東京大学
情報理工学系研究科 創造情報学専攻
修士論文

トークン列の尤度計算に基づいた コーディングスタイルの誤り検知

Error Detection of Coding Style
based on Token Sequence Likelihood Calculation

小林 佑樹
Yuki Kobayashi

指導教員　千葉 滋 教授

2017 年 8 月

概要

Ruby を代表とするような、多様なコーディングスタイルを許容する言語ではコーディングスタイルの組織内での統一が重要とされている。その一方で従来はコーディングスタイルをルールとして手動で明文化し、カスタマイズする手法しか存在しなかったため、組織毎のルールのカスタマイズ作業のコストが課題であった。

本研究では、コーディングスタイルを規則ではなく、トークン列の尤度で表現し、コーディングスタイルの適合評価を自動で行う手法の提案を行う。スタイルに合致するプログラム群から導かれる尤度関数を、そのスタイルとみなすことでコーディングスタイルの自動抽出を行う。導いた尤度関数を用いて検査対象のプログラム中で尤度を計算し、尤度の低いトークン列をコーディングスタイルに不適合なものとして検出する。さらに本手法を使ったシステムを実際に構築し、Ruby の静的コード解析ツールの Rubocop と本手法を使ったシステムでコーディングスタイルの適合評価の精度を比較した。

Abstract

It is important for teams and organizations to consistently use the same coding style, especially with programming languages that allows various coding styles. But, the coding style must be clarified and customized manually.

In our research, we represent coding style not by conventions but by likelihood and evaluate the coding style of source code automatically. We derive a likelihood function that represents the coding style from a group of programs which is written in a coding style that is considered to be correct by the team. We calculate the likelihood of token sequences with the likelihood function which we derived and detect tokens which have a low likelihood of being the wrong coding style. We built the system using this approach and compared the precision of this system with the precision of Rubocop which performs a static analysis of Ruby source code.

目次

第 1 章

はじめに

インターネットが普及し，誰もが生活する上で常時インターネットに接続するようになった現代において，インターネットを利用するためのソフトウェアやインターネット上で利用するソフトウェアコンテンツやアプリケーションの需要も常に増え続けている．需要の増加に伴い，そういったソフトウェアコンテンツやアプリケーションを提供している企業も増え，利用するユーザー数の増加に伴い，求められるソフトウェアのクオリティやスケールも年々増えてきている．開発するソフトウェアのクオリティやスケールが増えれば，一つのソフトウェア開発に同時に携わる開発チームや組織の人数も当然増えてくる．しかし一つのソフトウェアの開発を多くの開発者で開発していくのは小規模なチームや個人で開発するのに比べて，格段に開発の難易度が増してしまう．ソフトウェアの仕様やコーディングスタイルのチーム内での共有，他人が書いたソースコードの改良や修正，開発スケジュールの調整や開発の計画の常時見直しなど人数が増えることによるコミュニケーションコストによって開発のスピードや質が落ちてしまう．単純に人数が増えればチームの開発能力があがるということではないため，複数の開発者でソフトウェア開発をする上での効率のよい処置をとらなければ，最悪人数が増える前よりも開発結果が悪くなってしまうことも有り得るのである．そのため常にソフトウェア開発をする開発者がどうすればより効率よく開発できるかがソフトウェア工学の分野において重要なテーマであり，近年はそれに加えて大規模な開発チームでの効率の良い開発手法もより重要な問題になっている．

大規模なソフトウェア開発においてはコーディングスタイルのチーム内での統一も重要な課題の一つである．コーディングスタイルとはプログラムの表現方法を指す．コーディングスタイルが統一されているソースコードやプロジェクトは基本的にそのコーディングスタイルを知っている開発者ならプログラムの構造や処理内容がすぐに分かるようになっているため，結果的にプログラムの修正や拡張なども素早く正しく行え，開発効率が格段にあがる．一方でコーディングスタイルがソースコードによって統一されてなかったり，そもそもコーディングスタイルが守られていないソースコードはプログラムの内容を読み解くのに時間がかかってしまう．他人が書いたコードであれば尚更時間がかかる上に，間違った解釈をさせてしまい，結果的にバグや冗長なコードが混入してしまう事態に陥ってしまうことが多々ある．

近年は Ruby を代表とするような文法が他の言語に比べて，かなり柔軟で様々なプログラム

の書き方を許容するプログラミング言語が登場し始め，実際にソフトウェア開発の現場で使われるようになったため，ますますコーディングスタイルを統一しなくては開発効率に影響してしまうのである．また基本的にコーディングスタイルは各開発者のポリシーやこれまでの開発経験による慣れ，考え方によるところが大きい．そのため，たとえ経験の豊富な開発者でもある組織に新しく加入し，その組織の採用するコーディングスタイルが開発者がこれまで利用してきたコーディングスタイルと違うものであれば，その組織にとってコーディングスタイル的に間違ったコードを書いてしまったりすることもある．すなわちコーディングスタイルの問題はどんな開発者にも共通して起こりうるのである．そのため現在大規模なソフトウェアを開発しているチームや企業の中にはこういったコーディングスタイルの統一を行うために幾つかの施策を試みているものもある．

例えば Ruby on Rails を使って，多くのユーザーから利用されるレシピウェブサイトを開発，運営しているクックパッド株式会社では Ruby の書き方を始め，Objective-C，Swift，Java，CoffeeScript，Python のコーディングスタイルについて開発チームの中で採用しているコーディングスタイルをまとめ，ルールとして開発チーム内で共有している．[1] また一般的な企業でも開発チームに新しいメンバーが加入した際には，開発チームの中からメンターを選出し，そのメンターが新メンバーに付き添いながらコーディングスタイルなどの教育を行うことが多い．しかしこれらの方法は開発チームに多くのコストがかかってしまう．まずコーディングスタイルのルールを共有するためのドキュメントを作成しなくてはならない．またそれらのドキュメントは随時言語のアップデートの際に修正する必要がある．また開発しているメンバーもそのコーディングスタイルに慣れるまではそのドキュメントを見ながらの開発になるため，開発スピードに遅れがでてしまう．またメンターによる教育もメンターに選出されたメンバーは開発する時間が教育に割かれてしまうため，開発できる時間が減ってしまう．またメンターに選出されるメンバーは経験豊富で開発力のあるメンバーであることが多く，そのようなメンバーの開発時間が減るということはそれ自体が開発チーム全体の開発スピードの低下に繋がってしまう．

ソースコードのコーディングスタイルを自動で検査する静的解析ツールもいくつかの言語では開発されており，それらを使っている開発することもできる．特に様々なコーディングスタイルを許容する Ruby のコーディングスタイルを静的解析する Rubocop[2] というツールは Ruby の柔軟な文法に耐えうるようにユーザーが独自にルールをカスタマイズできるような設定ファイルが用意されている．しかしこういったルールを独自にカスタマイズして利用するツールはユーザーがルールを手動でカスタマイズしなくてはならないコストがある．ルール化する場合にはコーディングスタイルを明文化する必要があるため，チームのコーディングスタイルによってはルールとして用意されていないものなどが存在する場合があり，そのようなルールはこういった静的解析ツールでは検査することが難しくなる．

一方で近年はニューラルネットワークの開発や GPU の普及によって機械学習の研究分野の発展がめざましく，多くの分野での機械学習の活用が研究されている．そんな中，その活用先としてソフトウェアの研究が挙げられることがある．ソフトウェア工学は従来までルールベースでのアプローチが基本であったが，機械学習の精度の向上などが起因して，機械学習を応用

することで従来のルールベースでは解くことができなかった研究課題の新たなアプローチとして統計的な機械学習のアプローチが利用されることが増え始めている．例えば DeepFix[3] は C 言語のコンパイルエラーをニューラルネットワークとコンパイラを利用して自動で修正する研究を行い，実際に Sequence-to-Sequence というモデルを使うことで一部の C 言語プログラムに存在していたエラーを完全に修正した．また機械学習を使ってコードクローンを検出する研究 [4] や機械翻訳のアプローチを使ってプログラムの処理内容を自然言語に変換する研究 [5] なども存在する．こういった研究が多々ある中でコーディングスタイルの検査に機械学習を応用する研究は我々が調査したところ，まだ研究されていないことがわかった．

そこで我々は機械学習のアプローチを応用して，低コストで組織内のコーディングスタイルの統一を行えるようなコーディングスタイルの自動検査手法を提案する．この手法では組織のコーディングスタイルのルールを人が明文化する必要がないため，人手がかからない上に，明文化することが難しいルールについても評価を行うことができるなどのメリットがある．本手法は Bi-Directional LSTM という自然言語の分野で用いられるニューラルネットワークのモデルを利用してソースコードからトークンの尤度を計算できるような尤度関数の導出を行うことによって実現している．ニューラルネットワークにソースコードを入力できるように学習用のソースコードの前処理を行う．そしてニューラルネットワークにソースコードの内容を学習させ，尤度関数を導出し，検査対象のソースコードの尤度を計算することでコーディングスタイルの検査を行うことが可能となる．また今回実用の面も考慮し，検査を行ったソースコードにヒートマップをオーバーレイする形でコーディングスタイル検査の結果を可視化している．

本研究ではこの手法に基づいたシステムを設計，実装した．言語は Python を用いて，ニューラルネットワークの構築には Chainer を利用した．またこのシステムの実用性を測定するためにクックパッド株式会社の協力を得て，クックパッド株式会社が社内で保有するソースコード群の一部を実際に解析する実験を行った．秘密保持の観点からソースコードをそのまま受け取ることが難しかったため，クックパッド株式会社のオフィスを訪問し，社内に用意されたサーバー上で実験を行った．実験では Rubocop との検査結果を比較し，Rubocop の検査結果を正解とみなした時にどれだけ Rubocop に近い検査精度を出せるのかを測定した．精度の指標としては Precision，Recall，F 値という指標を用いた．これらの指標は分類などのタスクを担うシステムの精度を測定するために用いられることが多く，今回の実験はあるトークンがコーディングスタイル的に正しいトークンであるか，間違ったトークンであるかの分類問題という位置づけで実験を行ったため，これらの指標を利用している．なお今回の実験で使用したソースコードは Ruby on Rails で構築されたアプリケーションの処理を担う一部分であり，アプリケーション内で保存されているユーザー情報などを含めたデータには一切関与していない．

以降， 2 章ではコーディングスタイルの組織内での統一の重要性と難しさについて述べる．またコーディングスタイルを統一するためにこれまでとられてきた手法とその問題点についても述べ，近年数々の研究がなされている機械学習の基本的な概要とソフトウェア工学に応用された例について紹介する． 3 章では我々の提案手法としてコーディングスタイルの検査に機械学習を適用する手法について述べる．前処理を行うことによってニューラルネットワークに

ソースコード群を入力する方法や得られた尤度関数からどのようにコーディングスタイルを検査するのかについても紹介する． 4 章では我々が実際に 3 章で提案した手法に基づいて実装したシステムについて触れ，本研究の実用性を示すために行った実験内容とその結果についてついても述べる． 5 章では本論文のまとめと今後の課題について述べる．

第 2 章

研究背景

近年は Ruby を代表とするようなコーディングスタイルの自由度の高い言語がウェブシステムなどで多く使用されるようになったが，このような言語のコーディングスタイルを組織内などで統一することはシステムの開発，維持コストを低減させる大事な要素である．ここでコーディングスタイルとはプログラムの記述の仕方を指す．Ruby を使用するウェブサービス提供企業内でもコーディングスタイルを統一するためのコーディングスタイルの検査などの措置が取られているが，それら既存の方法では手動での検査やルールベースでの検査となるため，検査をするためのコストが大きくなり，その結果検査自体がきちんとなされないなどの問題がある．一方で近年は機械学習をソフトウェア工学に適応し，ソフトウェア工学のルールベースとは異なる統計ベースのアプローチで既存手法の問題を解決する研究が出てきている．

本章では，コーディングスタイルの概念について詳細に説明し，その重要性と統一の困難性について紹介する．さらに，コーディングスタイルを検査する既存手法とその問題点について触れる．最後に近年増え始めているソフトウェア開発の問題を機械学習で解決しているアプローチについて機械学習の基本と実際の関連研究を紹介する．

2.1　コーディングスタイルについて

コーディングスタイルとはプログラムの表現方法を指す．コーディングスタイルはプログラムのアルゴリズムやロジックとは無関係であるため，コーディングスタイルの相違でプログラムの挙動が変わることはないが，組織内で規定されたコーディングスタイルや推奨されているコーディングスタイルでプログラムを記述することでそのプログラムの可読性を向上させることができ，そのプログラムのメンテナンスや拡張が行いやすくなるというメリットがある．例えば図 2.1 と図 2.2 を比較してみるとその可読性や実装されている機能の把握の容易さは一目瞭然である．

コーディングスタイルは有名なプログラマが策定したものや有名なプロジェクトに採択されているものなどいくつかスタイルがある．各々のプログラマはその中で自分の好みのものを選択したり，所属する組織の中で使われているものを利用したり，もしくは独自に自分のコーディングスタイルを編み出したりする．コーディングスタイルは基本的にプログラムの挙動に

```
1  a = b + c;
2  d = a + r;
3
4  if (foo->next == NULL
5      && totalcount < d
6      && needed <= MAX_ALLOT
7      && server_active(current_input))
8  {
9      true_count++;
10     printf("This␣is␣true\n");
11 } else {
12     false_count++;
13     printf("This␣is␣false\n");
14 }
```

図 2.1. 良いコーディングスタイルの例

```
1  d = (a = b + c) + r;
2
3  if (foo->next==NULL && totalcount<d && needed<=MAX_ALLOT
4    && server_active(current_input)) {
5  true_count++;printf("This␣is␣true\n");
6  } else
7  {false_count++;printf("This␣is␣false\n");}
```

