A \rightarrow BC$ の現在の適用確率を表す。

2.1.3 Neural Network

単純な Neural Network[6] は、図 2.5 のような複数の層を連結させた構造を持つ。Neural

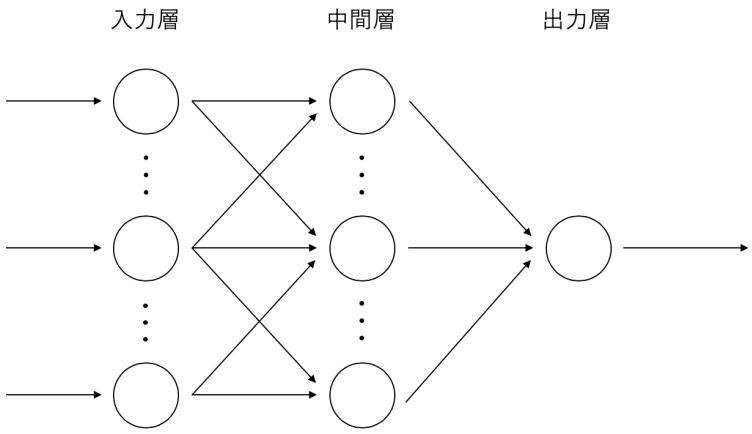

図 2.5. Neural Network の構造

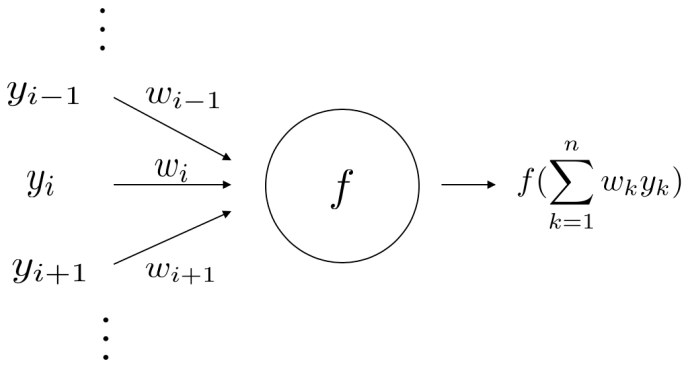

図 2.6. ユニットにおける活性化関数を用いた計算

Network の各層は入力層・中間層 (隠れ層)・出力層に分けられている。各層はユニットという計算ノードで構成され、各ユニットは活性化関数を持つ。各層の間には重みが定義されており、この重みを更新することによってネットワーク全体の学習を行う。計算時には入力層に入力データを与えて、連結する層の間で値を受け渡しながら順に計算を行うことによって出力層から計算結果を得る。ここで、各ユニットにはベクトル値が入力として与えられ、自身の活性化関数を適用した結果をそのユニットの出力とする (図 2.6)。ユニットの出力値は連結する次の層のユニットの入力値の一部となる。$(j-1)$ 層目に n 個のユニットがあるとすると、j 層目の i 番目のユニットへの入力 x_i^j は一般に式 2.2 で表される。

$$x_i^j = \sum_{k=1}^{n} w_k^j y_k^{j-1} \tag{2.2}$$

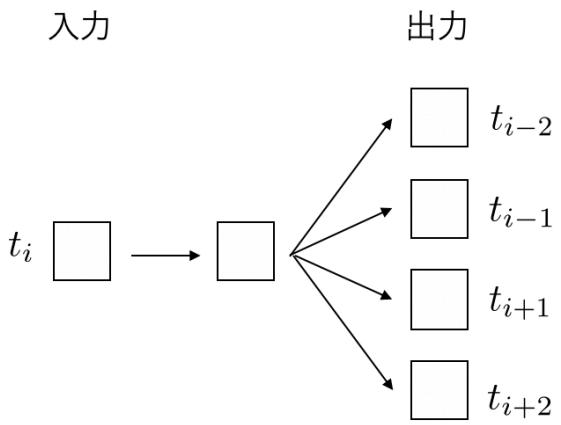

図 2.7. Skip-gram モデルの概略図

ここで、y_k^j は j 層目の k 番目のユニットの出力を表し、w_k^j は j 層目の k 番目の重みを表す。また、j 層目の i 番目のユニットの出力 y_i^j は入力 x_i^j を用いて式 2.3 で表される。

$$y_i^j = f(x_i^j) \tag{2.3}$$

ここで、f は該当ユニットの活性化関数である。単純な活性化関数としてはシグモイド関数 (式 2.4) や正規化線形関数 (式 2.5) 等が知られている。

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

$$f(x) = max(0, x) \tag{2.5}$$

2.1.4 Skip-Gram モデル

単語のベクトル埋め込みは重要なタスクの一つである。獲得した単語の分散表現は自然言語処理などの多くのタスクに応用されている。通常、単語のベクトル埋め込みでは、単語のベクトル表現同士の距離が近いことがその単語同士の意味が近いことに対応するように各単語にベクトル表現を割り当てる。

単語のベクトル表現の獲得手法として Mikolov らによって Skip-gram モデル [7, 8] が提案されている。Skip-gram モデルでは、注目している単語からその周辺に登場する単語を予測するような Neural Network を学習させる。図 2.7 に Skip-gram モデルの概略図を示す。ここで、t_i は注目している単語であり、$t_{i-2}, t_{i-1}, t_{i+1}, t_{i+2}$ は t_i の周辺の単語である。周辺の単語をいくつ隣まで使用するかを示す値をウィンドウサイズと呼ぶ。例として、以下のような単語列が与えられたとする。

Five hexing wizard bots jump quickly

ここで、今注目している単語を “wizard” とすると、ウィンドウサイズが 2 のときに文脈となる単語は “Five”、“hexing”、“bots”、“jump” となる。そのため、上記のネットワー

クに入力データとして与える値とラベルの組は (“wizard”, “Five”), (“wizard”, “hexing”), (“wizard”,“bots”), (“wizard”,“jump”) となる。注目する単語を移動させながら同様の入力データを各単語に対して作成し、ネットワークの学習を行う。これよって得られるネットワークの重み行列から各単語に対応するベクトル表現を抜き出すことで、単語のベクトル表現を獲得する。

2.1.5　評価指標

機械学習を用いた分類問題でモデルの評価をする際には、多くの評価指標が用いられる。ここでは、Accuracy、Precision、Recall、F1 score の四つの評価指標について説明をする。4 章ではこれら評価指標を用いて提案手法の評価を行う。

正と負の二値分類を考えるとき、分類器の予測値と実際の値の各組み合わせは以下のように呼ばれる。

True Positive(TP)　正であると予測して、実際に正である場合
True Negative(TN)　負であると予測して、実際に負である場合
False Positive(FP)　正であると予測したが、実際は負である場合
False Negative(FN)　負であると予測したが、実際は正である場合

単純な評価指標として、データ全体の中での正解した割合を表す Accuracy が使用される。Accuracy$A$ は式 2.6 で表される。

$$A = \frac{|TP| + |TN|}{|TP| + |TN| + |FP| + |FN|} \tag{2.6}$$

ここで、$|TP|, |TN|, |FP|, |FN|$ はそれぞれ True Positive の数、True Negative の数、False Positive の数、False Negative の数を表す。以降では同じ意味で $|TP|, |TN|, |FP|, |FN|$ を用いる。

正であると予測されたものの中で、実際に正であったものの割合を表す指標を Precision と呼ぶ。Precision P は式 2.7 で表される。

$$P = \frac{|TP|}{|TP| + |FP|} \tag{2.7}$$

Precision は [0, 1] の範囲で値をとる。正であると予測したが実際は負であるような場合を減らしたい時、つまり False Positive を低く抑えたい場合には Precision が高いモデルを採用すると良い。

実際に正であるものの中で、正と予測できたものの割合を表す指標を Recall と呼ぶ。Recall R は式 2.8 で表される。

$$R = \frac{|TP|}{|TP| + |FN|} \tag{2.8}$$

Recall は Precision と同様に [0, 1] の範囲で値をとり、False Negative を低く抑えたい場合には Recall が高いモデルを採用するべきである。Precision と Recall はトレードオフの関係に

あるため、アプリケーションによって Precision を重視するべきか Recall を重視するべきかを考慮する必要がある。

Precision と Recall の両方を考慮するような評価指標として F1 score がある。F1 score は Precision と Recall の調和平均であり、式 2.9 で表される。

$$F = \frac{2P \cdot R}{P + R} \tag{2.9}$$

F1 score も同様に [0, 1] の範囲で値をとり、値が大きいほど良い。Precision と Recall のどちらかが低いと F1 score は下がってしまうため、F1 score は両者の値が悪くないことを確認する際に用いられる。

2.2　ソースコードへの機械学習手法の適用

近年では、ソースコードに対して機械学習手法を用いた研究も行われている。オープンソースソフトウェアの増加や GitHub などのプラットフォームの普及によってソースコードを学習データとして集めることが以前より容易になったことが、その要因の一つである。ソースコードに対する機械学習手法は、自然言語処理分野などの知見を活かして幅広いタスクに適用されている。

ソースコードに対して機械学習を適用する研究は、既存の様々なタスクに対して行われている。例として、プログラミング言語のコード生成が挙げられる。コード生成は自動プログラミングと呼ばれ長く研究がされてきたタスクであるが、近年ではコード生成のタスクに Recurrent Neural Network(RNN)[9, 10] を用いる研究なども行われている [11]。また、Latent Prediction Network という Neural Network のモデルを提案してコード生成に取り組んだ研究も行われている [12]。コード生成以外のタスクでも、ソースコードに対する機械学習手法の応用は行われている。Hellendoorn らはソースコード中の変数等に対して型提案を行うタスクに RNN を利用した手法を提案している [13]。また、Stack Overflow などの Web サービスを通して、ソースコードと自然言語が紐付くようなデータの収集が可能になっていることもあり、プログラミング言語と自然言語の対応付けも大きな関心事の一つとなっている。このような研究として、Gu らは自然言語のクエリからコードスニペットの検索を行う手法を提案している [14]。この研究では、RNN に基づく CODEnn というモデルを提案し、同時に CODEnn モデルを使用した DeepCS というシステムを実装している。他のタスクでは、RNN を用いてコードクローンの検出を行った White らの研究 [15] やバグの検出に Neural Network を用いた研究 [16]、バグ修正に Neural Network を用いた研究 [17] なども行われている。

2.3　プログラミング言語の非終端記号の獲得

ソフトウェア開発の現場では、ソースコードの大規模化や複雑化が起こることなどから、ソースコード解析の需要が高まっている。ソースコードの解析技術として、バグの検出や型提

案などの多くのタスクが取り組まれている。ソースコードなどの解析においては、対象の構造をどのように獲得するかが非常に重要である。このような構造をプログラミング言語の文法として獲得することができれば、幅広いタスクへの適用が可能であると考えられる。既存のソースコード群から新たな構造を文法として獲得することで、コーディングスタイル検査器やログ解析などへの応用が期待できる。

プログラミング言語の文法獲得も既に取り組まれているタスクの一つである。通常、プログラミング言語は言語設計者が文法を含めて仕様を定める。つまり、その言語で記述されたソースコードからプログラムを実行するために従うべき文法が既に存在する。しかし、既に定めてあるプログラミング言語の文法を、ソースコードを基に改めて獲得することにも意味がある。例として、プログラミング言語の文法更新への対応を可能にすることができる。プログラミング言語に必要とされる言語機能や文法は時代とともに移り変わっていく。そのため、プログラミング言語の文法は言語自身のバージョンアップデートと共に変更されていくことになる。この文法の変更は、小さなシンタックスの追加から互換性を失うような大きな変更まで様々である。プログラミング言語の文法を前提としたソフトウェアは数多く存在するが、対象のプログラミング言語の文法の更新がある場合、これらのソフトウェアの開発者は文法の更新に伴う修正をそのソフトウェアに加える必要がある。また、プログラミング言語のバージョンアップデートによってオンラインドキュメントが不完全になってしまう [18] という問題も指摘されている。プログラミング言語の文法を既存のソースコードから獲得することによって、一部の文法を元に全体の文法を獲得するような手法が提案されている。

自然言語処理の分野でも文法推定は行われている。プログラミング言語と自然言語の違いは、非終端記号や生成規則の数が一般にプログラミング言語の方が膨大になることなどが挙げられる。また、プログラミング言語の場合は深い再帰構造を持つことや上記のように文法のアップデートが行われることなどの違いがある。

プログラミング言語の文法に大きく依存したソフトウェアや解析ツールなどでは、通常はプログラミング言語ごとに開発を行う必要がある。コーディングスタイル検査器のようなソフトウェアでは、既存のソースコードから事前の文法情報なしに文法を獲得することができればプログラミング言語に依存しないようなツールの開発を行うことも可能であると考えられる。

本研究では、プログラミング言語が持つ構造を文法として獲得することを目指す。また、そのために必要となる非終端記号の獲得手法について検討する。ここでは、ある言語のソースコード群から非終端記号と生成規則に対応する構造を獲得することをその言語の文法を獲得することであるとする。本研究で獲得を目指す文法は、必ずしも本来のプログラミング言語の文法とは限らない。コーディングスタイル検査器のようなアプリケーションでは、元のプログラミング言語の文法とは異なる何らかの文法のような構造があると仮定すると、ソースコードから文法として構造を獲得することによってコーディングスタイルの検査を行うことができると考えられる。

2.4 コーディングスタイル検査器への応用

ソフトウェア開発において、ソースコードの可読性の向上は重要視されている。コーディングスタイルを統一することによって、開発効率の向上やバグの低減に繋がるためである。複数人が同じソフトウェアの開発に関わるような大規模な開発になると、各個人のコーディングスタイルではなく開発者で共通のコーディングスタイルを設けてそれに従う必要がある。企業ごと、開発チームごとやプロジェクトごとなど、共通のコーディングスタイルを設ける単位は様々である。また、一般に用いられるコーディングスタイルの他にも独自のコーディングスタイルを規定してそれに従うということも少なくない。個人が複数のプロジェクトに関わっている場合などは、複数のコーディングスタイルを使い分けて記述する必要もでてくる。

このようなコーディングスタイルの検査を人力で行うとコストがかかり、人為的なミスや見落としも発生することからツールを用いて自動的に検査を行うことが望ましい。そのため、各プログラミング言語やコーディングスタイルごとにコーディングスタイルの静的解析器が開発されている。例として、Ruby 言語の RuboCop[*1]は Ruby コミュニティが定めているコーディングスタイル[*2]に従っているかどうかを検査する。また、Python 言語の pycodestyle[*3]は Python 設計者が定めた PEP8[*4]に従っているかどうかを検査する。これらのコーディングスタイル検査器のユーザは、追加のコーディングスタイルを定める場合に設定ファイルの記述をする必要がある。また、開発者側にもプログラミング言語ごとに同様のソフトウェアを開発しなければならないという問題がある。

小林は Bi-directional LSTM[19] を用いてコーディングスタイルの誤り検知を行う手法を提案している [20]。この手法は上記の問題を解決し得るが、コーディングスタイルに何故違反したかが分かりづらいという課題が残った。本研究内容をコーディングスタイル検査器に応用することができれば、コーディングスタイルを文法だと考えて検査することが可能である。獲得した文法を用いて構文解析をした際にどの規則に違反したかを文法エラーとして出力する事は可能であるため、コーディングスタイルの修正の指針とすることができるだろう。

2.5 関連研究: プログラミング言語の文法推定

プログラミング言語の文法推定に取り組んだ既存研究として、Saha が提案した Gramin[21] というシステムがある。Gramin は ANTLR[22] の形式で表された初期文法を更新することで最終的な文法を獲得し、ANTLR の形式で出力する。Gramin は初期値となる文法が既にあることを前提としており、その初期文法を基にプログラミング言語が従っている文法の獲得を行なっている。はじめに、対象のプログラミング言語の字句解析器を用いて文字列をトークン列

[*1] https://github.com/rubocop-hq/rubocop
[*2] https://github.com/rubocop-hq/ruby-style-guide
[*3] https://github.com/PyCQA/pycodestyle
[*4] https://www.python.org/dev/peps/pep-0008

へと変換し、CYK 構文解析 [3, 2] に基づくアルゴリズムと複数のヒューリスティックな手法の組み合わせによって文法の獲得を行う。Gramin は Java と XSB Prolog[*5]を用いて実装されている。この手法はプログラミング言語の文法がバージョンアップデートとともに変更されてしまうことによる問題を解決し得る。しかし、前述のコーディングスタイル検査器のような、プログラミング言語が既に定めている文法とは別の構造を抜き出したいような場合には応用が困難であると考えられる。

[*5] https://www.xsb.com/what-we-do/emerging-technologies/xsb-prolog.html

第 3 章

プログラミング言語の文法推定に向けた非終端記号の獲得

3.1 Skip-gram モデルを用いた非終端記号の決定と文法の学習法

本研究では、プログラミング言語の文法獲得へ向けた非終端記号の獲得手法を提案する。本節では、提案手法の基本となるアイデアについて述べる。プログラミング言語の文法を獲得するためには、その文法を構成する非終端記号や生成規則の獲得をしなければならない。ある文法において、同じ非終端記号から生成可能な文字列同士は、互いに置き換えたとしても文法としては正しい。したがって、非終端記号とは互いに交換可能なトークン列の集合であると考えることができる。ここで、交換可能なトークン列とは同じ非終端記号から生成可能な文字列である。例として、図 3.1 のような生成規則があった場合、以下のようなトークン列は全て同じ非終端記号 $Expr$ から生成可能であり、互いに入れ替えても文法としては正しいままである。

- “foo”
- “1”
- “$a + b$”
- “$a + b + 2$”
- “$1 + 2 + 3$”

非終端記号を獲得することは、上記のような交換可能なトークン列の集合を獲得することと等しい。生成規則が既にわかっている場合は同じ非終端記号から生成可能なトークン列を選ぶこ

$$
\begin{array}{lcl}
Var & \rightarrow & [\text{“}a\text{”} - \text{“}z\text{”}]+ \\
 & \rightarrow & [0-9] \\
Expr & \rightarrow & Var \\
 & | & Var \quad \text{“}+\text{”} \quad Expr
\end{array}
$$