図 2.2. 悪いコーディングスタイルの例

影響を与えないため，どのコーディングスタイルが良いかといった議論は常に行われ，常に正解がでることがない．

ここでは C 言語のインデントを例にいくつかのコーディングスタイルを紹介する．インデントは Python などの一部の言語を除いて，必須ではない．しかしインデントがあることによって自分自身，もしくはプログラムを見る他のプログラマがそのプログラムのループや条件分岐などの制御構造を把握しやすくなる．インデントはその大きさ一つとってもコーディングスタイルにより様々である．初期のプログラムではインデントはタブ文字が使用されていた．これはソースファイルの大きさを無駄に大きくせずに済むという理由があったためである．しかしタブ文字はそのプログラムを実行，編集する環境によって定義が異なる．UNIX 環境ではタブ文字は一般的に空白 8 文字分であるのに対して，macOS では 4 文字分である．そのため，最近の傾向としてはインデントにはタブ文字を使用することを禁止し，決められた空白文字でインデントを表現することを定めたコーディングスタイルも多く出てきている．

C 言語で最も一般に使われるコーディングスタイルは K&R スタイルである．K&R スタイルは *Brian Wilson Kernighan* と *Dennis MacAlistair Ritchie* の著書，“*The C Programming Language*” で採用されたコーディングスタイルを指す．このスタイルで書かれた C 言語プログラムを図 2.3 に示す．このスタイルはループや条件分岐のブロック開始の中括弧を制御文と

同じ行に書く．そしてブロック内の内容をインデントする．ブロックを閉じる際は改行されて宣言と同じインデントレベルになる．しかし関数は開き中括弧が宣言の次の行の先頭に記され，関数の宣言と同じインデントレベルに記述される．またインデント幅はスペース 4 つ分である．オールマンスタイルというものも存在する．このスタイルでは K&R スタイルとは対照的に制御文の後の中括弧を次の行に書き，開き括弧のインデントレベルを制御分と同じにしている．図 2.4 にこのスタイルで書いた C 言語のプログラムの例を示す．このスタイルはブロックの前後の中括弧が空白の行で囲まれるのでブロックの範囲を目視で簡単に確認できるなどのメリットがある．一方で改行が他のスタイルと比較して多くなるので，ソースコードの行数が増えるというデメリットがある．またインデント幅はスペース 4 つ分である．BSD/KNF スタイルは *Berkeley Software Distribution* 系の OS でよく使用されているコーディングスタイルである．主にカーネルコードで使われるが，ユーザープログラムでも採用されることもある．このスタイルの特徴はインデントの幅にあり，ハードタブとソフトタブの二種類が用意されている．ハードタブは 8 文字分の幅のタブで，ソフトタブは 4 文字分のタブである．ハードタブはブロックを示すために使用され，ソフトタブは一行の文字数が長くなった際に，折り返した行の先頭に使われる．例えば図 2.5 ではブロックでは 8 文字幅のハードタブを使っているが，11 行目で長くなりすぎた文を改行している箇所では 12 行目で前の行との間に 4 文字幅のソフトタブを使っている．ホワイトスミス・スタイルはあまりメジャーではない．このスタイルは制御文の後の中括弧を次の行に置き，インデントを行う．インデントレベルはブロック内と同じである．またインデント幅はスペース 4 つ分である．GNU スタイルはホワイトスミス・スタイルに似ており，制御文の後で改行し，インデントレベルを制御文とブロック内の間のレベルにする．すなわち中括弧の前に 2 文字分の空白を入れる．GNU スタイルで記述された C 言語プログラムを図 2.7 に記述する．以上のように文法に自由があまりない C 言語でも多くの慣習的なコーディングスタイルが存在している．

2.2　組織内でのコーディングスタイルの統一の重要性と難しさ

近年は Ruby を始めとしたような文法の自由度の高い言語が使われる機会が増えている．文法の自由度が高い言語ではコーディングスタイルが多様になるため，Ruby を使用してウェブサービスを提供する企業などの組織では，内部でコーディングスタイルを統一することがより重要となる．実際に多くのレシピをウェブサービスとして提供しているクックパッド株式会社や家計簿アプリを提供している株式会社マネーフォワードでは Ruby のコーディングスタイルのドキュメントを作成し，組織内で共有することでコーディングスタイルの統一を図っている．過去の研究でもデータマイニングの ‘‘`Frequent itemset mining`’’ という手法を用いて，大規模なソフトウェアコードの暗黙的なプログラミングルールを抽出できるツールの研究 [6] やオープンソースプロジェクトのソースコードとクローズドプロジェクトのソースコードのコーディングスタイルの比較を行った研究 [7] などが存在する．

しかしコーディングスタイルが多様であるが故に組織によってコーディングスタイルのルールが異なることが多々ある．例えば図 2.8 に示されるようにクックパッド株式会社では配列の

```
1  void func(argc, argv)
2      int argc;
3      char *argv[];
4  {
5      ...
6      while (i == 0) {
7          do_something();
8          i--;
9      }
10
11     if (flag)
12         do_correct();
13     else
14         do_wrong();
15     ...
16 }
```

図 2.3. K&R スタイルの C 言語プログラム

```
1  void func(argc, argv)
2      int argc;
3      char *argv[];
4  {
5      ...
6      while (i == 0)
7      {
8          do_something();
9          i--;
10     }
11
12     if (flag)
13     {
14         do_correct();
15     }
16     else
17     {
18         do_wrong();
19     }
20     ...
21 }
```

図 2.4. オールマンスタイルの C 言語プログラム

```
1  void func(argc, argv)
2          int argc;
3          char *argv[];
4  {
5          ...
6          while (i == 0) {
7                  do_something();
8                  i--;
9          }
10
11         if (judge_flag(condition1, condition2, parameter1,
               parameter2,
12             option1, option1,) {
13                 do_correct();
14         } else {
15                 do_wrong();
16         }
17         ...
18 }
```

図 2.5. BSD/KNF スタイルの C 言語プログラム

```
1  void func(argc, argv)
2      int argc;
3      char *argv[];
4  {
5      ...
6      while (i == 0)
7          {
8          do_something();
9          i--;
10         }
11
12     if (flag)
13         {
14         do_correct();
15         }
16     else
17         {
18         do_wrong();
19         }
20     ...
21 }
```

図 2.6. ホワイトスミス・スタイルの C 言語プログラム

```
1  void func(argc, argv)
2      int argc;
3      char *argv[];
4  {
5      ...
6      while (i == 0)
7        {
8          do_something();
9          i--;
10       }
11
12     if (flag)
13       {
14         do_correct();
15       }
16     else
17       {
18         do_wrong();
19       }
20     ...
21 }
```

図 2.7. GNU スタイルの C 言語プログラム

```
1  array = [
2    :foo,
3    :bar,
4    :baz,
5  ]
```

図 2.8. クックパッド株式会社の配列のコーディングスタイルの例

```
1  [
2    :foo,
3    :bar,
4    :baz
5  ]
```

図 2.9. MoneyForward の配列のコーディングスタイルの例

最後の要素の後ろにはカンマを付与することをルールとして定めているが，図 2.9 に示されるように Moneyforward ではカンマは付与しないことと定められている．また組織間のコーディングスタイルの違いだけでなく，個人的な好みや慣れで異なるコーディングスタイルを使用してしまうケースなども存在し，プログラムを記述するプログラマは都度意識しながらコーディングスタイルを選ばなくてはならないため，組織のルールにそぐわないコーディングスタイルを選択してしまうなどの人的ミスを発生させてしまう可能性がある．

2.3　コーディングスタイル統一のためのアプローチ

組織内でコーディングスタイルを統一するために考えうるアプローチとして，以下のようなアプローチが考えられる．

(a) ドキュメントの作成やメンターの教育などの人手による補助
(b) 静的解析ツールを活用したルールベースでの自動検査

(a) の手法は最もシンプルで実行しやすい手法である．ウェブ上や紙面の形で組織内で採用しているコーディングスタイルを明記し，開発メンバーに共有する．また同時に新しく加入したメンバーにはチームの中でもコーディングスタイルに熟知しているメンバーをメンターとして割り当て，一定期間コーディングスタイルを指導する．(b) の手法はルールベースを基本とするソフトウェア工学では正攻法なアプローチである．ユーザーは自分，もしくは自分が所属する組織が採用しているコーディングスタイルを基に静的解析ツールの設定ファイルにルールを記述する． 2.3.1 節， 2.3.2 節では上に列挙した手法について更に詳細に説明する．

2.3.1　人手の補助によるコーディングスタイルの統一

本節では人手の補助によるコーディングスタイルの統一のアプローチについて説明する．具体的には組織やチームが採用しているコーディングスタイルについて列挙したドキュメントを作成し，チーム内で共有する方法やメンターを割り振る方法がある．実際に Ruby をメイン言語として開発を行うクックパッド株式会社や株式会社マネーフォワードでは Ruby のコーディングスタイルのドキュメントを作成し，github 上で公開することで社内のメンバーに共有している [1][8].

クックパッド株式会社が用意するコーディングスタイルドキュメントでは Ruby の他，Objective-C，Swift，Java，CoffeeScript，Python のコーディングスタイルも用意されており，各々のルールについて “MUST”，“SHOULD” という強制度のランク分けがなされている．例えば図 2.10 は MUST レベルのコーディングスタイルであり，長いメソッドチェインの最後のメソッド呼び出しでブロックを渡す場合に，最後のメソッド呼び出しのレシーバをローカル変数として抽出し，ブロック付きメソッド呼び出しを独立した式として書くことを指定している．クックパッド株式会社のコーディングスタイルドキュメントには MUST レベルのコーディングスタイルの指定は 56 項目用意されている．一方で図 2.11 は SHOULD レベルのコーディングスタイルであり, 式の途中で改行する場合には 2 行目以降を 1 行目より 1 段深くインデントすることを指定している．クックパッド株式会社のコーディングスタイルドキュメントには SHOULD レベルのコーディングスタイルの指定は 40 項目用意されている．

また新しく開発に参加するメンバーにはメンターをつけて，コーディングスタイルの指導やチェックを行うなどしてコーディングスタイルの保守を管理している．メンターとしてつけられるのはチームの中でもある程度所属していた期間が長く，チームが採用しているコーディン

```
1  # good
2  posts = Post.joins(:user).
3    merge(User.paid).
4    where(created_at: target_date)
5  posts.each do |post|
6    next if stuff_ids.include?(post.user_id)
7    comment_count += post.comments.size
8  end
9
10 # bad
11 posts = Post.joins(:user).
12   merge(User.paid).
13   where(created_at: target_date).each do |post|
14     next if stuff_ids.include?(post.user_id)
15     comment_count += post.comments.size
16   end
```

図 2.10. MUST レベルのコーディングスタイルの例

```
1  # good
2  User.active.
3    some_scope(foo).
4    other_scope(bar)
5
6  # bad
7  User.active.
8  some_scope(foo).
9  other_scope(bar)
```

図 2.11. SHOULD レベルのコーディングスタイルの例

グスタイルについて熟知しているメンバーがアサインされる．

2.3.2　静的解析ツールを活用したルールベースでの自動検査

本節では静的解析ツールを活用して，ルールベースでコーディングスタイルを自動検査するアプローチについて説明する．コーディングスタイルを静的に解析するツールはユーザーが事前に設定ファイルに自分，もしくは自分が所属する組織が採用しているコーディングスタイルをルールとして記述する．ツールはその設定ファイルをもとに対象のソースコードについてコーディングスタイルの検査，また可能な場合には自動修正を行う．

実際に Ruby では Rubocop，またその他の言語でも IDE 内に Auto formatter というツールが存在する．これらのツールはルールベースで自動的にコーディングスタイルの検査が行えるため，正確にコーディングスタイルの誤りを発見することが可能となる．また Ruby の静的解析が可能な Rubocop は Ruby の自由度の高い文法に対応するため YAML 形式の設定ファイルを記述することでユーザーがルールをカスタマイズすることが可能となる．Rubocop に

```
1 LineLength:
2   Max: 128
3
4 Style/NumericLiterals:
5   Enabled: true
6
7 Layout/IndentArray:
8   Enabled: true
9   EnforcedStyle: consistent
```

図 2.12. Rubocop の設定ファイルの記述例

は Cops という Ruby の記述ルールが予め定義されていてユーザーはこれらの Cops のオンオフの切り替えやパラメーターを変更することによって設定ルールをカスタマイズすることが可能となる．

図 2.12 に Rubocop の設定ファイルの記述例を示す．図 2.12 に記されるようにユーザーは自分がチェックしたいコーディングスタイルに適用されるルールを順番に記述する．この例では 3 つのルールがカスタマイズルールとして設定されている．1, 2 行目では `Linelength` という Cops で一行に許容する文字数を 128 文字に設定している．4, 5 行目では `Style` というグループの `NumericLiterals` という Cops をオンにすることで 5 桁を超える数値を表記する際には `1_000_000` というような 3 桁毎にアンダースコアを挿入するような記述がなされているかをチェックするルールを Rubocop にカスタマイズすることが可能となる．7 から 9 行目では `Layout` というグループの `IndentArray` という Cops をオンにして，`EnforcedStyle` オプションを `consistent` に設定している．この Cops は Ruby の配列内のインデントのコーディングスタイルをチェックするルールであるが，`EnforcedStyle` オプションに適用したいインデントルールを適用することができる．`IndentArray` の `EnforcedStyle` で適用できるオプションルールは以下の 3 つが用意されている．

(a) special_inside_parentheses
(b) consistent
(c) align_brackets

(a) の special_inside_parentheses を設定すると図 2.13 に記されているように純粋に配列を定義する際にはインデントは配列の定義行の行頭を基準としてインデントを行う．しかしメソッド呼び出しのブラケット中で配列を定義する場合には開きブラケットの位置を基準にインデントを行う．(b) の consistent を設定すると図 2.14 に記されているように配列定義の際もメソッド呼び出し内での配列定義でも行頭を基準にしてインデントを行う．(c) の align_brackets を設定すると図 2.15 に記されているように配列定義の際もメソッド呼び出し内での配列定義でも開きブラケットを基準にしてインデントを行う．

Rubocop では上記の例のように予めよく利用されるルールを Cops として定義してあり，ユーザーはこれらの Cops を YAML ファイル内で呼び出すことでルールをカスタマイズする

```
1  # special_inside_parentheses
2  array = [
3    :value
4  ]
5
6  array_in_a_method_call([
7                            :value
8                          ])
```

図 2.13. Indent Array の special_inside_parentheses オプション

```
1  # consistent
2  array = [
3    :value
4  ]
5
6  array_in_a_method_call([
7    :value
8  ])
```

図 2.14. Indent Array の consistent オプション

```
1  # align_brackets
2  array = [
3            :value
4          ]
5
6  array_in_a_method_call([
7                            :value
8                          ])
```

図 2.15. Indent Array の align_brackets オプション

ことができる．

2.4　機械学習を応用したソフトウェア工学の研究

近年は機械学習の研究が盛んに行われ，その研究結果も著しく向上している．自然言語の研究もそんな近年機械学習分野で研究が盛んな分野の 1 つであるが，ソフトウェア工学の分野でそういった自然言語で適用されているアプローチをソフトウェアのタスクに応用する研究も増え始めている．ソフトウェア工学は従来までルールベースのアプローチが主流であったが，機械学習の発達でソフトウェア工学の研究の新しいアプローチとして着目されている．本節ではまず機械学習の基本的な仕組みについて触れる．そして機械学習機械学習をソフトウェア工学に応用した関連研究を紹介する．

2.4.1　機械学習の基本知識

機械学習は大量のデータからそのデータ内に潜むパターンを自動的に抽出することを指す．抽出したパターンを新たなデータに当てはめることで予測や自動判別などのタスクを可能とする．機械学習はこれまで人手で考えられてきたアルゴリズムを機械が自動的に構築するため，多くの分野で応用が試みられている．以下では *Understanding LSTM Networks*[9] の構成を参考に LSTM の仕組みについて説明している．

機械学習の手法には達成したい目標やタスクによって様々な種類があり，大きくは「教師あり学習」と「教師なし学習」に分類できる．教師あり学習は正解となるデータを用意する必要がある．ここで正解となるデータとはある入力に対して，どのような出力が期待されているかを示すデータを指す．対して教師なし学習とは正解となるデータを必要としない学習を指し，データの中から本質的な構造や分類ルールを抽出する際に用いる．更に教師あり学習，教師なし学習は各々以下のような手法が存在する．

(a) 教師あり学習
  (a1) 回帰
  (a2) 木
  (a3) ニューラルネットワーク
  (a4) ベイズ
  (a5) 時系列
  (a6) クラスタリング
  (a7) アンサンブル学習
(b) 教師なし学習
  (b1) クラスタリング
  (b2) 自己組織化マップ

この中でも近年は教師あり学習のニューラルネットワークを利用する手法が最も盛んに研究され，様々な分野に応用されている．ニューラルネットワークは脳の機能を計算機上で表現した数学モデルを指す．脳の中には多数のニューロンが存在しており，各ニューロンは多数の他のニューロンから信号を受け取り，その信号をまた次のニューロンへと渡している．脳はこの信号の流れで様々な情報処理を行っている．ニューラルネットワークはこの仕組を計算機上で表現しており，脳がニューロンに情報を入力するように計算機がシュミレートしたニューロンに情報を入力して学習する．単純なニューラルネットワークの構造を図 2.16 に示す．図中の丸，四角，ひし形は脳のニューロンに対応し，ユニットと称される．また各ユニット同士を結ぶ矢印はユニットからユニットへの情報の流れを示す．各ニューロンは層を形成しており，丸のユニットが形成する層を入力層と呼び，外部からの情報を受け取る．ひし形のユニットが形成する層を出力層と呼び，入力情報を処理した結果を外部に出力する．四角のユニットが形成する層を隠れ層と呼び，入力層から受け取った情報を出力層へ受け渡す．

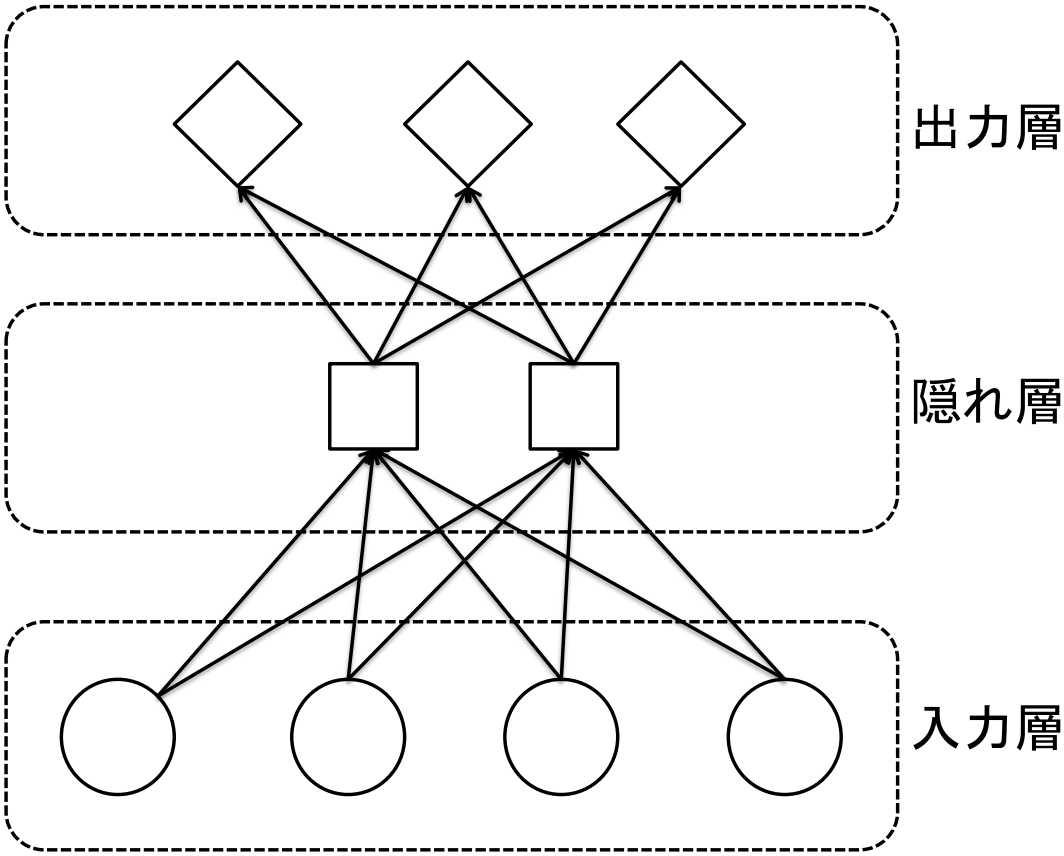

図 2.16. ニューラルネットワークの構造図

図 2.16 は最もシンプルなニューラルネットワークを示したが，ネットワークの隠れ層の階層を深めたアルゴリズムとして深層学習というアルゴリズムが存在し，ニューラルネットワークの研究の中でも特に研究が盛んに行われている．ディープラーニングの中でも特に利用される頻度の高いアルゴリズムとしては Convolutional Neural Network (CNN) と Recurrent Neural Network (RNN) が挙げられる．CNN は画像認識に多用される一方で，RNN は自然言語処理でよく利用される．プログラムのソースコード解析などソフトウェア工学の研究課題は自然言語処理の課題と共通部分が多いため，ソフトウェア工学に応用されるニューラルネットワークも RNN が多い．

従来のニューラルネットワークは入力情報，出力情報が独立のデータを扱っている．それに対して RNN は過去のニューラルネットワークの状態を保持することで，連続的な入力情報を扱うことが可能となる．連続的な入力情報の例としては自然言語で構成された文章，人が発する音声などが挙げられる．RNN の簡単な構造図を図 2.17 に示す．また RNN の展開図を図 2.18 に示す．図中の x_t，h_t，y_t はそれぞれ連続的な情報が入力された時の t ステップ目の入力層，隠れ層，出力層を示す．1 ステップ毎に単語や文字など連続的な情報を作る単体の情報を処理する．図 2.17 が示すように，RNN は内部にループ構造をもつことで t ステップ目の処理の際，隠れ層には x_t と一緒に $t-1$ ステップ目の隠れ層である h_{t-1} が入力される．このような仕組みにより各ステップの処理に過去の処理結果を反映することが可能となる．

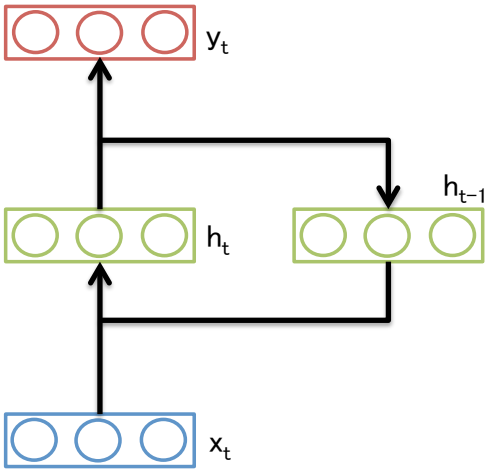

図 2.17. RNN の構造図

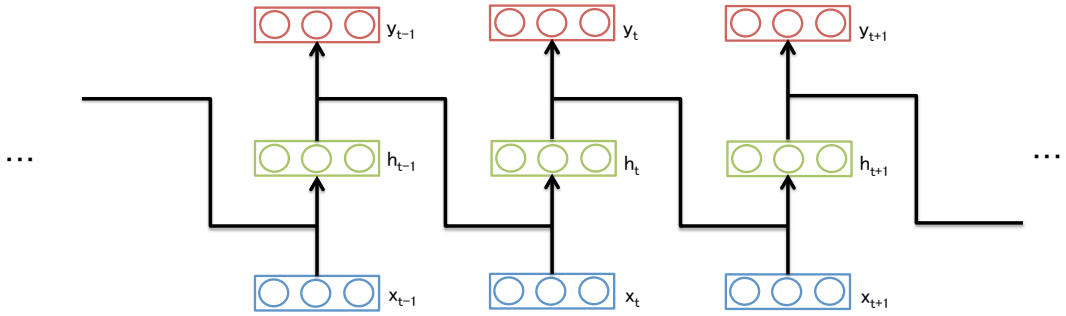

図 2.18. 展開された RNN の構造図

しかし単純な RNN では隠れ層に多くの情報を保持できないため，長い連続情報は扱うことができない [10][11]．そのためより長い連続情報を扱える RNN として Long Short Term Memory (LSTM) が実際の研究などでは利用させることが多い．LSTM の基本的な構造は先述した RNN と同じである．しかし LSTM は隠れ層を LSTM Block と呼ばれるメモリと 3 つのゲートを持つブロックに置き換えている．この LSTM Block が長いステップ間の情報を保持できるようにしている [12]．

以下では RNN と LSTM の差異が明確になるように各々の隠れ層の中身をクローズアップした図を図 2.19，図 2.20 に示しながら LSTM の構造を説明する．なお図中では緑の長方形は学習されるニューラルネットワークの一層を表す．オレンジの丸はベクトルの加算や乗算などの演算を表す．矢印はベクトルの移動の流れを表しており，矢印の合流は移動しているベクトルの合流，分岐はコピーを表している．

図 2.19 に示すように RNN の隠れ層の内部には単一の tanh 層という非常に単純な構造を持っている．一方で図 2.20 に示すように LSTM では内部には複数のニューラルネットワーク層とベクトル演算を保持していおり，複雑な仕組みで成り立っている．以下ではこの内部の仕組みを順番に説明する．

LSTM の中心的な特徴は図 2.21 ハイライトされている隠れ層の上部を通る C_t の情報の変

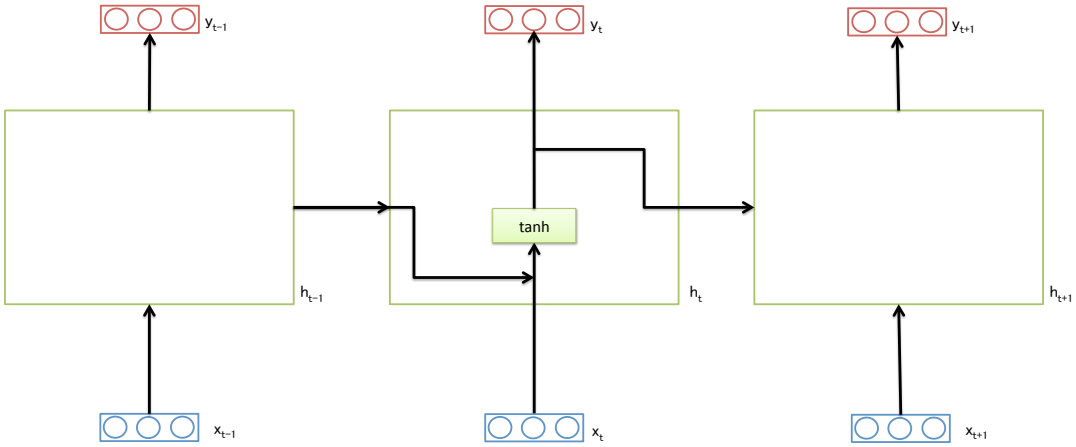

図 2.19. RNN の隠れ層をクローズアップした構造図

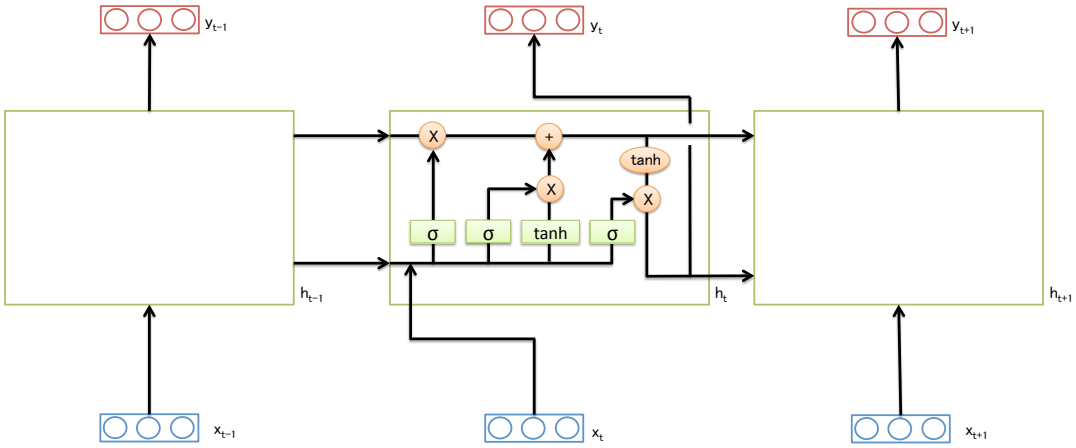

図 2.20. LSTM の隠れ層をクローズアップした構造図

化である．この C_t をセル状態と呼び，LSTM ではこのセル状態に対して情報を付与したり逆に削除したりすることによって長期的な記憶を保持できるようになっている．この操作をゲートと呼ばれる構造で制御している．ゲートは情報を選択的に通す方法でシグモイド・ニューラルネット層と一点の乗算により構成される．シグモイド・ニューラルネット層は 0 から 1 までの数値を出力する層のことを指す．