図 3.1. 生成規則の例

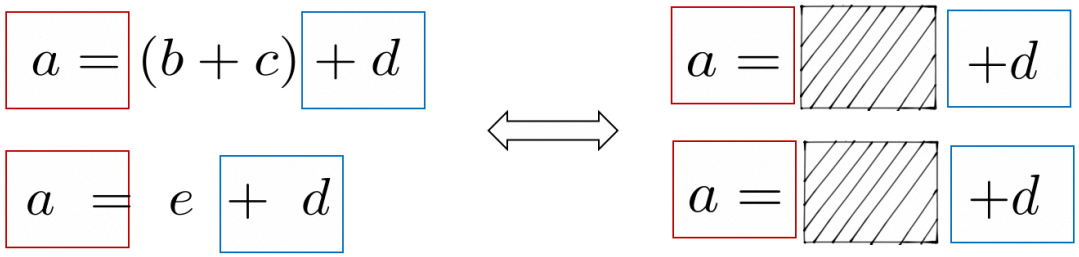

図 3.2. 互いに交換可能であるとみなせるトークン列の例

とで交換可能なトークン列の集合がわかったが、今回獲得したいのはその生成規則自身である。そこで、生成規則がわからない状態では、どのように交換可能なトークン列を定めると良いかを考える。ここで、同じ非終端記号から生成されたトークン列同士は、周辺に同様のトークンが多く現れると考えられる。そのため、本手法では周辺に現れるトークンが同様なトークン列同士を交換可能であることとする。例として、図 3.2 のような二種類の式があるとき、トークン列 “(b + c)” と “e” は周辺に同様のトークンを持つことから、互いに交換可能なトークン列であると見なすこととする。また、本手法では上記の例のように厳密に同じトークンが現れる場合のみではなく、トークン列の周辺情報を元に周辺情報が似ているトークン列同士をまとめ上げる事とする。ここで、Skip-gram モデルを用いてトークン列にベクトル表現を与えることを考える。Skip-gram モデルによって与えられたトークン列のベクトル表現の距離が近い場合は、それらのトークン列同士の文脈が似ているということを表している。そのため、言語内に出現し得るトークン列に対してベクトル表現を与えて、それらをクラスタリングすることによって交換可能なトークン列の集合を獲得することを考える。通常の Skip-gram モデルはトークン (単語) 単位でベクトル表現を与えるため、トークン列に対してベクトル表現を与えることは愚直にはできない。また、ある文法から生成されるトークン列は無限に考えられるため、それらを列挙することは不可能である。そこで、本手法では各トークンに対して「左側の文脈のみを考慮したベクトル表現」と「右側の文脈のみを考慮したベクトル表現」の二種類のベクトル表現を与えることでこの問題を解決する。トークン列のベクトル表現は、そのトークン列の周辺情報を埋め込んでいることが重要であった。そのため、今回はトークン列に対するベクトル表現を以下の二つのベクトル表現を組み合わせたベクトル表現を用いて表すこととする。

i). トークン列の左端のトークンの「左側の文脈のみを考慮したベクトル表現」
ii). トークン列の右端のトークンの「右側の文脈のみを考慮したベクトル表現」

以降では、この考え方に基づいて作成したシステムについての説明を行う。また、本システムの具体的な非終端記号の獲得手法と文法の推定手法について述べる。

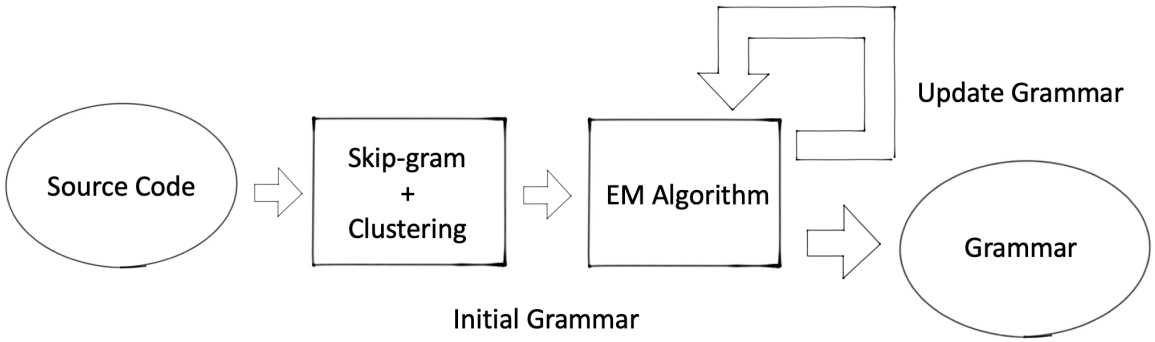

図 3.3. 本システムの概略図

3.2　本研究の提案手法とシステムの文法獲得手順

本論文で提案する手法では、はじめに「生成規則の集合」の初期値を決定し、その「生成規則の集合」の各適用確率を更新していくことで最終的な文法を獲得する。本システム全体の概略図を図 3.3 に示す。図 3.3 のように、本手法は処理が大きく二段階に分かれる。前半部分で初期の「生成規則の集合」を決定し、後半部分で現在の「生成規則の集合」の各適用確率の更新処理を複数回行うことで最終的な文法獲得を行う。後半部分の更新回数はユーザがパラメータとして指定する。前半部分の初期の「生成規則の集合」を作成するまでの処理の説明を 3.2.1 節で行う。また、後半部分の「生成規則の集合」を用いて構文解析を行うことで「生成規則の集合」の各適用確率を更新する処理の説明は 3.2.2 節で行う。

3.2.1　前半部分: 「生成規則の集合」の初期値を決定するための前処理

はじめに、データセットから「生成規則の集合」の初期値を決定するための処理について述べる。データセットは特定のプログラミング言語で記述された複数のファイルから成り、前処理はデータセット内の全てのファイルに対して行われる。前半部分では、以下の処理を順に行うことで、最終的に「生成規則の集合」の初期値を決定する。

1. 対象のプログラミング言語の字句解析器を用いて文字列をトークン列に変換する。
2. 出現ファイル数に基づいてトークンの置換を行う。
3. Skip-gram モデルを用いて各トークンに対して二種類のベクトル表現を与える。
4. トークンの組に対して単一のベクトル表現を割り当てる。
5. 割り当てたトークンの組のベクトル表現を用いてトークンの組をクラスタリングする。
6. 生成規則の集合の初期値を決定する。

以降では、これらの処理の詳細について述べる。

$$a = (b + c) + d$$

$$\Downarrow$$

$$\text{“}a\text{”}, \text{“}=\text{”}, \text{“}(\text{”}, \text{“}b\text{”}, \text{“}+\text{”}, \text{“}c\text{”}, \text{“})\text{”}, \text{“}+\text{”}, \text{“}d\text{”}$$

$$Var, \quad Eq, \quad Lp, \quad Var, \quad Op, \quad Var, \quad Rp, \quad Op, \quad Var$$

図 3.4. 文字列からトークンとトークンの型情報への変換

字句解析器を用いた文字列からトークン列への変換

はじめに、データセット内の各ファイルに記述された文字列を字句解析器を用いてトークン列に分割する。ここで使用する字句解析器は、学習の対象とするプログラミング言語の字句解析器をそのまま用いる。字句解析処理では、「字句解析時に何であると認識されたか」を示す情報が各トークンに対して与えられる。以降では、この情報をトークンの型と呼ぶこととする。この処理では、図 3.4 の a, b のように、トークンの型が同じトークンであっても別々のトークンとして分割される。トークンの型がスペース、改行またはコメントのいずれかであると認識されたトークンは、この段階で取り除かれる。

出現ファイル数に基づくトークンの置換

字句解析処理の次に、出現ファイル数に基づいてトークンをトークンの型で置換する処理を行う。そのために、まずデータセット内に出現する各トークンの出現ファイル数の数え上げを行う。出現ファイル数を数え上げる対象はトークンの型ではなくトークン自身である。本システムでは、出現回数がデータセットに用いたファイル数の 5% に満たないトークンをトークンの型で置き換える。図 3.5 の a, b, c, d のような変数名は、通常は開発するアプリケーションや処理内容によって決定される。特定の変数名のようなデータセット全体に共通して現れないトークンは、トークン自身をトークンの型で置換する。図 3.5 の場合は、a, b, c, d の出現回数が 5% に満たないとするとトークンの型である Var に置換されることになる。この処理は、「一般的に用いられていない出現度の低いトークン」を「同一のトークンの型を持つ場合には同一のトークンとして扱う」ことを目的として行っている。一般に、同じトークン名でも共通のファイルやライブラリ内等では何度も使用される。同じトークン名が一つのファイル内で多く用いられる場合にそのトークンを残してしまうのは目的に沿わないので、総出現回数ではなく出現ファイル数による置き換えとしている。