図 2.22 は LSTM の最初のステップを示す．最初のステップは忘却ゲートと呼ばれており，セル状態から捨てる情報を選択する．忘却ゲートで行われる処理をまずは式 2.1 に示す．h_{t-1} と x_t の値から σ が 0 から 1 の間の数値 f_t を出力する．1 は C_{t-1} の情報を完全に維持することを示し，0 は完全に取り除くことを示す．

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.1}$$

次のステップでは図 2.23 に示すようにセル状態 C_{t-1} に新たに保存する情報を判定する．このステップは 2 つの部分に分けられる．図 2.23 の左側が入力ゲートと呼ばれるシグモイド層は更新する値を判定する．図 2.23 の右側は tanh 層で構成されており，C_{t-1} に新たに追加

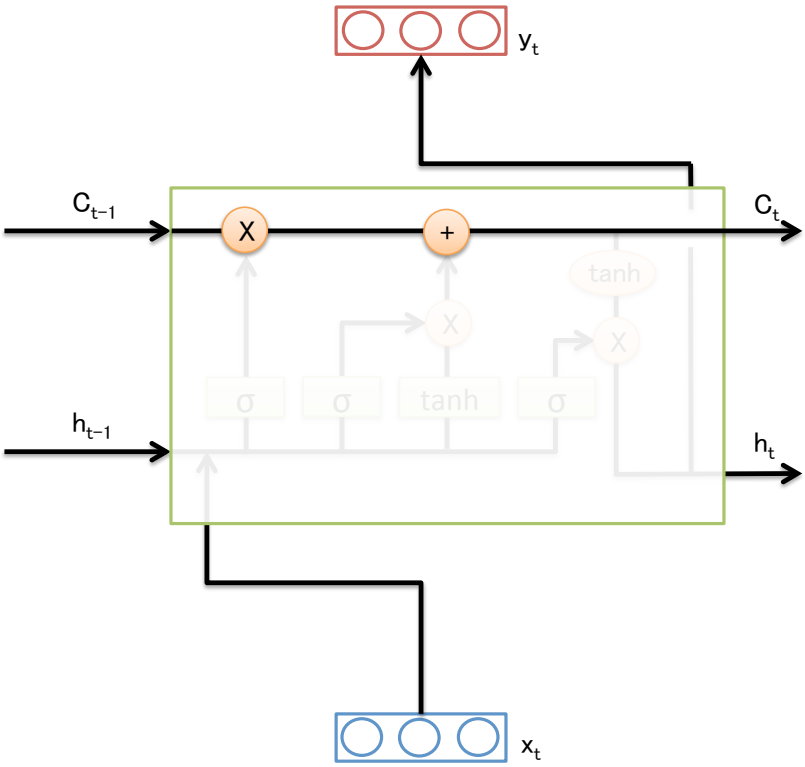

図 2.21. LSTM の情報の流れ 1

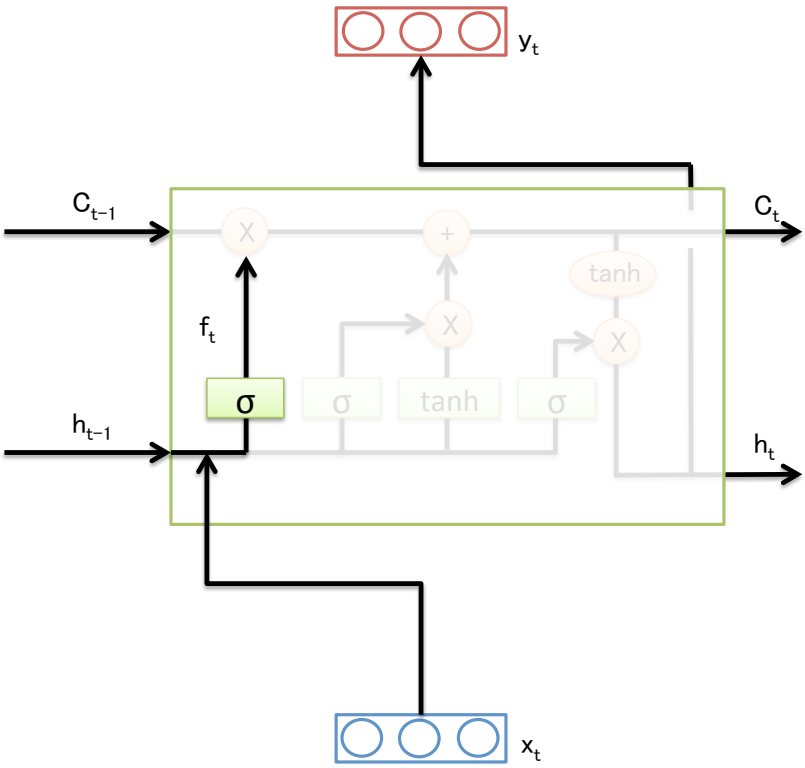

図 2.22. LSTM の情報の流れ 2

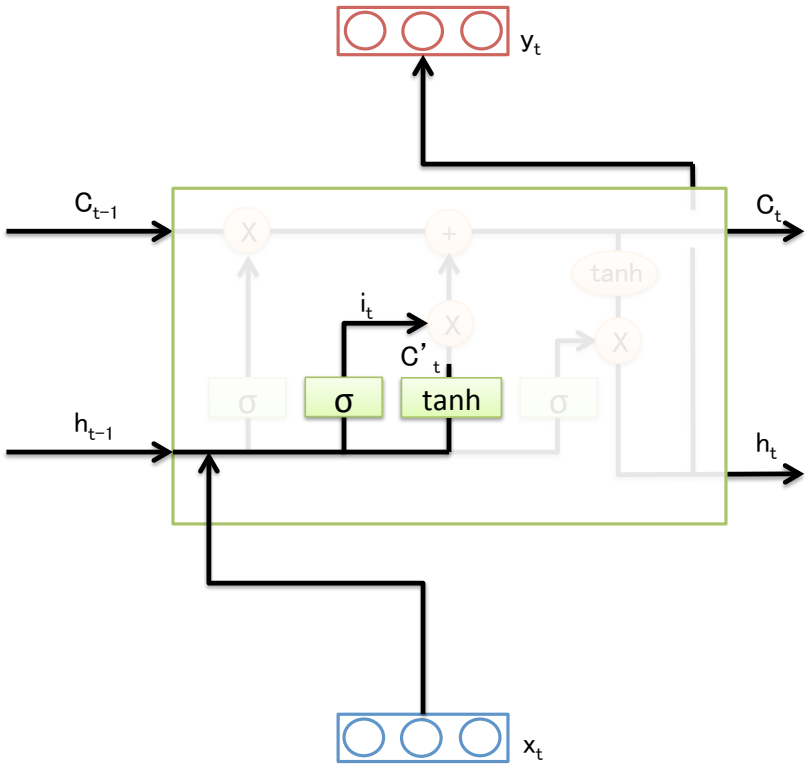

図 2.23. LSTM の情報の流れ 3

される新たな候補値ベクトル C'_t を生成する．そしてそれら 2 つの値を情報更新のために組み合わせる．これらの処理内容を式で表したものが式 2.2 である．

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.2}$$
$$C'_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.3}$$

そして前のステップで計算した値を使って，古いセル状態 C_{t-1} を新しいセル状態 C_t に更新する．最初のステップで計算した f_t を C_{t-1} にかけ合わせ，忘れるべきと判断されたものを忘れる処理を行う．そして 2 つ目のステップで計算した $i_t * C'_t$ を加える．これによりセル状態に新しい情報を付加させることができる．これらの処理をハイライトした図が図 2.24 であり，処理内容を計算式で表したのが式 2.4 である．

$$C_t = f_t * C_{t-1} + i * C'_t \tag{2.4}$$

最後のステップでは出力するものを判定する処理を行う．図 2.25 にその処理をハイライトしたものを示す．まず $t-1$ の状態での隠れ層の出力 h_{t-1} にシグモイド層を実行する．この層がセル状態のどの部分を出力するのかを判定した o_t を計算する．そしてこの o_t に tanh 層を適用したセル状態 C_t と掛け合わせる．これらの処理内容を計算式で表したのが式 2.5 である．

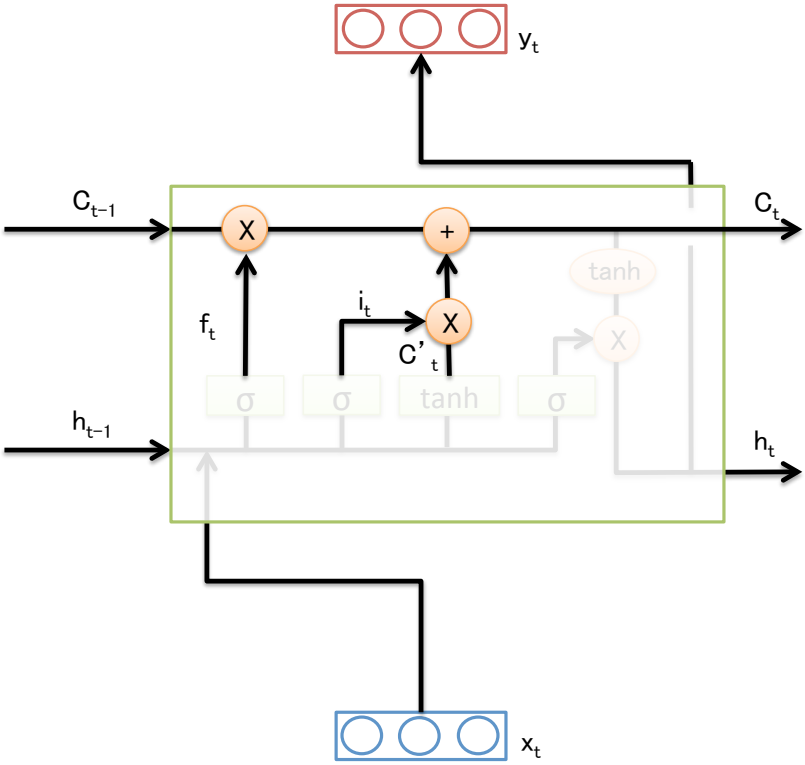

図 2.24. LSTM の情報の流れ 4

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.5}$$

$$h_t = o_t * \tanh(C_t) \tag{2.6}$$

上記が LSTM の簡単な仕組みの説明である．本節で説明した LSTM の他に拡張された LSTM なども存在する [13][14]．LSTM はこのような仕組みを有することで長期の記憶を保持することが可能となり，例えば自然言語のタスクであれば長い自然言語の文章の尤度を計算するなどのタスクを精度よく計算することが可能となった．すなわち文末の尤度を計算する際に文頭で得られた情報を保持した状態で尤度計算が可能となっているのである．

2.4.2　機械学習を応用したソフトウェア工学の研究

近年，GPU を安価に活用できるようになったことや，ニューラルネットワークを利用することでより精度良く学習が可能になったため，機械学習の研究が盛んに行われている．そんな中で従来まではルールベースで行われてきたソフトウェア工学の研究に機械学習を応用して統計ベースでソフトウェア開発の改善や開発者の支援を行う研究がいくつか発表されている．

DeepFix[3] は機械学習とコンパイラを組み合わせた C プログラムの自動修正手法である．プログラムの自動修正をテーマにした研究はソフトウェア工学の分野では昔から盛んに行われていた．しかし既存のルールベースのアプローチではプログラム中の 1 つのエラーを修正する

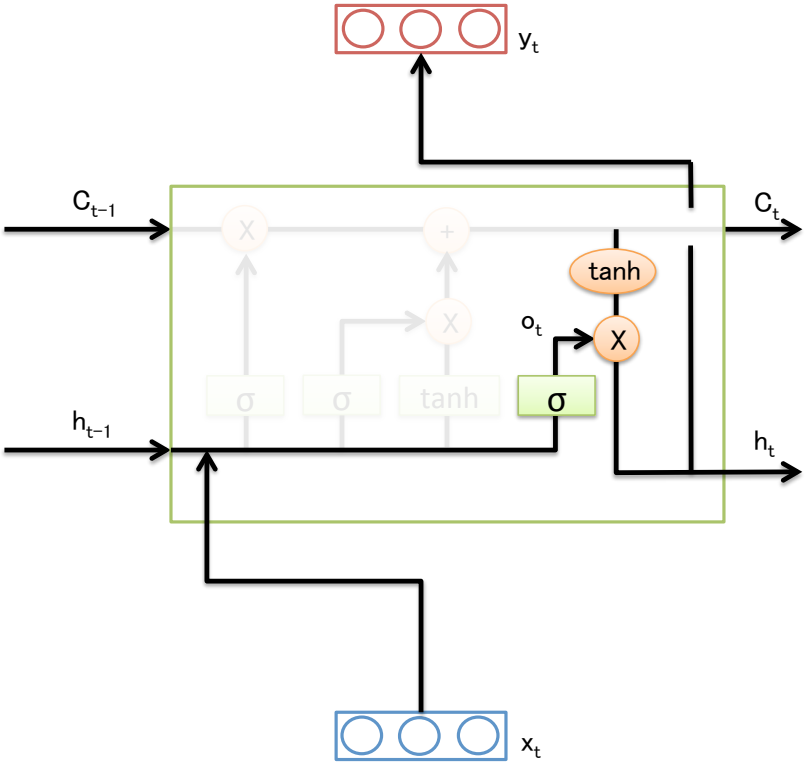

図 2.25. LSTM の情報の流れ 5

にしても，プログラム全体を解析する必要があり，プログラムに複数のエラーが含まれていた場合にはプログラム全体の解析が正しく行えないため，修正がより難しくなってしまう．そこで機械学習分野で盛んに研究されている自然言語の文法エラー修正の手法を応用し，プログラムの自動修正を機械学習で実現したのが DeepFix である．DeepFix は C コンパイラ以外の外部ツールを使わずに C プログラムの自動修正を可能とする．DeepFix はニューラルネットワークを構成する RNN としては LSTM に更新ゲートというゲートを付加した *Gated Recurrent Unit (GRU)* を採用しており，この GRU を 2 つ組み合わせた Sequence-to-Sequence モデルをニューラルネットワークに採用している．Sequence-to-Sequence モデルは機械翻訳などのタスクで採用されるモデルで，あるシーケンス列を入力するとそれに対応するシーケンス列を出力する．DeepFix では誤りを含むプログラムソースコードを入力シーケンスとして入力し，修正されたソースコードを出力シーケンスとして出力する．さらに DeepFix は修正したソースコードを C コンパイラでコンパイルし，発生するエラーメッセージが修正前と比較して減少した際にその修正を採用して再度修正を試みるというイテレーティブな手法を採用している．実際に学生が書いた誤りを含む C プログラム 6971 ファイルで修正を試みたところ 1881 ファイル，全体の 27% を完全に修正することに成功し，また 1338 ファイル，全体の 19% を部分的に修正することができた．

DeepFix で利用されている Sequence-to-Sequence モデルは 2 言語間の機械翻訳のタスクで採用され，近年の機械学習分野での主要な研究テーマの一つになっているが，機械翻訳の分野

```
1 def fizzbuzz(n):
2     if not isinstance(n, int):
3         raise TypeError("n␣is␣not␣an␣integer")
4     if n % 3 == 0:
5         return "fizzbuzz" if n % 5 == 0 else "fizz"
6     elif n % 5 == 0:
7         return "buzz"
8     else:
9         return str(n)
```

図 2.26. 擬似コードを生成するためのソースコード

```
1 # define the function fizzbuzz with an argument n.
2 #   if n is not an integer value,
3 #       throw a TypeError exception with a message...
4 #   if n is divisible by 3,
5 #       return "fizzbuzz" if n is divisible by 5, or "fizz
      " if not.
6 #   if not, and n is divisible by 5,
7 #       return the string "buzz".
8 #   otherwise,
9 #   return the string representation of n.
```

図 2.27. 図 2.26 を基に生成された擬似コード

では統計的機械翻訳という手法を用いて機械翻訳を実現する手法も多く研究されている．そんな統計的機械翻訳をソフトウェア工学の分野に応用し，プログラムの内容からプログラムの機能を自然言語で説明するドキュメントを擬似コードとして自動生成する研究 [5] も存在する．例えば図 2.26 のような Python のソースコードを入力すると図 2.27 のように図 2.26 の処理内容を説明する擬似コードが自動で生成される．この翻訳器が実際に翻訳できるようになるためにはソースコードと擬似コードのペアのコーパスが必要になる．実験では実際のプログラムに自然言語で説明を付与するプログラマを雇い，ソースコードと擬似コードのペアを学習用のペアコーパスとして準備した．評価としては機械翻訳の分野でも利用される BLEU 値を使った定量的な評価と被験者に対して擬似コードの精度をインタビューする定性的な評価が行われた．擬似コードの言語としては英語と日本語が使用され，定量評価は両言語で，定性評価は日本語のみで行われた．結果として最も改良された擬似コード生成器を利用することで，Python-to-English は BLEU が 54.08%，Python-to-Japanese は BLEU が 62.88% という結果を出すことができた．またインタビューの結果，0 から 5 のスコア中，4.155 というスコアを出すことができた．

機械学習の手法を使ってコードクローンを検出する研究 [4] も存在する．コードクローンを捜索する研究はソフトウェア工学の分野で盛んに行われていた．この研究では Recurrent Neural Network と Recursive Neural Network を組み合わせてソースコードをベクトル表現にして，コードクローンを捜索している．Recurrent Neural Network がシーケンスの順番

にネットワークに入力するのに対して，Recursive Neural Network は木構造に沿ってネットワークに入力を行いベクトルを生成する．

上に挙げた関連研究のように機械学習で利用されるアプローチ，特に自然言語分野で採用されているアプローチを学習するニューラルネットワークの部分を改良して，プログラムを解析できるような形に工夫することで従来までルールベースでのみのアプローチだったソフトウェア工学の研究テーマに新しいアプローチとして統計ベースとなる機械学習を用いたアプローチが有用である例が出始めている．

2.5　各手法の比較と問題点

2.3.1 節， 2.3.2 節で触れたようにコーディングスタイルを統一するための手法はいくつか考えられる．また実際に 2.3.1 節で触れたドキュメントの作成やメンターのアサインはエンジニアを多く抱える企業や組織では実施されている．また 2.3.2 節でも触れたような Rubocop などのツールを使ってプロジェクトを管理する企業や組織も存在する．

しかし 2.3.1 節や 2.3.2 節で紹介した手法は人手を要するため，運用のためのコストが高い．マニュアルの作成やメンターのアサインは開発のメンバーの中から担当者が選ばれるため，開発の人手の一部がドキュメント作成やメンターに割かれてしまう．Rubocop などのルールベースのツールを利用する場合，コーディングスタイルのルールは手動で設定しなくてはならず，ルールが変化したり，プログラミング言語のバージョンの移行で言語仕様そのものが変わった際にも手動で設定を変更しなくてはならないという手間がかかる．またコーディングスタイルをルールとして明文化する必要があるため，表現することが難しいコーディングスタイルのルールを設定することができない場合や Rubocop のソースコードに直接変更を加えなければ実現できないものなどが存在してしまうという問題点がある．

一方で 2.4 節で紹介したように機械学習のアプローチ，特に自然言語で行われたアプローチはソフトウェア工学の分野への応用例が多く，実際にソフトウェア工学のこれまでのテーマに新しい切り口になりうる．コーディングスタイルも実際にはソースコードの特性と考えれば機械学習のアプローチを使ってその特性を統計的に解析することも可能であると考えられる．2.3.2 節で紹介したルールベースでのアプローチが常にルールに沿った正しい判定を下すのに比べると 2.4 節で取り上げられた関連研究のアプローチは確率ベースになるため多少の誤判定が生じてしまうが， 2.4 節でとりあげた関連研究のように学習するニューラルネットワークを工夫し，学習の精度を向上させることでルールベースに近い精度の判定や分類が可能になる考えるとその点についても誤判定を出来る限り小さくできると考えられる．本研究を実施するにあたり過去の関連研究を調査した結果，機械学習を用いてコーディングスタイルの自動検査を行う研究はなされていないことがわかった．

第 3 章

提案手法

本論文では，機械学習を応用することでより低コストに組織内でのコーディングスタイルの統一を行えるようなコーディングスタイルを自動で検査する手法を提案する．この手法では組織のコーディングスタイルのルールを人が明文化する必要がないため，人手がかからない上に，明文化することが難しいルールについても評価を行うことができるなどのメリットがある．本章では本手法の概要について述べる．さらに本手法が具体的にどのようにコーディングスタイルの尤度関数を導出し，尤度からコーディングスタイルを評価するのかを紹介する．

3.1 概要

2 章で触れたようにコーディングスタイルをルールとして明文化する手法では手動で作成するためコストがかかり，また明文化が難しいルールなどが存在するなどの問題がある．提案手法ではコーディングスタイルをルールではなく，トークン列の尤度で表現し，コーディングスタイルの適合評価を自動で行う．

提案手法では機械学習の自然言語処理のアプローチを応用することでコーディングスタイルの尤度関数を導出することを可能にしている．過去に組織内で作成された「コーディングスタイルが正しいソースコード群」をニューラルネットワークに入力できる形に変形させる前処理を行い，トークン列に変換する．前処理で得られたトークン列を学習データとしてニューラルネットワークに学習させ，学習済みのニューラルネットワークからコーディングスタイルの尤度関数を導出する．導出した尤度関数から計算される尤度を基に検査対象のソースコードのコーディングスタイルを検査することが可能となる．ニューラルネットワークには RNN の一つである LSTM を組み合わせた Bi-Directional LSTM を利用する．Bi-Directional LSTM を利用することで LSTM よりも高い精度で尤度を計算する事が可能となる．より具体的な処理内容についての説明は 3.2 節， 3.3 節， 3.4 節で触れる．

提案手法を用いることでベースとなるツールを各組織のコーディングスタイルに合わせてカスタマイズする作業がほぼ自動化されるため，ルール作成や拡張のコストがかからずに様々なコーディングスタイルを自動で評価できる．これにより規則化の作業コストが不要になるため、コーディングスタイルの誤り検出ツールの開発コストが下がると考えられる．

3.2　前処理

本提案手法ではプログラムのソースコードからコーディングスタイルを学習し，尤度関数を導出するために，尤度関数導出前にソースコードに前処理を行う．これによりコーディングスタイルを評価するために必要な情報をソースコードから抽出し，ニューラルネットワークに学習させることが可能になる．本節ではその具体的な処理方法を説明する．

3.2.1　ソースコードのトークン列化

本手法ではニューラルネットワークにソースコードの内容を入力するために Ruby の字句解析器を利用して，ソースコードをトークン列に変換する．本手法では通常の字句解析器を改良したものを利用する．改良された字句解析器は通常の字句解析器では取得できなかった改行情報や空白情報を取得する．コーディングスタイルを評価する上で，改行や空白，インデントの位置は重要な情報になる．

さらにインデント情報は前行との相対的なインデント情報として取得される．図 3.1 に示されるソースコードを字句解析器でトークン化する場合を考える．1 行目の行頭にはインデントが存在しないため，必ずインデントレベルは 0 であり，字句解析器はそのインデント情報を記憶しておく．1 行目が改行され，2 行目の解析を始める時，行頭でインデントを検知する．この時，字句解析器は 2 行目のインデントレベルが 1 つあがったことと 1 行目のインデントレベルが 0 であったことから相対的なインデントは `+1` であることを計算し，2 行目のトークン列の始めに`<INDENT1>`を記録する．そして字句解析器は 2 行目のインデントレベルが 1 であったことを記憶し，3 行目の解析を行う際にはその記憶と 3 行目の行頭に存在するインデントの数から相対的なインデントレベルを計算する．逆に 4 行目では前行よりもインデントレベルが 1 つさがっているため，インデントの相対的なレベルは`-1` となり，`<INDENT-1>`が記録される．このような手法を取り入れるのはコーディングスタイルを評価する観点ではインデントの絶対的なレベルよりも前後との相対的なレベルの方が重要であると考えるからである．例えば図 3.2 で 2 行目に出現する `if` 制御文と 7 行目に出現する `if` 制御文は同じ制御文であるため，どちらも次行のコードは `if` よりもインデントレベルが 1 つあがる．しかし 2 行目の `if` は `for` 制御構造の中に記述されているため元々インデントレベルが 1 つ高い．そのため絶対的なインデントレベルで見ると 2 行目の `if` の次行は`<INDENT2>`となり，7 行目の `if` の次行は`<INDENT1>`となるため，学習するネットワークからすると `if` トークンの次行に全く違うインデント構造を持つと学習されてしまう．そこで相対的なインデント情報をトークンの中に付加してやれば図 3.2 中の `if` の次行の行頭はどちらもインデントが`<INDENT1>`となるためどちらも学習時に同じ構造として学習できる．

本手法では字句解析後，結果として得られたトークンは抽象化して利用する．例えば変数名が `hoge` の変数を得られた場合，それは`<ID>`に変換される．これにより変数名などのプログラム特有の情報が学習に影響されることを防いでいる．Ruby のデータとトークンの変換の対応

```
1 class Barbeque::AppsController < Barbeque::
    ApplicationController
2   def index
3     @apps = Barbeque::App.all
4   end
5   ...
```

図 3.1. 相対的なインデント情報の取得例

```
1 for num in 1..3 do
2   if num % 2 == 0 then
3     print("num␣=␣", num, "\n")
4   end
5 end
6
7 if flag == true then
8   print("Condition␣is␣OK")
9 end
```

図 3.2. インデント情報の例

表 3.1. 抽象化のための各トークンの表記とデータ型の対応表

データの型	トークン表記
空白	<SPACE>
改行	<NEWLINE>
インデント	<INDENTX>
変数	<ID>
クラス	<CLASS>
インスタンス変数	<INSTANCE_VAL>
関数	<FUNCTION>
文字列	<STRING>
数値	<NUMBER>
リテラル	<LITERAL>
オペレーター	<OPERATOR>
クラス変数	<CLASS_VAL>
定数	<CONSTANT_ID>
名前空間	<NAMESPACE>
グローバル変数	<GLOBAL_VAL>
コメント	<COMMENT>
解析に失敗したトークン	<ERROR>

付けは 3.1 に示す．このような抽象化を行うことでコーディングスタイルに着目した尤度計算をより正確に行うことを可能にしている．また Ruby にはいくつか予約語が存在するが，それらは特別なトークンとして抽象化せずにそのままトークン化している．予約語として扱っているものを以下に列挙する．