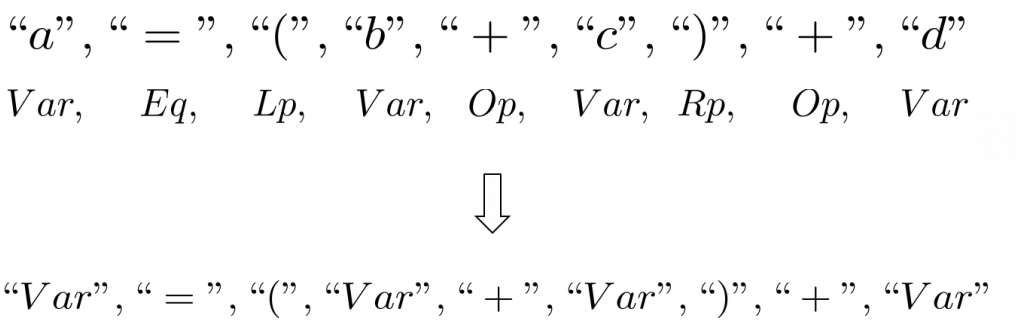

図 3.5. トークンの出現ファイル数に基づくトークンの置き換え処理

トークンに対する二種類のベクトル表現の獲得

Neural Network のモデルの一つである Skip-gram モデルを用いて各トークンに対してベクトル表現を与える。通常の Skip-gram モデルでは、注目している単語に対してその左右に出現する単語をラベルとするような教師あり学習を行う。そして、この学習によって得られる Neural Network の重みベクトルを単語に対するベクトル表現として扱う。本手法では、図 3.6 のように各トークンに対して左右の文脈を区別した二種類のベクトル表現を与える。一種類目のベクトル表現を得るために、注目しているトークンに対して、その左隣に出現するトークンをラベルとして与えて学習を行う。二種類目のベクトル表現も同様に、考慮する文脈の向きを反対にして各トークンに対してその右隣に出現するトークンをラベルとするような学習を行う。ここで用いる Skip-gram モデルのネットワークはそれぞれ異なるものを使用する。これにより、各トークンに対して「左側の文脈のみを考慮したベクトル表現」と「右側の文脈のみを考慮したベクトル表現」が与えられる。

トークンの組に対するベクトル表現の割り当て

次に、トークンの組に対してベクトル表現の割り当てを行う。ここまでの処理によって、各トークンに対して「左側の文脈のみを考慮したベクトル表現」と「右側の文脈のみを考慮したベクトル表現」が与えられている。まず、データセットに出現したトークンを用いてトークンの組を作る。データセット内にトークン数が n 個あるとき、n^2 個の組が作成される。ここで、トークンの組 (t_i, t_j) のベクトル表現を t_j の「左側の文脈のみを考慮したベクトル表現」と t_j の「右側の文脈のみを考慮したベクトル表現」を連結したベクトル表現とする。トークン列 $t_i, \ldots, t_j$ が与えられるとき、トークン列のベクトル表現をトークンの組 (t_i, t_j) のベクトル表現とする。各トークンの組に対して「左側の文脈のみを考慮したベクトル表現」と「右側の文脈のみを考慮したベクトル表現」を連結したベクトル表現を作成し、この連結したベクトル表現をトークンの組のベクトル表現として扱う。例として、図 3.7 のように、トークン列

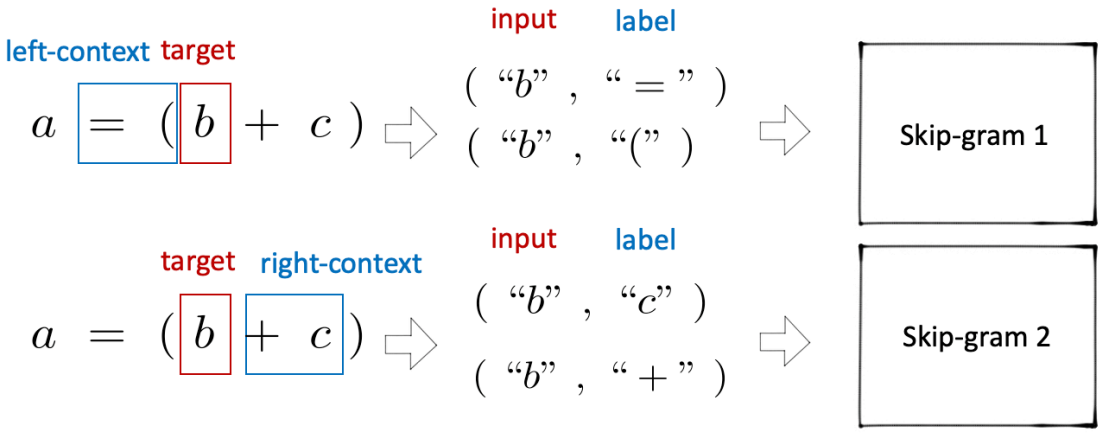

図 3.6. Skip-gram モデルによる二種類のベクトル表現の付与

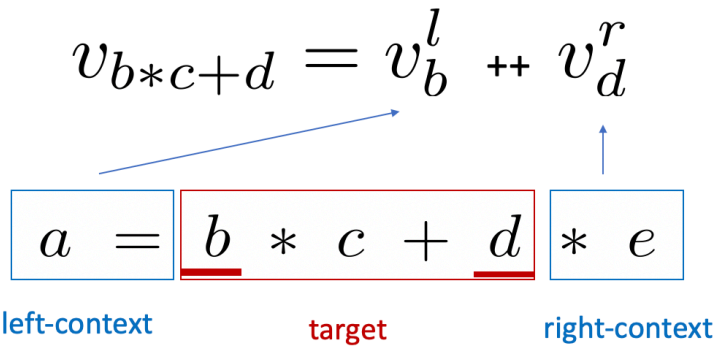

図 3.7. 本手法におけるトークン列のベクトル表現

$b*c+d$ のベクトル表現は、トークン b の「左側の文脈のみを考慮したベクトル表現」とトークン d の「右側の文脈のみを考慮したベクトル表現」を連結したベクトル表現となる。

トークンの組のベクトル表現を用いたクラスタリング

トークンの組のベクトル表現の距離を用いてクラスタリングをする。ここで、クラスタリング手法には X-means 法 [23] を用いた。クラスタリングによって、各トークンの組は何らかのクラスタに分類される。

トークンの組のベクトル表現は、左端と右端にそのトークンが現れるようなトークン列のベクトル表現であった。二つのトークン列に対して、それらのトークン列と対応するトークンの組が同一のクラスタに割り当てられている場合、それらのトークン列は同一の非終端記号に還元可能であるものとする。このクラスタリング処理はシステム全体を通して一回のみ行われる。

「生成規則の集合」の初期値の決定

本システムではチョムスキー標準形で表される形式で文法を獲得する。初期値となる生成規則をはじめに決定し、その後各生成規則の尤度を更新することによって最終的な文法を決定する。ここでは、そのための初期文法の決定を行う。トークン t に対して「左側の文脈のみを考慮したベクトル表現」を v^l と表し、「右側の文脈のみを考慮したベクトル表現」を v^r とする。データセット内のファイルに記述されたトークン列 $t_i, \ldots, t_j, \ldots, t_k$ を考える。部分列 $t_i, \ldots, t_j$ のベクトル表現は左端 t_i の左側ベクトル表現 v_i^l と右側ベクトル表現 v_j^r を連結させたベクトルである。このトークン列 $t_i, \ldots, t_j$ のベクトル表現をを $v_{(i,j)}$ と表す。同様に、トークン列 $t_j, \ldots, t_k$ のベクトル表現を $v_{(j,k)}$ とし、トークン列 $t_i, \ldots, t_k$ のベクトル表現を $v_{(i,k)}$ とする。前段階のクラスタリングによって得られている $v_{(i,j)}$ が属するクラスタはトークン列 $t_i, \ldots, t_j$ から還元可能な非終端記号であった。$t_i, \ldots, t_j, \ldots, t_k$ から作成できる生成規則は、式 3.1 である。

$$C_{v_{(i,k)}} \quad \rightarrow \quad C_{v_{(i,j)}} \quad C_{v_{(j,k)}} \tag{3.1}$$

ここで、$C_{v_{(i,k)}}, C_{v_{(i,j)}}, C_{v_{(j,k)}}$ は各ベクトルが属するクラスタである。データセット内の全トークン列に対して式 3.1 のような生成規則を作成し、それらを生成規則の集合の初期値として扱う。また、各生成規則の適用確率にはランダムに初期値を与える。ここまでの処理によって得られた「生成規則の集合」を生成規則の集合の初期値として扱う。

3.2.2 後半部分: 「生成規則の集合」の更新

本システムの後半部分では、最終的な文法を獲得するために「生成規則の集合」の各適用確率の更新を行う。生成規則の初期値には、3.2.1 節の前処理によって得られた生成規則を使用する。文法の更新は Inside-Outside アルゴリズムを使用し、パラメータとして指定された回数だけ繰り返し更新を行う。通常の Inside-outside アルゴリズムでは既知の生成規則に対して与えられている適用確率を更新していくが、本手法では前処理で決定した生成規則とその適用確率を用いて適用確率の更新を行う。適用確率の更新時には、初期値の決定時と同様に、生成規則の適用確率の更新を行う際も非終端記号と前半処理で獲得したクラスタを対応させる。つまり、式 3.1 の形式の生成規則を用いて生成規則の適用確率の更新を行う。

3.3 獲得した文法による構文解析

一般の文法と同様に、本手法で獲得した文法を用いて構文解析を行うことができる。構文解析時は、閾値と文字列を入力として与えて、入力文字列が今回獲得した文法に従っているかどうかを出力する。ここで、閾値はどの生成規則が妥当であると見なすかを決定するために使用する。与えた閾値よりも適用確率が低い生成規則は、生成規則として妥当でないものとして構文解析時には使用をしない。

入力として与えた文字列には、「生成規則の集合」の獲得時と同様の前処理を施す。具体的

には、字句解析器によるトークンの分割を行い、出現ファイル数に基づくトークンの置換処理を行う。出現ファイル数の数え上げは学習時に数え上げたものを使用し、学習データ上で出現ファイル数が 5% 以下であったトークンはトークンの型で置換をする。ベクトル表現の割り当てとクラスタリング処理は既に行なっているのでこの時点で再度行う必要はない。その後は、閾値以上の適用確率を持つ生成規則のみを用いて、前処理を行なった入力文字列に対して CYK 構文解析を行う。CYK 構文解析を用いているため、入力トークン列長を n とすると $\mathcal{O}(n^3)$ で構文解析可能である。最終的にトークン列全体に対して何らかの非終端記号が生成可能であるとき、獲得した文法として正しい文字列であるとみなす。

第 4 章

実験

本研究では、非終端記号の決定方法が妥当であることを検証するために、簡易的な文法に対して文法の獲得をする実験を行った。本章では、行なった実験の内容とその結果について述べる。また、実験結果を踏まえて考察を行う。

本研究の実験には二種類の計算機を用いた。各計算機のハードウェア情報を表 4.1 に示す。本手法を用いた文法の獲得と評価データに対する構文解析の両方をこれらの計算機で行なっている。

表 4.1. 実験環境 (ハードウェア)

	計算機 1	計算機 2
CPU	Intel i7-6850K	Intel Xeon E5-2637 v3
CPU クロック周波数	3.60GHz	3.50GHz
CPU コア数	6	8
CPU スレッド数	12	16
GPU	Nvidia Quadro P6000	Nvidia Tesla P100
GPU メモリ	24GB	16GB

また、実験に用いたソフトウェアとソフトウェアのバージョンを表 4.2 に示す。開発環境の構築には Docker を使用し、Ubuntu 16.04 の Docker コンテナの上で学習・実験を行なった。本システムは主に Python を用いて実装されており、実装の一部にオープンソースライブラリを用いている。各トークン列に対するベクトル表現の割り当てに用いている Skip-gram モデルの実装には Tensorflow[*1]を使用している。また、クラスタリング処理には Python の機械学習ライブラリの一つである scikit-learn[*2]の実装を使用している。

今回は、実験のために四種類の文法からデータセットを自動生成している。ソースコードの自動生成を行うために、今回は Nim 言語[*3]を用いて文字列生成ツールを作成した。この生成

[*1] https://www.tensorflow.org/
[*2] https://scikit-learn.org/stable/
[*3] https://nim-lang.org/

表 4.2. 実験環境 (ソフトウェア)

ソフトウェア	バージョン
Docker	18.06.1-ce
OS	Ubuntu 16.04
CUDA	8.0
cuDNN	6.0
Python	3.6.5
Tensorflow	1.4.0
scikit-learn	0.19.2

$$
\begin{aligned}
Expr &\rightarrow Expr \quad \text{“}+\text{”} \quad Term \\
Expr &\rightarrow Expr \quad \text{“}-\text{”} \quad Term \\
Expr &\rightarrow Term \\
Term &\rightarrow Term \quad \text{“}*\text{”} \quad Num \\
Term &\rightarrow Term \quad \text{“}/\text{”} \quad Num \\
Term &\rightarrow Num \\
Num &\rightarrow 0 \\
Num &\rightarrow [1-9][0-9]*
\end{aligned}
$$

図 4.1. 文法 1(四則演算のみを持つ文法) の生成規則

ツールでは、DSL を用いて文法を定義し、その文法に従う文字列をランダムに生成する。

4.1　実験に使用した文法と各文法の生成規則

本論文では、提案手法を用いて以下の四種類の文法を対象とした実験を行った。

文法 1　四則演算のみを持つ文法
文法 2　四則演算と括弧を持つ文法
文法 3　四則演算と括弧と変数名、代入文を持つ文法
文法 4　if 文、四則演算、括弧、変数名、代入文を持つ文法

文法 1 から文法 4 へ順に、生成規則を拡張していくような形で文法を作成している。それぞれの文法の生成規則を図 4.1、4.2、4.3、4.4 に示す。 各文法は四則演算を表現でき、0 以外から始まる数字を使用できるような文法になっている。文法 2 から文法 4 の括弧を持つような文法は、一段のみ括弧がネストするような文法になっている。文法 3 と文法 4 では、小文字のアルファベットのみからなる変数名を使用することができ、$x = 1$ のような代入文を記述できるようになっている。また、文法 4 では if 文が追加されており、“if” というキーワードの直後に $Expr$ から生成可能な文字列を括弧で囲って記述し、その後ろに $StmtList$ から生成可能な

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr \quad \text{“}+\text{”} \quad Term \\
Expr & \rightarrow & Expr \quad \text{“}-\text{”} \quad Term \\
Expr & \rightarrow & Term \\
Term & \rightarrow & Term \quad \text{“}*\text{”} \quad Factor \\
Term & \rightarrow & Term \quad \text{“/”} \quad Factor \\
Term & \rightarrow & Factor \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}+\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}-\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}*\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“/”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & Num \\
Num & \rightarrow & 0 \\
Num & \rightarrow & [1-9][0-9]*
\end{array}
$$

図 4.2. 文法 2(四則演算と括弧を持つ文法) の生成規則

$$
\begin{array}{lcl}
Program & \rightarrow & Stmt \\
Expr & \rightarrow & Expr \quad \text{“}+\text{”} \quad Term \\
Expr & \rightarrow & Expr \quad \text{“}-\text{”} \quad Term \\
Expr & \rightarrow & Term \\
Term & \rightarrow & Term \quad \text{“}*\text{”} \quad Factor \\
Term & \rightarrow & Term \quad \text{“/”} \quad Factor \\
Term & \rightarrow & Factor \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}+\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}-\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}*\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“/”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & Num \\
Factor & \rightarrow & Var \\
Num & \rightarrow & 0 \\
Num & \rightarrow & [1-9][0-9]* \\
Var & \rightarrow & [a-z]+ \\
Stmt & \rightarrow & Var \quad \text{“}=\text{”} \quad Expr
\end{array}
$$

図 4.3. 文法 3(四則演算と括弧と変数名、代入文を持つ文法) の生成規則

文字列を中括弧 “{}” で囲って記述するような構文になっている。

4.2 生成ツールを用いたデータセットの生成

本実験のために作成した文字列生成ツールでは、ユーザが文法を図 4.5 のような DSL で記述し、その文法を基にランダムに木を作成する。ここで作成された木を辿ることで木から最終

$$
\begin{array}{lcl}
Program & \rightarrow & StmtList \\
Expr & \rightarrow & Expr \quad \text{“}+\text{”} \quad Term \\
Expr & \rightarrow & Expr \quad \text{“}-\text{”} \quad Term \\
Expr & \rightarrow & Term \\
Term & \rightarrow & Term \quad \text{“}*\text{”} \quad Factor \\
Term & \rightarrow & Term \quad \text{“}/\text{”} \quad Factor \\
Term & \rightarrow & Factor \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}+\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}-\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}*\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & \text{“(”} \quad Num \quad \text{“}/\text{”} \quad Num \quad \text{“)”} \\
Factor & \rightarrow & Num \\
Factor & \rightarrow & Var \\
Num & \rightarrow & 0 \\
Num & \rightarrow & [1-9][0-9]* \\
Var & \rightarrow & [a-z]+ \\
Stmt & \rightarrow & Var \quad \text{“}=\text{”} \quad Expr \quad \text{“;”} \\
Stmt & \rightarrow & IfStmt \\
StmtList & \rightarrow & Stmt \quad StmtList \\
StmtList & \rightarrow & Stmt \\
IfStmt & \rightarrow & \text{“}if\text{”} \quad \text{“(”} \quad Expr \quad \text{“)”} \quad \text{“\{”} \quad StmtList \quad \text{“\}”}
\end{array}
$$

図 4.4. 文法 4(if 文、四則演算、括弧、変数名、代入文を持つ文法) の生成規則

```
rules:
  Stmt -> Var "=" Num
  Num -> randomInt
  Var -> randomStr
```

図 4.5. 文法を定義する DSL

的な文字列を出力する。例として、図 4.5 からは $a = 1$ や $abc = 123$ のような文字列が生成される。ここで、randomInt と randomStr はあらかじめ定義してある関数の名前であり、それぞれランダムに数値リテラルと文字列リテラルを生成する関数である。これらの関数は文字列の生成時に毎回呼び出される。プログラムの実行時には、rules ブロックの先頭に記述された生成規則の左辺の非終端記号を開始記号とみなし、宣言された文法に従う文字列をランダムに生成する。各終端記号の間には 1 個の空白文字を入れて出力が行われる。

実際に、実験に用いるデータセットを作成するために記述した生成規則の宣言部を図 4.6、4.7、4.8、4.9 に示す。 また、これらのコードを実行することで実際に生成した文字列の例を図 4.10、4.11、4.12、4.13 に示す。 図 4.13 からもわかるように、文法 4 の生成規則の一つで

```
rules:
  Expr -> Expr "+[add]" Term
  Expr -> Expr "-[sub]" Term
  Expr -> Term
  Term -> Term "*[mul]" Num
  Term -> Term "/[div]" Num
  Term -> Num
  Num -> randomInt
```