- BEGIN
- class
- ensure
- nil
- self
- when
- END
- def
- false
- not
- super
- while
- alias
- defined?
- for
- or
- then
- yield
- and
- do
- if
- redo
- true
- begin
- else
- in
- rescue
- undef
- break
- elsif
- module
- retry
- unless
- case
- end
- next
- return
- until
- except
- let
- raise
- __LINE__
- __FILE__
- __ENCODING__

3.2.2　ソースコードの分割

ソースコードをトークン列に変換したものをそのままニューラルネットワークに入力してしまうとトークン列長が長くなりすぎてしまい，コーディングスタイルのルールをうまく学習できない．一方でソースコードの各行毎に学習を行ってしまうと行を跨いで表現されるコーディングスタイルを学習することができなくなる．そこで本提案では Windows Slide Algorithm を使用し，予め設定された行数を Windows size として，重複を許しながらソースコードから該当するトークン列を切り出す．切り出した一つのトークン列を 1 つの学習データとして学習させる．こうすることで学習データとして適切な長さに変換できるだけでなく，一つのソースコードからより多くの学習データを生成できるというメリットがある．

トークン化と分割を行うことでニューラルネットワークに学習させるためのデータを用意できる．例えばトークン化前のソースコードが図 3.3 であったするとトークン化後のトークン列は図 3.4 のようになる．図 3.4 からわかるように 1 行が 1window になっており，学習時には 1 つのシーケンスとして扱われる．トークンの区切りは空白で示されている．図 3.4 では 5 行を 1window としているため，各 window で前後の window と 4 行分のトークンが重複していることがわかる．

```
1  class Barbeque::AppsController < Barbeque::
       ApplicationController
2    def update
3      @app = Barbeque::App.find(params[:id])
4      # Name can't be changed after it's created.
5      if @app.update(params.require(:app).permit(:
           docker_image, :description))
6        redirect_to @app, notice: 'App␣was␣successfully␣
             updated.'
7      else
8        render :edit
9      end
10   end
11 end
```

図 3.3. トークン化する前のソースコード例

3.3 コーディングスタイルの尤度関数導出

本節では 3.2 節で述べた方法で得られたトークン列の学習データをニューラルネットワークに入力し，コーディングスタイルの尤度関数を導出する方法を紹介する．本手法ではニューラルネットワークには RNN の一つである LSTM を組み合わせた Bi-Directional LSTM を利用する．尤度関数を導出する仕組みとしては Bi-Directional LSTM の他に単純な LSTM やニューラルネットワーク以外の手法として N-gram という手法が存在する．N-gram はあるシーケンス情報の中の隣り合う N 個の単語群，または文字群を指す．例えば，`‘‘I am Yuki Kobayashi’’` という英文があった時，この文の N を 2 とした単語 N-gram は {I-am, am-Yuki, Yuki-Kobayashi} となる．このように隣り合う単語や文字情報で分割されたシーケンス群からその単語や文字が隣り合う確率を計算することでシーケンス内の単語や文字の尤度が計算できる．N-gram はニューラルネットワークを構築する手法に比べれば手軽だが，パラメータ N が大きすぎると単語や文字の組み合わせが大きくなりすぎて，正しい精度を求めることができない．一方で N の値が小さいためにある単語の尤度を計算する場合にはその周辺の単語しか予測の要素としては取り入れられない．また LSTM と Bi-Directional LSTM を比較した場合に，Bi-Directional LSTM は後ろ向きのシーケンス情報も入力できるため単純に入力できる情報が 2 倍となり，LSTM よりもより多くの情報を考慮して尤度を計算することが可能となる．コーディングスタイルの検査には前後のより広い範囲のトークンの情報が非常に重要になると考えられる．以上のような理由から今回 Bi-Directional LSTM を尤度計算のためのニューラルネットワークとして採用した．

Bi-Directional LSTM のネットワーク構成図を図 3.5 に示す．Bi-Directional LSTM では 2.4 節で紹介した LSTM を 2 つ組み合わせる．自然言語の文章やプログラムのトークン列などのシーケンス情報の並びの尤度を計算するタスクなどに利用される．実際に *context2vec*[15]

```
1 class <SPACE> <CLASS> :: <CONSTANT_ID> <SPACE> < <SPACE> <
    CONSTANT_ID> :: <CONSTANT_ID> <NEWLINE> <INDENT1> def <
    SPACE> <FUNCTION> <NEWLINE> <INDENT1> <INSTANCE_VAL> <
    SPACE> = <SPACE> <CONSTANT_ID> :: <CONSTANT_ID> . <ID>
    ( <ID> [ <LITERAL> ] ) <NEWLINE> <INDENT0> <COMMENT> <
    NEWLINE> <INDENT0> if <SPACE> <INSTANCE_VAL> . <ID> ( <
    ID> . <ID> ( <LITERAL> ) . <ID> ( <LITERAL> , <SPACE> <
    LITERAL> ) ) <NEWLINE>
2 <INDENT1> def <SPACE> <FUNCTION> <NEWLINE> <INDENT1> <
    INSTANCE_VAL> <SPACE> = <SPACE> <CONSTANT_ID> :: <
    CONSTANT_ID> . <ID> ( <ID> [ <LITERAL> ] ) <NEWLINE> <
    INDENT0> <COMMENT> <NEWLINE> <INDENT0> if <SPACE> <
    INSTANCE_VAL> . <ID> ( <ID> . <ID> ( <LITERAL> ) . <ID>
     ( <LITERAL> , <SPACE> <LITERAL> ) ) <NEWLINE> <INDENT1
    > <ID> <SPACE> <INSTANCE_VAL> , <SPACE> <LITERAL> : <
    SPACE> <LITERAL> <NEWLINE>
3 <INDENT1> <INSTANCE_VAL> <SPACE> = <SPACE> <CONSTANT_ID>
    :: <CONSTANT_ID> . <ID> ( <ID> [ <LITERAL> ] ) <NEWLINE
    > <INDENT0> <COMMENT> <NEWLINE> <INDENT0> if <SPACE> <
    INSTANCE_VAL> . <ID> ( <ID> . <ID> ( <LITERAL> ) . <ID>
     ( <LITERAL> , <SPACE> <LITERAL> ) ) <NEWLINE> <INDENT1
    > <ID> <SPACE> <INSTANCE_VAL> , <SPACE> <LITERAL> : <
    SPACE> <LITERAL> <NEWLINE> <INDENT-1> else <NEWLINE>
4 <INDENT0> <COMMENT> <NEWLINE> <INDENT0> if <SPACE> <
    INSTANCE_VAL> . <ID> ( <ID> . <ID> ( <LITERAL> ) . <ID>
     ( <LITERAL> , <SPACE> <LITERAL> ) ) <NEWLINE> <INDENT1
    > <ID> <SPACE> <INSTANCE_VAL> , <SPACE> <LITERAL> : <
    SPACE> <LITERAL> <NEWLINE> <INDENT-1> else <NEWLINE> <
    INDENT1> <ID> <SPACE> <LITERAL> <NEWLINE>
5 <INDENT0> if <SPACE> <INSTANCE_VAL> . <ID> ( <ID> . <ID> (
     <LITERAL> ) . <ID> ( <LITERAL> , <SPACE> <LITERAL> ) )
     <NEWLINE> <INDENT1> <ID> <SPACE> <INSTANCE_VAL> , <
    SPACE> <LITERAL> : <SPACE> <LITERAL> <NEWLINE> <INDENT
    -1> else <NEWLINE> <INDENT1> <ID> <SPACE> <LITERAL> <
    NEWLINE> <INDENT-1> end <NEWLINE>
6 <INDENT1> <ID> <SPACE> <INSTANCE_VAL> , <SPACE> <LITERAL>
    : <SPACE> <LITERAL> <NEWLINE> <INDENT-1> else <NEWLINE>
     <INDENT1> <ID> <SPACE> <LITERAL> <NEWLINE> <INDENT-1>
    end <NEWLINE> <INDENT-1> end <NEWLINE>
7 <INDENT-1> else <NEWLINE> <INDENT1> <ID> <SPACE> <LITERAL>
     <NEWLINE> <INDENT-1> end <NEWLINE> <INDENT-1> end <
    NEWLINE> end <NEWLINE>
```

図 3.4. トークン化した後のソースコード例

では自然言語の文章の文脈をベクトルで表現するために Bi-Directional LSTM を利用している．Bi-Directional LSTM はシーケンス情報を前向きから学習する LSTM と後ろ向きから学習する LSTM を用意する．前向きの LSTM はシーケンスの先頭から順番に情報を入力し，出力層に情報を出力する．後ろ向きの LSTM はシーケンスの末尾から順番に情報を入力し，出力層に情報を出力する．Bi-Directional は前方向からのみ情報を記憶する通常の LSTM に比べて多くの情報を記憶することができるため，より精度の高い尤度を計算できると考え，今回 Bi-Directional LSTM を採用することにした．Bi-Directional LSTM は自然言語処理の分野のタギングやラベリングの研究でもニューラルネットワークの一部として利用されている [16][17].

図 3.5 では “`class <SPACE> <CLASS> ::  <CONSTANT ID>`” というトークンの並びを例にしている．各々のトークンは ID が割り振られ，その ID を示す one hot ベクトルとして扱われる．各 one hot ベクトルを embedding ネットワーク入力することでトークンのベクトル表現に変換される．そして変換されたベクトル表現を LSTM に入力する．例えばシーケンスの中央に位置する`<CLASS>`というトークンの尤度を計算する処理について考える．前向きの LSTM では`<CLASS>`の前に存在する `class` と`<SPACE>`が入力された LSTM の出力 C_f を使う．後ろ向きの LSTM では`<CLASS>`の後ろに存在する`<CONSTANT ID>`と`::`が入力された LSTM の出力 C_b を使う．この C_f と C_b を結合させたベクトルを Multi Layers Perceptron (MLP) に入力して，次元を落とした後，その出力ベクトル C を Softmax 関数で確率表現に変え，各トークンの尤度を計算する．softmax 関数は複数の値が入力された際に，全ての値を足すと 1 になるような値に変換して出力する．この関数を Bi-Directional LSTM の最後に用いることによって出力値を確率，すなわち尤度として扱うことができるのである．softmax 関数を用いた Bi-Directional LSTM は予め各単語に割り振られている ID をインデックスにした尤度の配列をベクトルとして出力する．学習時にはシーケンス中で実際に用いられているトークンの尤度が最も高くなるように学習する．逆に検査時には実際に用いられているトークンの尤度をその単語のインデックスを使って抽出する．

3.4　尤度に基づいたコーディングスタイルの適合検査と可視化

本節では 3.3 節で述べた方法で得られた学習済みの Bi-Directional LSTM を使って，コーディングスタイルの適合を自動で検査する方法を紹介する．本手法では学習済みの Bi-Directional LSTM をコーディングスタイルの尤度関数とみなし，コーディングスタイルの適合検査を行いたいソースコードの各トークンの尤度を計算する．学習済みのネットワークでは Softmax 関数からコーパス中の全てのトークンの尤度が計算される．その中から実際にシーケンス中で使われているトークンの尤度を抽出し，それをそのトークンの尤度とする．

また本手法では学習時と適合検査時のトークンの状態をできるだけ同じ状況にして尤度を計算するため，適合検査時にも 3.3 節で触れた Window Slide Argolithm を用いて，学習時と同じウィンドウサイズで検査対象のソースコードを分割している．そのため各トークンの尤度を求めた際に重複して尤度を計算されるトークンが存在する．そこで一度分割されたソースコー

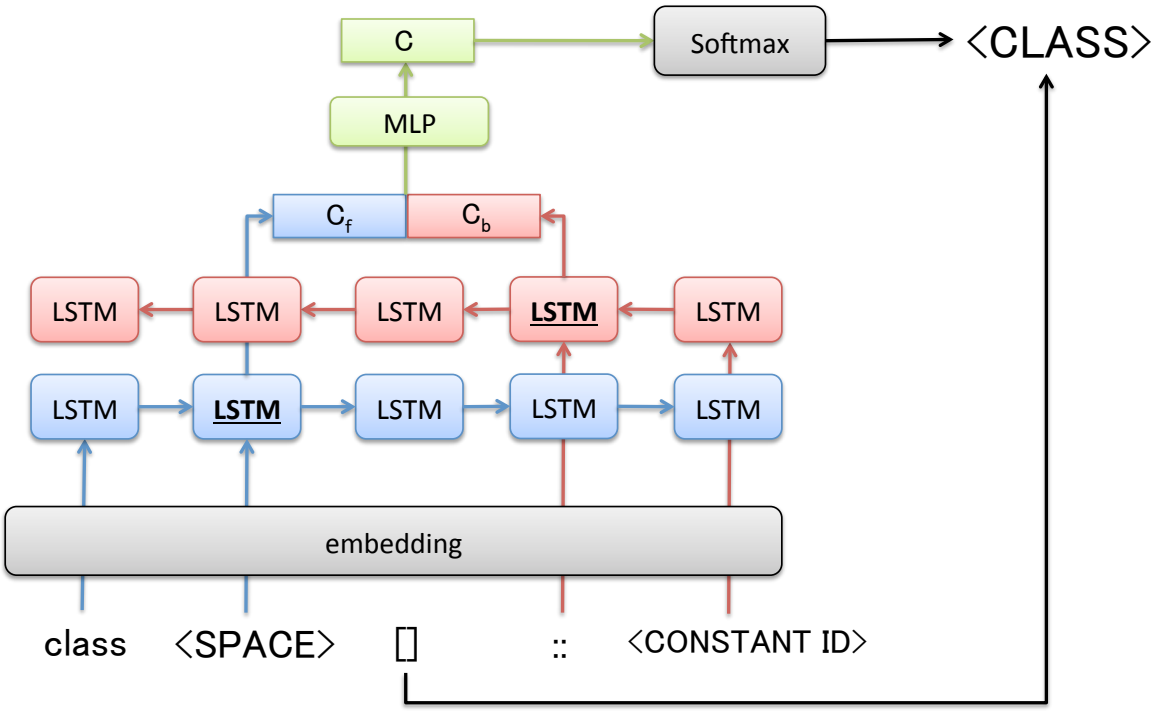

図 3.5. Bi-Directional LSTM のネットワーク構成図

ドを再度 1 つに統合するため，同一であるトークンを探しだし，複数の尤度が求められているトークンはそれらの平均の尤度をトークンの尤度としている．

本手法ではこのように各トークンの尤度を計算し，尤度が低いと見積もられたトークンを正しいコーディングスタイルに沿わないトークンと判断し，ユーザーに可視化する．ユーザーがソースコードのどの部分が不適切なコーディングスタイルを使っているか認識するために検査結果の可視化を行う．可視化はヒートマップを使って行う．図 3.6 にオープンソースプロジェクトの一つである Redmine[18] 内のソースコードの一部の可視化の結果を示す． 3.3 節で学習したネットワークを用いて， 3.4 節で説明したように尤度を各トークン毎に計算する．計算の結果，尤度が高いと判定されたトークンは背景が白に近くなり，尤度が低いと判断されたトークンは赤黒い背景になっていく．尤度の値によって背景色にコントラストをつけており，色の濃いトークンほど尤度が低い，すなわちコーディングスタイルが誤っている箇所であることを示している．

```
require 'uri'
require 'cgi'

class Unauthorized < Exception; end

class ApplicationController < ActionController::Base
  include Redmine::I18n
  include Redmine::Pagination
  include Redmine::Hook::Helper
  include RoutesHelper
  helper :routes

  class_attribute :accept_api_auth_actions
  class_attribute :accept_rss_auth_actions
  class_attribute :model_object

  layout 'base'

  protect_from_forgery

  def verify_authenticity_token
    unless api_request?
      super
    end
  end
```

図 3.6. ヒートマップを使って可視化されたソースコードの例

第 4 章

実験

本章では 3 章で提案した手法がどれだけコーディングスタイルを検査するのに有用であるかを示すために行った実験について述べる．実験では 2.3.2 節で触れたルールベースの Ruby の静的解析ツールである Rubocop との比較を行う．比較するためのデータとしてはクックパッド株式会社の協力を得て，クックパッド株式会社が github 上で公開しているオープンソースプロジェクトを学習データとして，クックパッド株式会社が社内に保有する内部データを検査対象のデータとした．実験環境は Ubuntu release 16.04，Intel(R) Xeon(R) CPU E5-2637 v3 @ 3.50GHz である．

4.1　実験の概要と目的

実験では本手法に則って実装したシステムが 2.3.2 節で紹介したルールベースの Ruby の静的解析ツールである Rubocop の解析結果にどれだけ近い結果が出せるかどうかで提案手法の有用性を示すことを目的とする． 2.5 節でも触れたようにルールベースの静的解析ツールはルールの設定などに人手がかかる分，設定されたルールについて精度は 100% の検査結果を出すことができる．一方の我々の提案手法は人手をかけずにコーディングスタイルの検査をすることが可能である．そこで提案手法を用いて実装されたシステムが Rubocop と同じかそれに近い精度で検査が可能となれば従来まで人手をかけていたコストを除いて，コーディングスタイルの検査をすることを可能にしたと考えられ，本提案手法の有用性が示されると考えた．

本実験では Rubocop の判定結果を正解として，提案手法を用いて実装されたシステムがどれだけの精度を出せるのかを Precision，Recall，F 値の 3 つの値を使って示す．この 3 つの値は分類問題などの予測結果の評価尺度の 1 つである．コーディングスタイルの誤りがあると指摘したトークンを誤りトークンと呼ぶとすると，Precision はシステムが誤りトークンであると判定したトークンの内，Rubocop も誤りトークンであると指摘し，実際にコーディングスタイル的に誤りであった割合を指す．Recall は Rubocop が誤りトークンであると判定したトークンの内，システムが誤りトークンであると指摘できたトークンの割合を指す．F 値は Precision と Recall の調和平均であり，Precision と Recall から計算される．F 値の計算式は式 4.1 に示す．基本的に Precision と Recall はトレードオフの関係にある．

表 4.1. Rubocop で測定したクックパッド株式会社のソースコード中の誤りトークンの比率

ソースコード群の種類	全トークン数	違反検知数
オープンソースコード群	481419	461
内部ソースコード群	324518	1476

$$F\ 値 = \frac{2Recall \cdot Precision}{Recall + Precision} \tag{4.1}$$

4.2　実験データの準備

実験データは 2 種類のソースコード群が必要になる．1 つは学習用データとして利用するためのコーディングスタイル的に誤りの無いソースコード群である．もう 1 つは検査用データとして利用するためのコーディングスタイル的に誤りを含むソースコード群である．今回はより実用に則した実験にするため，クックパッド株式会社の協力を得て，クックパッド株式会社が保有するソースコード群を使用して実験することにした．クックパッド株式会社のオープンソースプロジェクトは github 上からダウンロードし，研究室内のサーバーで解析を行った．一方，内部のソースコードは秘密保持などの観点から研究室で解析することが難しく，今回は共同研究という形をとり，インターンシップ生としてオフィス内からクックパッド株式会社から保有するコンピューター上にアクセスして，そのコンピューター上で解析を試みた．

クックパッド株式会社でヒアリングした結果，クックパッド株式会社が保有するソースコードの中でもクックパッド株式会社のオープンソースプロジェクトとして外部に公開されているソースコード群は比較的クックパッド株式会社のコーディングスタイル規約に合致したソースコードが多く含まれており，逆に内部で保有するソースコード群はコーディングスタイル規約に反したソースコードが多いことがわかった．実際に Rubocop をクックパッド株式会社のコーディング規約 [1] を基にカスタマイズし，オープンソースプロジェクトのソースコード群と内部のソースコード群で測定してみたところ表 4.1 のようになった．

今回の実験ではクックパッド株式会社が github 上で公開 [19] している以下 6 つのオープンソースプロジェクトの ruby ソースコードを学習データとして利用している．

- barbeque
- blouson
- dokumi
- garage_client
- gem_collector
- kuroko2

また学習用のデータとして入力したデータ量については表 4.2 に記載している．なお今回実験を行うにあたり，学習の精度をより高めるためオープンソースコード群の中で Rubocop に検

表 4.2. 実験で入力した学習用の Ruby ソースコードのデータ量

ファイル数	行数	トークン数
507	21439	481419

表 4.3. 実験で入力した検査用の Ruby ソースコードのデータ量

ファイル数	行数	トークン数
74	8233	324518

知された違反については 2 つの方法で誤りトークン数完全に存在しない状態にし，実験を行うことにした．Rubocop には自動修正機能が実装されており，一部の修正が容易な違反検知を取り除くことができる．まずこの自動修正機能を利用して修正できる箇所を一部修正した．しかしこの方法は自動修正が可能な簡単な違反のみしか修正されないため，一部の違反は残る結果となる．実際に今回のオープンソースコード群に自動修正機能を適用したところ，461 件あった違反検知数が 189 件に減った．そこで残存した一部 Rubocop のルールについては実験の検査の際に検査対象として除外して実験を行った．これはソースコードは先程自動修正した状態で残った違反検知を出しているルールを検査の際にルールとして除外することによってオープンソースコード群がもっとも理想的な学習データと仮定できる状態にするためである．

検査用のデータは外部には公開されていないクックパッド株式会社の内部ソースコードを利用した．今回利用したのはクックパッド株式会社の主要ウェブサービスであるクックパッド株式会社のモバイル端末に対して表示する Rails プロジェクト内のアプリケーション部分を担う app フォルダ内の Ruby ソースコードを利用した．実際に検査したソースコードのデータ量については表 4.3 に記載している．

4.3　Rubocop のルール設定

2.3.2 節で紹介したように Rubocop はある決められたフォーマットの YAML 形式のファイルを用意することで Rubocop がチェックするコーディングスタイルの種類を選択したり，採用しているコーディングスタイルの変更したりすることができる．

そこで今回クックパッド株式会社が社内で共有している企業専用のコーディングスタイル規約 [1] を基に Rubocop の設定ファイルを作成した．クックパッド株式会社のコーディングスタイルの中にはそのスタイルの重要度により `MUST` と `SHOULD` のレベル分けがなされている．今回の実験では `MUST` レベルのコーディングスタイルルールのもののみを設定ファイルに登録し，検査するようにしている．またコーディングスタイル規約に記述されたものの中で Rubocop の設定ファイルでは記述できないものについては今回は検査対象から除外している．また今回 Rubocop の設定ファイルに設定したルールの一覧とそのルールの内容をを以下に列挙する．またルールの横の括弧内に表記されているのはユーザーが選択できるルールのパラメータの内，今回実験で選択したものを示している．

- Style
  - EmptyLiteral: 空のリテラルを定義する際に，new を使わない．
  - TrailingCommaInArguments (comma): 引数の定義が複数行に渡る場合，最後の引数の後ろにもカンマを付与する．
  - TrailingCommaInLiteral (comma): 複数行に渡る配列やハッシュの定義で最後の要素の後ろにもカンマを付与する．
  - MultilineBlockChain: 長いメソッドチェインの最後のメソッド呼び出しでブロックを渡す場合，最後のメソッド呼び出しのレシーバをローカル変数として抽出し，ブロック付きメソッド呼び出しを独立した式として書く，
  - NumericLiterals: 桁数が大きな 10 進法の数値リテラルは 3 桁毎に下線を入れる．
  - HashSyntax (ruby19): ハッシュの定義で Ruby1.9 から導入された `key: value` 形式をつかう．
  - AndOr: 論理演算子の `and` や `or` を使わず，代わりに`&&`や`||`を使う．
  - NestedTernaryOperator: ネストした三項演算子は記述しない．
  - MultilineTernaryOperator: 複数行にまたがる三項演算子は記述しない，
  - NegatedIf (both): 条件が否定のときには `if` ではなく，`unless` を利用する．
  - NegatedWhile: 条件が否定のときには `while` ではなく，`until` を利用する．
  - UnlessElse: `unless` に対して，`else` を使用しない．
  - MultilineIfThen: 複数行にまたがる `if` に対して `then` を使用しない．
  - WhileUntilDo: 複数行にまたがる `while` 及び `until` では，`do` を使用しない．
  - ConditionalAssignment (assign_to_condition): 制御構造が代入の右辺に来る場合は，本体コードのインデントを 2 レベル下げ，制御構造の終わりの end 行のインデントを本体コードより 1 レベル戻す．
  - RedundantReturn: `return` が意味のある値を返さない場合は、後ろに式を書かない．
  - MethodCallWithoutArgsParentheses: 引数なしのメソッド呼び出しは括弧を省略する．
  - BeginBlock: `BEGIN` ブロックは使わない．
  - EndBlock: `END` ブロックは使わない．
  - Alias (prefer_alias_method): メソッドのエイリアスを定義する場合は表記揺れを防止するため，`alias` ではなく `alias_method` を使用する．
  - Attr: アクセサを定義する場合は `attr_accessor`，`attr_reader`，`attr_writer` を使用し，`attr` は使用しない．
  - GlobalVars: グローバル変数を新たに導入しない．
  - ClassVars: クラス変数を使用しない．代わりに `class_attribute` を使用する，
- Layout
  - TrailingWhitespace: 行末に空白を置かない．
  - IndentArray (consistent): 代入の後ろに配列リテラルを複数行で書く場合は，[の

後ろで改行し，要素行のインデントを 1 レベル下げ，] は独立した行に置いて [を書いた行の行頭にインデントを揃える．
  - SpaceInsideHashLiteralBraces: ハッシュリテラルを一行に書くときは，{ と第一要素のキーとの間，および最終要素の値と } との間にそれぞれ空白を一つずつ入れる．
  - IndentHash (consistent): 代入の後ろにハッシュリテラルを複数行で書く場合は，{ の後で改行し，要素行のインデントを 1 レベル下げ，} は独立した行に置いて { を書いた行の行頭にインデントを揃える．
  - SpaceAroundOperators: 演算子の両側に空白を入れる．
  - SpaceAroundEqualsInParameterDefault: 代入記号の両側に空白を入れる．
  - SpaceAfterMethodName: メソッド名とメソッド呼び出しの括弧の間に空白を入れない．
  - BlockEndNewline: ブロック付きメソッド呼び出しでは `do/end` 記法でブロックを書く．
  - SpaceBeforeBlockBraces (space): 中括弧によるブロックを 1 行で書く場合は，ブロックパラメータと本体コードの間に空白を 1 つずつ入れる．
  - SpaceInsideBlockBraces (space): 中括弧によるブロックを 1 行で書く場合は，{ と本体コードの間、および本体コードと } の間に空白を 1 つずつ入れる．
  - AccessModifierIndentation: `private` や `protected` や `public` を引数なしで使用する場合，インデントレベルはメソッド定義と同じレベルにする．
  - EmptyLinesAroundAccessModifier: `private` や `protected` や `public` を引数なしで使用する場合，前後に 1 行ずつ空白を入れる．
- Lint
  - AssignmentInCondition: 条件式で代入式を書かない．
  - BlockAlignment (either): `do/end` によるブロックでは，`do` の前後に空白を 1 つ入れ，ブロックパラメータの後で改行し，`end` は独立した行に書く．ブロック本体のインデントは 1 レベル下げ、`end` のインデントはメソッド呼び出しの 1 行目にあわせる．
  - ParenthesesAsGroupedExpression: メソッドと引数リストの括弧の間に空白を置かない．
- LineLength (128): 一行は 128 文字以内に収める．

4.4　前処理を行うプログラムの実装

3.2 節でも述べたように尤度計算を行うニューラルネットワークに Ruby ソースコードを入力する際にはソースコードをトークン化する処理とトークン化したソースコードを分割する処理が必要となる．そこで今回，本手法を用いた実験をするためにこの 2 つの処理を行うプログ