図 4.6. データセット 1 を生成する文法の宣言部分

```
rules:
  Expr -> Expr "+[add]" Term
  Expr -> Expr "-[sub]" Term
  Expr -> Term
  Term -> Term "*[mul]" Factor
  Term -> Term "/[div]" Factor
  Term -> Factor
  Factor -> "([lp]" Num "+[add]" Num ")[rp]"
  Factor -> "([lp]" Num "-[sub]" Num ")[rp]"
  Factor -> "([lp]" Num "*[mul]" Num ")[rp]"
  Factor -> "([lp]" Num "/[div]" Num ")[rp]"
  Factor -> Num
  Num -> randomInt
```

図 4.7. データセット 2 を生成する文法の宣言部分

ある $IfStmt \rightarrow$ “if” “(” $Expr$ “)” “{” $StmtList$ “}” が非終端記号 $StmtList$ を含むため、データセット 4 は他のデータセットよりも長い文字列を多く含む傾向にあった。本システムでは対象プログラミング言語の字句解析器の存在を仮定しているため、データセット用の字句解析器の作成を行なった。データセット用の字句解析を単純化するために、出力文字列の終端記号には角括弧 (“[”,“]”) を用いてトークンの型を記述している。これにより、字句解析器は空白によるトークンの分割と角括弧で囲われたトークンの型の読み出し程度の処理のみを行なっている。例として、図 4.10 の文字列を字句解析器に与えた場合は、以下のような文字列が与えられたのと同様に扱われる。

```
727 + 930 + 923 + 388 / 735
```

```
rules:
  Program -> Stmt
  Expr -> Expr "+[add]" Term
  Expr -> Expr "-[sub]" Term
  Expr -> Term
  Term -> Term "*[mul]" Factor
  Term -> Term "/[div]" Factor
  Term -> Factor
  Factor -> "([lp]" Num "+[add]" Num ")[rp]"
  Factor -> "([lp]" Num "-[sub]" Num ")[rp]"
  Factor -> "([lp]" Num "*[mul]" Num ")[rp]"
  Factor -> "([lp]" Num "/[div]" Num ")[rp]"
  Factor -> Num
  Factor -> Var
  Num -> randomInt
  Var -> randomStr
  Stmt -> Var "=[eq]" Expr
```

図 4.8. データセット 3 を生成する文法の宣言部分

この時、通常の字句解析器と同様にトークンの型は記録される。

4.3　実験内容

今回の実験では、前述の四種類のデータセットに対して本システムを用いて文法構造の獲得を行う。そして、評価データに対して、獲得した文法で構文解析を行うことで定量的な評価を行う。評価データについては、各データセットについてそのデータセットの文法として正しいデータと、反対に文法として誤りのデータをそれぞれ作成した。ここで、誤りのデータは正しい文法の生成規則の左辺を変更してソースコードを生成することによって作成した。謝りのデータでは、正しいデータセットには現れないようなトークンの並びが現れるようになっている。表 4.3 に本実験のために生成した学習データ数、文法として正しい評価データ数、文法として誤りの評価データ数を示す。また、Skip-gram モデルを用いて各トークンにベクトル表現を与える際に使用したパラメータを表 4.4 に示した。本システムでは、構文解析時に各生成規則を用いるかどうかを判断するためにパラメータとして閾値を与える。3 章で述べたように、ここで与えた閾値よりも尤度が低いような生成規則は構文解析時に使用しないものとしている。本実験では、閾値を [0.0, 0.5] の範囲を 0.1 刻みで変更させながら、文法として正しい評価データのうちどれだけ構文解析に成功したかを数え上げた。また、同様に文法として誤りの評価データに対しても構文解析を行い、どれだけ成功したか数え上げを行なった。

```
rules:
  Program -> StmtList
  Expr -> Expr "+[add]" Term
  Expr -> Expr "-[sub]" Term
  Expr -> Term
  Term -> Term "*[mul]" Factor
  Term -> Term "/[div]" Factor
  Term -> Factor
  Factor -> "([lp]" Num "+[add]" Num ")[rp]"
  Factor -> "([lp]" Num "-[sub]" Num ")[rp]"
  Factor -> "([lp]" Num "*[mul]" Num ")[rp]"
  Factor -> "([lp]" Num "/[div]" Num ")[rp]"
  Factor -> Num
  Factor -> Var
  Num -> randomInt
  Var -> randomStr
  Stmt -> Var "=[eq]" Expr ";[semicolon]"
  Stmt -> IfStmt
  StmtList -> Stmt StmtList
  StmtList -> Stmt
  IfStmt -> "if[if_keyword]" "([lp]" Expr ")[rp]" "{[lb]"
      StmtList "}[rb]"
```

図 4.9. データセット 4 を生成する文法の宣言部分

```
727[num] +[add] 930[num] +[add] 923[num] +[add] 388[num] /[div]
    735[num]
```

図 4.10. 図 4.6 から生成される文字列の例

4.4 実験結果と考察

本節でははじめに、学習を行う際に得た「トークンの組」のクラスタリング結果を示す。また、学習によって得られた文法を用いて構文解析を行なった結果を用いて Accuracy、Precision、Recall、F1 score を計算を行なった。本実験の学習において、Skip-gram によるベクトル割り当てとクラスタリングの結果得られたトークンの組のクラスタを図 4.14、4.15、4.16、4.17 に示す。 データセット 3 とデータセット 4 についてはクラスタ数が多く、出力が膨大になって

```
([lp] 862[num] *[mul] 567[num] )[rp] *[mul] ([lp] 783[num] -[sub
    ] 506[num] )[rp]
```

図 4.11. 図 4.7 から生成される文字列の例

```
xwmks[var] =[eq] 939[num] /[div] ([lp] 464[num] *[mul] 883[num]
    )[rp] *[mul] 722[num]
```

図 4.12. 図 4.8 から生成される文字列の例

```
if[if_keyword] ([lp] 593[num] -[sub] 968[num] /[div] ([lp] 394[
    num] -[sub] 756[num] )[rp] *[mul] 605[num] /[div] ([lp] 73[
    num] -[sub] 809[num] )[rp] /[div] f[var] )[rp] {[lb] xek[var]
     =[eq] ([lp] 515[num] +[add] 793[num] )[rp] +[add] 40[num] +[
    add] ([lp] 658[num] /[div] 19[num] )[rp] /[div] ([lp] 687[num
    ] /[div] 877[num] )[rp] -[sub] ([lp] 644[num] +[add] 411[num]
     )[rp] +[add] ([lp] 174[num] *[mul] 613[num] )[rp] /[div]
    agsjww[var] -[sub] ([lp] 798[num] /[div] 53[num] )[rp] -[sub]
     ([lp] 64[num] *[mul] 923[num] )[rp] +[add] 115[num] ;[
    semicolon] }[rb]
```

図 4.13. 図 4.9 から生成される文字列の例

表 4.3. 各データセットのデータ数

	学習データ数	評価データ数 (正)	評価データ数 (誤)
文法 1	10000	5000	5000
文法 2	10000	5000	5000
文法 3	10000	5000	5000
文法 4	10000	5000	5000

しまったため一部抜粋している。各行の “[]” で囲われた要素同士は同じクラスタに属することを示し、各要素の “()” で囲われた組はトークンの組を表す。(a, b) とある場合は、トークン a の「左側の文脈のみを考慮したベクトル表現」とトークン b の「右側の文脈のみを考慮したベクトル表現」を連結させたベクトル表現を用いてクラスタリングを行なっている。また、左端が a であり右端が b であるようなトークン列に対して、そのトークン列から還元可能な非終端記号は (a, b) が属するクラスタと同値となる。図 4.16 に多く見られるような num や var は、ランダムに生成された数値や文字列のうち同一のものが少なかったためにトークンの型で置換された値である。図 4.17 における if はキーワードとして生成規則の中で直接記述をした

表 4.4. ベクトル表現の獲得時に使用したパラメータ

ウィンドウサイズ	1
埋め込み次元数	100
バッチサイズ	128
エポック数	100
学習率	0.050
ネガティヴサンプル数	5

```
Cluster 0: [ (+, +) ]
Cluster 1: [ (+, −) ]
Cluster 2: [ (/, num), (/, ∗), (/, +), (/, /), (/, −) ]
Cluster 3: [ (num, /) ]
Cluster 4: [ (−, −) ]
Cluster 5: [ (∗, ∗), (∗, +), (∗, −) ]
Cluster 6: [ (+, num) ]
Cluster 7: [ (+, /) ]
Cluster 8: [ (−, num) ]
Cluster 9: [ (−, /) ]
Cluster 10: [ (num, num), (num, ∗), (num, +) ]
Cluster 11: [ (+, ∗) ]
Cluster 12: [ (−, ∗) ]
Cluster 13: [ (−, +) ]
Cluster 14: [ (∗, num) ]
Cluster 15: [ (∗, /) ]
Cluster 16: [ (num, −) ]
```

図 4.14. データセット 1 から得た「トークンの組」のクラスタ

ものである。また、if 以外の文字列の多くはランダムに生成された文字列のうち出現ファイル数が多かったことでトークンの型で置換されずに残った文字列である。

獲得した文法を用いて構文解析を行う実験では、構文解析の結果を用いて Accuracy、Precision、Recall、F1 score を計算した。ここでは、データの正誤と構文解析結果の組み合わせを以下のように定めて評価指標の計算を行なった。

True Positive　正しい文法から生成されたデータに対して、構文解析に成功した場合
True Negative　誤りである文法から生成されたデータに対して、構文解析に失敗した場合
False Positive　誤りである文法から生成されたデータに対して、構文解析に成功した場合
False Negative　正しい文法から生成されたデータに対して、構文解析に失敗した場合

```
Cluster 0: [ (), −) ]
Cluster 1: [ (−, /) ]
Cluster 2: [ (num, num), (num, )) ]
Cluster 3: [ (), num), (), )) ]
Cluster 4: [ ((, (), (num, () ]
Cluster 5: [ (−, num), (/, num), (∗, num) ]
Cluster 6: [ (), ∗) ]
Cluster 7: [ (+, num) ]
Cluster 8: [ (), /) ]
Cluster 9: [ (−, +), (/, +), (∗, +) ]
Cluster 10: [ ((, +), (num, +) ]
Cluster 11: [ ((, ∗), (num, ∗) ]
Cluster 12: [ (−, −), (/, −), (∗, −), (+, −) ]
Cluster 13: [ ((, −), (num, −) ]
Cluster 14: [ (+, )) ]
Cluster 15: [ (−, () ]
Cluster 16: [ (/, (), (∗, (), (+, () ]
Cluster 17: [ (/, /), (∗, /), (+, /) ]
Cluster 18: [ ((, num), ((, )) ]
Cluster 19: [ (−, ∗), (/, ∗), (∗, ∗), (+, ∗) ]
Cluster 20: [ (), (), (), +) ]
Cluster 21: [ (+, +) ]
Cluster 22: [ (−, )), (/, )), (∗, )) ]
Cluster 23: [ ((, /), (num, /) ]
```

図 4.15. データセット 2 から得た「トークンの組」のクラスタ

今回の場合は、Precision は「構文解析に成功した場合のうち、正しい文法のデータに対して成功をした割合」を表す。また、Recall は「正しい文法から生成されたデータのうち、構文解析に成功した割合」を表す。実験によって得られた各評価指標の値をデータセットごとに表 4.5、4.6、4.7、4.8 に示す。

図 4.8 のデータセット 4 に対する実験結果で、閾値が 0.0 の場合にも Recall が 1.00 になっていない理由は、学習データ内で一度も登場しないような生成規則が構文解析に必要となるデータが評価データセット内にあるためである。本手法では、学習データを元に初期の生成規則を定めて、その生成規則に対してのみ適用確率を与える。そのため、学習データ内に登場しない非終端記号の組を使用するような構文解析は閾値を 0.0 にしても行うことができないようになっている。

コーディングスタイル検査器等のアプリケーションへの応用を考える場合、閾値の決定はア

```
Cluster 0: [ (*, num), (*, )) ]
Cluster 1: [ (/, ()), (-, ()), (*, () ]
Cluster 2: [ (/, +), (-, +), (*, +), (+, +) ]
Cluster 3: [ (), num), (), )) ]
Cluster 4: [ (), var) ]
Cluster 5: [ (num, num), (num, )) ]
Cluster 6: [ (=, num), (=, )) ]
Cluster 7: [ (var, /), (num, /), ((, /) ]
Cluster 8: [ (var, *), ((, *) ]
Cluster 9: [ (-, /) ]
Cluster 10: [ (=, =), (=, () ]

...

Cluster 28: [ (+, () ]
Cluster 29: [ (var, ()), (num, ()), ((, () ]
Cluster 30: [ (var, -), (num, -), ((, -) ]
Cluster 31: [ (=, var) ]
Cluster 32: [ (/, -), (-, -), (*, -) ]
Cluster 33: [ (num, *) ]
Cluster 34: [ (), +) ]
Cluster 35: [ (/, *), (*, *), (+, *) ]
Cluster 36: [ (=, *) ]
Cluster 37: [ (+, var) ]
Cluster 38: [ (var, =), ((, =) ]
Cluster 39: [ (), /) ]
```

図 4.16. データセット 3 から得た「トークンの組」のクラスタ (一部抜粋)

プリケーションを実運用する上で重要な要素となる。実験結果からも、閾値による性能の変化が大きいことがわかる。本手法では、文法の獲得時ではなく構文解析時に閾値を指定する。これにより、閾値を学習時に与えるようなモデルと比べるとユーザが閾値の選定を短い時間で行うことができるという利点があると考えられる。

文法 1、文法 2、文法 3 から生成したデータセットに対する実験では、適切な閾値を与えることができれば良い性能を出すことができていることが実験結果からわかる。しかし、文法 4 の場合においては他のデータセットの場合と比べて Recall が大きく減少している。文法 4 は、他の文法に加えて if 文が追加されていることが大きな違いである。if 文が追加されたことが

```
Cluster 0: [ (a, *) ]
Cluster 1: [ (-, z), (), z), (*, z), (+, z), (/, z) ]
Cluster 2: [ (x, {) ]
Cluster 3: [ (a, /) ]
Cluster 4: [ ({, ()， ({, -), ({, =) ]
Cluster 5: [ (f, n) ]
Cluster 6: [ (n, y) ]
Cluster 7: [ (d, i) ]
Cluster 8: [ (v, if) ]
Cluster 9: [ (if, d), (}, d) ]
Cluster 10: [ (q, var), (q, w), (q, v), (q, r) ]

    ...

Cluster 749: [ (g, q) ]
Cluster 750: [ (r, s) ]
Cluster 751: [ (j, f) ]
Cluster 752: [ (t, +) ]
Cluster 753: [ (c, }) ]
Cluster 754: [ (p, s) ]
Cluster 755: [ (z, f) ]
Cluster 756: [ (v, b) ]
Cluster 757: [ (c, q) ]
Cluster 758: [ (i, n) ]
Cluster 759: [ (x, f) ]
```

図 4.17. データセット 4 から得た「トークンの組」のクラスタ (一部抜粋)

性能に大きな影響を与えた理由として、「括弧 “()” の文法上の意味が重複している」ということが考えられる。文法 4 では、演算式中で四則演算を囲うような場合に用いられる括弧と、“if” というキーワードの直後に現れる括弧が存在する。本手法では、各トークンに対して左右のベクトル表現を一組だけ与えるため、これらの括弧に対しても両方の文脈を考慮した単一のベクトル表現のみが与えられている。また、図 4.17 のクラスタリング結果からわかるように、データセット 4 ではトークンの型で置換されずに元のトークンのままベクトル表現が与えられたものが多いようである。これにより非終端記号の数が増加してしまったことが、性能を落とす原因となっている可能性も考えられる。

表 4.5. データセット 1 に対する実験結果

閾値	Accuracy	Precision	Recall	F1 score
0.0	0.58	0.54	1.00	0.71
0.1	0.67	0.60	1.00	0.75
0.2	0.60	0.57	0.78	0.66
0.3	0.56	0.64	0.28	0.39
0.4	0.60	0.79	0.28	0.41
0.5	0.56	1.00	0.11	0.20

表 4.6. データセット 2 に対する実験結果

閾値	Accuracy	Precision	Recall	F1 score
0.0	0.80	0.71	1.00	0.83
0.1	0.78	0.71	0.94	0.81
0.2	0.73	0.71	0.78	0.74
0.3	0.69	0.72	0.61	0.66
0.4	0.64	0.74	0.43	0.54
0.5	0.65	0.84	0.38	0.52

表 4.7. データセット 3 に対する実験結果

閾値	Accuracy	Precision	Recall	F1 score
0.0	0.89	0.83	1.00	0.91
0.1	0.99	0.99	1.00	1.00
0.2	0.99	0.99	1.00	1.00
0.3	0.99	0.99	1.00	1.00
0.4	0.94	0.99	0.88	0.93
0.5	0.79	0.99	0.59	0.74

4.4.1 トークンの三つ組を用いたベクトル表現の割り当て

実験結果から、文法上複数の意味を持つようなトークンが現れる場合に精度が下がってしまうということがわかった。これは、トークンに対して与えるベクトル表現が単一 (左右で一つずつ) であるために、トークンが属するクラスタが一つとなってしまうことが問題であると考えられる。そこで、解決方法としてトークンの三つ組を使用する方法を検討した。例えば、$if(x)\{\dots\}$ や $a = (b + c)$ といった文字列がデータセットに現れる場合、各トークンに左右のベクトル表現を与えるのではなく、(“if”, “(”, “Var”) や (“ $=$ ”, “(”, “Var”) といった連続す

表 4.8. データセット 4 に対する実験結果

閾値	Accuracy	Precision	Recall	F1 score
0.0	0.54	0.84	0.41	0.55
0.1	0.54	0.85	0.40	0.55
0.2	0.50	0.93	0.39	0.54
0.3	0.47	0.94	0.35	0.51
0.4	0.42	0.93	0.29	0.45
0.5	0.40	0.93	0.23	0.38

るトークンの三つ組に対して左右のベクトル表現を与えることを考える。ここで、各トークンは本システムと同様に字句解析結果を用いた置換を行なっているため x, a, b, c は Var に置換されているものとする。また、ファイルの最初に現れるトークンの左側のトークンと最後に現れるトークンの右側のトークンは存在しないため、$None$ として扱う。これにより、「左側に if が現れて右側に Var が現れるような “(”」や「左側に $=$ が現れて右側に Var が現れるような “(”」に対して異なるベクトル表現を与えられるため、上記のような問題は回避できると考えられる。しかし、この方法では考慮するべきトークンの三つ組の数が膨大になってしまうことや、ファイルの最初や最後に現れるようなトークンの三つ組に対して生成規則を学習しづらいという問題があることがわかった。

第 5 章

まとめと今後の課題

5.1 まとめ

本論文では、既存のソースコード群からのプログラミング言語の文法獲得に向けて非終端記号の獲得を行う手法を提案した。本手法では、はじめに Skip-gram モデルによって与えたトークン列に対するベクトル表現を基に「生成規則の集合」の初期値の決定を行なった。また、前処理で得られた「生成規則の集合」の初期値から開始し、Inside-outside アルゴリズムによって生成規則の適用確率を更新することで最終的な文法を獲得を行なった。本手法の有用性を検証するために、単純な文法からランダムに生成したデータセットに対して、提案手法を用いて文法獲得を試みる実験を行なった。データセットから提案手法によって生成した文法を用いて、評価データの構文解析を行なったところ、四則演算のみの文法や四則演算に代入文を追加したような文法に対しては高い性能を出すことに成功した。しかし if 文を追加したデータセットに対しては実用に耐え得る性能を出すことはできていないようであった。これは、「演算に用いられる括弧」と「if 文の直後に用いられる括弧」の文法上意味の異なる二種類の括弧が現れるという点に原因があるのではないかと考えられる。

5.2 提案手法の制限

本手法の応用や実用を考える際に、大きく二点の制限が考えられる。

一点目に、文法を獲得する対象の字句解析がわかっていることが前提であるということが挙げられる。本手法では、学習データの各ファイルの文字列をトークン列に分割する際に対象プログラミング言語の字句解析器を使用する。そのため、本システムをそのまま使用するためにはプログラミング言語の字句解析器が公開されている必要がある。公開されている字句解析器の例として、Ruby 言語では Ripper という字句解析ライブラリが標準ライブラリとして提供されている。また、Java 言語の構文解析ライブラリとして JavaParser[*1]などもオープンソースソフトウェアとして公開されているため、これらのツールで提供されている字句解析器は用いることができる。しかし、そのような字句解析器が存在しない言語やもともと文法がないよ

[*1] https://github.com/javaparser/javaparser

うな文字列への応用を考える際には別途字句解析器を用意する必要がある。

二点目に、トークン列への分割時に失われる情報やプログラムの実行時の情報を用いるタスクには使用できないということが挙げられる。2 章では、プログラミング言語の文法獲得の応用例としてコーディングスタイル検査器を挙げた。コーディングスタイルには、命名規則やトークン列の並びから決定できるようなルールの他に、空白の数などフォーマッティングに関するルールも存在する。本システムを適用する場合、字句解析器によってトークン列に分割する時点で空白文字や改行文字は取り除いてしまうため、コーディングスタイルの中でフォーマッティングに関するスタイルについての検査は愚直には行うことができない。解決方法として、字句解析を行う時点で空白文字や改行文字を特別なトークンとして残しておくなどの方法が考えられるが、この方法による影響は本論文では未検証である。また、本論文で既に述べたように、今回の手法で獲得を試みる文法は必ずしも対象のプログラミング言語で既に定義されている文法であるとは限らない。一般に、プログラミング言語では構文解析によってソースコードに対応する抽象構文木を生成し、その抽象構文木を利用してプログラムの実行を行う。しかし、本手法で得られる文法構造から生成される構文木は、プログラムの実行のために用いられる構文木とは異なるため、プログラムの実行に関するタスクには応用が難しいと考えられる。

5.3　今後の課題

本研究では、プログラミング言語の文法獲得に向けて、言語に対して汎用的な手法を検討してきた。確率的文脈自由文法の形式で獲得することなどを除くと、生成規則に対する強い仮定や制限は極力避けた手法となっている。しかし、多くのプログラミング言語に共通するような仮定を設けることで獲得する文法に対する制限を加えることも可能であると考えられる。例として、括弧 “()” や中括弧 “{}” のような特定の文字の組は一般に対応関係を持つ。新たなプログラミング言語を習得する際などは、人間の場合、暗黙のうちにこれらの対応関係を仮定してソースコードを読もうとすると思われる。同様に、プログラミング言語の文法を設計する開発者もそのような文法構造を定義するのが一般的である。文法獲得時に、このような対応関係を抜き出す処理を加えて、その区間が必ず一つの非終端記号に変換される (部分木が構築される) ような学習をさせる方法も一例として考えられる。括弧の他にも、多くのプログラミング言語で共通するものとして、“if”・“for”・“class” などのキーワードが定義されているということがある。本手法では、出現ファイル数による置換を行うことによって、通常はこれらのキーワードが残る (トークンごとに区別される) ようになっているが、あらかじめキーワードを与えた上で文法の獲得を行うというアプローチも考えられる。また、これらのキーワードを元にした生成規則の決定方法も考え得るかもしれない。「どのようなアプリケーションに応用するのか」や「どのような構造を獲得したいか」によってこれらの仮定をおくかどうかは議論をする必要がある。今回の研究では、多くのツールに応用可能な手法を目指したため、汎用的なアプローチとなっているが、将来的に上記のような生成規則に対する制限を加えた手法も検討を行う必要がある。

本手法では、文脈自由文法を仮定して文法の獲得を行なった。文脈自由文法の他にも、より制限のある LL 文法や LR 文法に対するアプローチを考えると、異なる性能が得られる可能性がある。これらの文法は文脈自由文法よりも表現力は劣るがプログラミング言語の文法としては十分である場合も多い。今回作成したシステムの文法獲得部分は、文脈自由文法であることに強く依存した手法を用いたため、これらの文法を仮定する場合は本手法とは大きく異なる手法を検討する必要があるだろう。本研究では、これらの文法に対しての手法検討は行なっていない。

本研究の提案手法は、単純な文法に対しては良い性能を出したが、if 文を取り入れたデータセットにおいては良い性能を出すことはできていないようであった。またこの結果から、実際に広く用いられているプログラミング言語の文法に対しても、実用に耐え得る性能を出すことは現状は困難であると考えられる。前述のようなアプローチ等を検討することで、実用可能な段階まで精度の向上をさせる必要があるだろう。

参考文献

[1] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, Vol. 2, No. 3, pp. 113–124, Sep. 1956.

[2] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA†, 1965.

[3] D. H. Younger. Context-free language processing in time n^3. In *7th Annual Symposium on Switching and Automata Theory (swat 1966)(FOCS)*, Vol. 00, pp. 7–20, 10 1966.

[4] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, Vol. 39, No. 1, pp. 1–38, 1977.

[5] J. K. Baker. Trainable grammars for speech recognition. In D. H. Klatt and J. J. Wolf, editors, *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pp. 547–550, 1979.

[6] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, Berlin, Heidelberg, 1996.

[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, Vol. abs/1301.3781, , 2013.

[8] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pp. 3111–3119, USA, 2013. Curran Associates Inc.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, Vol. abs/1409.0473, , 2014.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *CoRR*, Vol. abs/1510.07211, , 2015.

[12] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W.

Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *CoRR*, Vol. abs/1603.06744, , 2016.

[13] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pp. 152–162, New York, NY, USA, 2018. ACM.

[14] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pp. 933–944, New York, NY, USA, 2018. ACM.

[15] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pp. 87–98, New York, NY, USA, 2016. ACM.

[16] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *CoRR*, Vol. abs/1805.11683, , 2018.

[17] Srinivas S. Kruthiventi, Kumar Ayush, and R. Venkatesh Babu. Deepfix: A fully convolutional neural network for predicting human eye fixations. *CoRR*, Vol. abs/1510.02927, , 2015.

[18] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, Vol. 31, No. 15, pp. 1395–1438, December 2001.

[19] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, Vol. 45, No. 11, pp. 2673–2681, November 1997.

[20] Yuki Kobayashi. Error detection of coding style based on token sequence likelihood calculation. Master's thesis, The University of Tokyo, 2017.

[21] Diptikalyan Saha and Vishal Narula. Gramin: A system for incremental learning of programming language grammars. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pp. 185–194, New York, NY, USA, 2011. ACM.

[22] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[23] Dan Pelleg and Andrew W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pp. 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

謝辞

本研究を進めるにあたり、研究方針の相談や論文・発表についての助言など幅広く指導してくださった千葉 滋教授に心より感謝いたします。また、発表の指導や研究に関する助言をいただいた穐山 空道助教に感謝いたします。そして、研究について議論を交わしたり、助言をしていただいた山崎 徹郎氏、中丸 智貴氏、Daniel Perez 氏、市川 和央氏、奥田 勝己氏に感謝いたします。加えて、研究生活を通じてご支援くださった佐藤 輝年氏と西田 秀之氏にも感謝いたします。