ラムを実装した．

ソースコードをトークン化するプログラムは Ruby の字句解析器を改良する必要があるが，今回は Python の Pygments というモジュールが提供する Ruby Lexer を使って改良した Ruby の字句解析器を実装した．Pygments は Python のモジュールで従来は様々な言語のシンタックスハイライトを行うために利用される．シンタックスハイライトを行うにあたり，ソースコード中の文字列を字句解析する必要があるため，Pygments には Lexer というサブモジュールが用意されている [20]．今回の改良された字句解析器はこの Lexer に用意された Ruby Lexer を用いている．Pygments の Ruby Lexer はソースコードを入力すると字句解析を行い，トークン名とトークンの種類をタプルにした配列を返す．今回実装したプログラムではトークンの種類を順番に確認し，表 3.1 の対応に従ってトークン情報を書き換える．また Pygments の Ruby Lexer ではインデントや改行，空白の解析はできないため，それらが含まれるトークンについては実装プログラムで再度字句解析してトークン化した．その後，各行の先頭のインデントトークンについては前行とのインデントの差異を計算して，インデントトークンを相対的な情報に書き直している．

ソースコードの分割については分割のサイズを行数で指定できるような引数を用意し，改行情報を基に各ソースコードを指定された行数で分割している．分割されたトークン列は一行を分割された 1 つのトークンの塊として，各トークンを空白で区切った形でテキストファイルに保存される．ニューラルネットワークはこのテキストファイルを入力として受けとり，尤度計算を行う．

4.5　ニューラルネットワークの実装

本提案手法ではニューラルネットワークを構築し，ソースコードを入力する必要がある．今回の実験では Python のニューラルネットワーク構築のライブラリとして Preferred Networks から提供されている Chainer[21] を利用して Bi-Directional LSTM のネットワークを構築した．Chainer は GPU を利用した高速な並列計算が可能な他，様々な形のニューラルネットワークを柔軟に記述することができる．自然言語分野の研究でも利用されることが多く，Chainer 内に自然言語を扱うためのモジュールが多く揃っているため，今回はニューラルネットワークの構築に Chainer を採用した．また GPU などを利用してより高速に学習，尤度計算が行えるように cupy ライブラリを利用した．cupy は CUDA 上で簡単に NumPy を用いた数値計算が行えることを実現したライブラリである．cuda とは半導体メーカの NVIDIA 社が提供する GPU コンピューティング向けの統合開発環境で，PC 上でより簡単に GPU 計算を実装できるようになる．NumPy は Python に用意された数値計算をより効率よく行うための拡張モジュールである．cupy を使えば，CUDA 上で NumPy のように数値計算が行えるため，より効率的に高速な並列計算を簡単に行うことが可能となる．また今回ニューラルネットワークを GPU 上で実装するため cuDNN という同じく NVIDIA 社が提供しているディープラーニング用のライブラリも利用した．cuDNN を使うことでディープラーニングの計算速度を向上させることが可能となる．以下に各モジュールの利用バージョンを記載する．

表 4.4. 実装に利用したモジュールのバージョン

モジュール名	バージョン
Python	2.7.12
Chainer	1.21.0
cuda	8.0
cudNN	5.1.5

今回は当専攻中山研究室の西田典紀氏が Chainer で実装した LSTM を参考に Bi-Directional LSTM の実装を行った．ニューラルネットワークの学習を高速にするためのユーティリティプログラムは西田氏が実装したプログラムを利用し，Bi-Directional LSTM のネットワークは我々が実装した．また Bi-Directional LSTM の出力結果から尤度計算を行う部分の実装については chainer モジュールから提供されている softmax 関数を利用して尤度計算を行った．また今回のニューラルネットワークでは層の次元数として変更できるパラメータは入出力層と隠れ層の次元数として，隠れ層の状態を対比する層の次元数は隠れ層の 4 倍の次元数とした．

4.6　比較プログラムの作成

3 章で提案した手法では 3.4 節で紹介したように検査の結果をヒートマップにして可視化しているが，これでは Rubocop の結果と比較ができない．また Rubocop はコーディングスタイルが間違っているかそうでないかの二値で判定しているのに対して，提案手法では尤度を使ってコンストラクトをつけて間違いの度合いを表示しているので比較が難しい．そこで今回の実験では実装したシステムを改良し，閾値を設定し，尤度がその閾値を下回ったトークンを間違ったトークンとして判定している．アウトプットとしては json ファイルに判定したトークンの情報を記録しており，Rubocop にはデフォルトで json ファイル形式で間違ったトークンの情報を記録する機能が存在するため，システムの出力した json ファイルと Rubocop の json ファイルを解析して，Precision，Recall，F 値をそれぞれ計算するプログラムを実装した．

システムの出力する json ファイルのフォーマットは図 4.1 のようになっている．プログラムを実行すると解析されたプログラムの名前がファイル名になった json ファイルが出力される．`points` の項目にはシステムが間違っていると指摘したトークンの行数とその行の何トークン目かが格納されたペアの配列が格納されている．一方の Rubocop が生成する json ファイルのフォーマットは図 4.2 のようになっている．こちらも出力の際に解析対象のファイル名が json のファイル名になっている．こちらは `metadata` の項目には rubocop や対象としている ruby のバージョンが記載されており，`files` の項目に Rubocop の検査の結果が格納されている．`path` には検査対象のファイルパスが格納され，`offenses` にこのファイルでの Rubocop が指摘する規約違反が列挙される．各規約違反にはどのルールに違反し，どのような修正を行

```
1 {"points": [
2       [24, 10],
3       [24, 11],
4       [24, 12],
5       [24, 13],
6       [24, 14],
7       ...
8   ]
9 }
```

図 4.1. システムが出力する json フォーマット

えばいいのかが記載されている他，`location` の項目にソースコードのどの部分の指摘なのかが記載されている．今回の比較プログラムではこの `location` の情報を利用している．

さらに比較を行うために検査対象の ruby プログラム群をトークン化する．ただしこの時に 3.2 節で紹介したようなトークンの抽象化は行わない．今回のトークン化はあくまで Ruby ソースコードのトークンの構成を把握することを目的とするため，トークンの内容を抽象化することはせず，トークンの文字列とトークンのソースコード内での位置情報を json 形式で保存している．トークン化されたソースコードの一例を図 4.3 に示す．全体の構造は辞書形式になっており，プログラム名をキーにしてプログラムソースコード内のトークンの位置情報が配列として並んでいる．位置情報の配列は行毎にネストされていて，行毎の配列内にはその行内に並んでいる各トークンの情報が並んでいる．各トークンの位置情報は 0 番目がトークンの文字列情報，1 番目がその行の何文字目から始まるか，2 番目がその行の何文字目に終了するかが記されている．トークンの位置情報を文字単位で表したのは図 4.2 に示したように Rubocop の出力フォーマットが文字列単位で示されていることに対応するためである．Rubocop の出力フォーマットが文字列単位なのはアラートの種類によっては複数のトークンにまたがる場合があるためである．

比較プログラムはシステムから出力された json ファイルと Rubocop から出力された json ファイル一覧を走査して，まず同じプログラムを対象にしたシステム出力の json ファイルと Rubocop から出力された json ファイルのペアを作る．そしてそのプログラムのトークン情報を先程トークン化した位置情報ファイルから検索し，トークンの位置情報を得る．システムから出力された json ファイルはトークンの行数とその行内の何番目かが記載されているので，配列のインデックスからどのトークンを示しているかを計算する．一方，Rubocop の出力した json は何行目のトークンかは記載されているが，その行の中のどこにあるかは文字数で示されているため，どのトークンが Rubocop によって間違いであると指摘されているかを文字数の位置情報から計算する必要がある．

こうして今回実装したシステムと Rubocop がそれぞれどのトークンに対して，アラートを出したのかを登録した配列を Rubocop のものと実装したシステムのものの両方作る．配列の中身は 0 か 1 であり，ソースコードの左上のトークンが配列の 0 番目で一番最後のトークンが配列の末尾に対応する．0 はそのトークンにアラートがでていないことを示し，1 はアラー

```
1  {"metadata":
2      {"rubocop_version":"0.49.1",
3       "ruby_engine":"ruby",
4       "ruby_version":"2.1.2",
5       "ruby_patchlevel":"95",
6       "ruby_platform":"x86_64-darwin13.0"},
7   "files":[
8       {"path":"<filepath>",
9        "offenses":[
10           {"severity":"convention",
11            "message":"<error␣message>",
12            "cop_name":"<cop␣name>",
13            "corrected":false,
14            "location":{
15                "line":42,
16                "column":32,
17                "length":1
18            }
19           },
20          {"severity":"convention",
21           "message":"<error␣message>",
22           "cop_name":"<cop␣name>",
23           "corrected":false,
24           "location":{
25               "line":43,
26               "column":48,
27               "length":8}
28          }
29       ]
30      }],
31      "summary":{
32          "offense_count":2,
33          "target_file_count":1,
34          "inspected_file_count":1
35      }
36 }
```

図 4.2. Rubocop が出力する json フォーマット

トがでていることを示す．双方の配列を生成したら，Precision，Recall，F 値を計算する．今回これらの値を計算するのに Python のモジュールとして提供されている scikit-learn[22] の classification_report 関数を利用した．scikit-learn はオープンソースの機械学習ライブラリであり，機械学習のプログラムを実装するにあたり必要となる様々な数値計算を行える関数が用意されている。今回利用した classification_report は先程生成した 0 と 1 から成る配列を引数にして Precision，Recall，F 値を自動で計算してくれる．今回の比較プログラムでは各プログラムファイルの結果を列挙した後，最後に全てのプログラムの値を合算した結果を計算し，テキストファイルとして保存している．結果の表示の一例を図 4.4 に示している．まず検査対

```
1  {
2      "test/pagination_helper_test.rb": [
3          [
4              ["#␣Redmine␣-␣project␣management␣software", 0,
                   39],
5              ["\n", 39, 40]
6          ],
7          [
8              ["#␣Copyright␣(C)␣2006-2017␣␣Jean-Philippe␣
                  Lang", 0, 45],
9              ["\n", 45, 46]
10         ],
11             ...
12
13         [
14             ["require", 0, 7],
15             ["␣", 7, 8],
16             ["File", 8, 12],
17             [".", 12, 13],
18             ["expand_path", 13, 24],
19             ["(", 24, 25],
20             ...
```

図 4.3. プログラムの構成を示すトークン情報

象となったプログラムのファイルパスが記載され，その下に Precision，Recall，f 値，正解の数が記載されている．正解の数というのは今回正解データとした Rubocop が示したトークンの数である．`Correct token` はアラートがでなかったトークン，すなわちコーディングスタイル的に正しいとされたトークンで `Wrong token` はアラートがでたトークン，すなわちコーディングスタイル的に間違っていると指摘されたトークンである．今回は実装したシステムが如何に Rubocop に近いアラートが出せるかどうかを指標としているので `Wrong token` の行の値を実装したシステムの結果として扱う．また今回，我々が実装したシステムはトークン単位でのコーディングスタイルの間違いを指摘するのに対し，Rubocop はルール単位でのコーディングスタイルの間違いの指摘をするため，ルールの中には複数のトークンに跨って間違い検知を受ける場合があり，本システムと Rubocop で指摘の範囲が異なってしまうケースがあった．そこで今回の比較では Rubocop と本システムの間違いを検知した範囲が重複していた場合には同じ箇所を検知したものとして比較を行った．また今回実装したシステムでは設定されたウィンドウサイズより短い行数のソースコードは解析が行えないため，それらのソースコードは解析結果から除外して結果を算出している．

```
1  ../intra/mobile_verify/app/helpers/support_requests_helper
     .rb
2
3                  precision    recall  f1-score   support
4
5  Correct token       0.90      0.92      0.91        83
6    Wrong token       0.65      0.62      0.63        21
7
8    avg / total       0.85      0.86      0.85       104
```

図 4.4. 比較結果の表示例

表 4.5. ニューラルネットワークのハイパーパラメータを調整した際の総合的な指標値

入出力層の次元数	隠れ層の次元数	Precision	Recall	F 値
150	256	0.19	0.24	0.21
100	256	0.19	0.26	0.22
100	128	0.19	0.27	0.22

4.7　実験結果

本節では 4.3 節， 4.4 節， 4.5 節で実装したシステムと 4.2 節で準備したデータを用いて我々の提案手法の実用性を示すために行った結果を述べる．実験ではニューラルネットワークのハイパーパラメータを調整した結果の比較と，ソースコード分割サイズを示すウィンドウサイズを調整した結果の比較，今回我々が実装したシステムの検査の閾値を調整した結果の比較を行っている．それぞれの実験の結果として検査対象の全てのソースコードを解析した結果の Precision，Recall，F 値の数値を算出した一覧と各ソースコードにおける Precision と Recall の相関を示すグラフ，ソースコード内に実在したコーディングスタイルの誤りと F 値の相関を示すグラフを掲載している．

最初にニューラルネットワークのハイパーパラメータを調整する実験を行った．この実験では Bi-Directional LSTM の入出力層の次元と Bi-Directional LSTM 中で利用されている隠れ層の次元を変更して比較を行った．どの実験もウィンドウサイズは 5，閾値は 0.05 にして実験を行っている． 4.2 節で用意した検査対象のソースコード群を全て解析した総合的な結果を次元数の値とともに表 4.5 に示す．また各ソースコードの検査結果についても解析を行い，その各数値をグラフにプロットしたグラフを図 4.5 と図 4.6 に示す．図 4.5 は Precision と Recall の相関を表したグラフであり，図 4.6 は誤りトークン数と F 値の相関を表したグラフである．実験の結果をみるとニューラルネットワークのハイパーパラメータを調整した際の実験ではそこまで各実験結果に差異は生まれなかった．

次にソースコードを分割するサイズを示すウィンドウサイズを調整する実験を行った．この実験ではウィンドウサイズの大きさを変更することでトークンの尤度，ひいてはコーディン

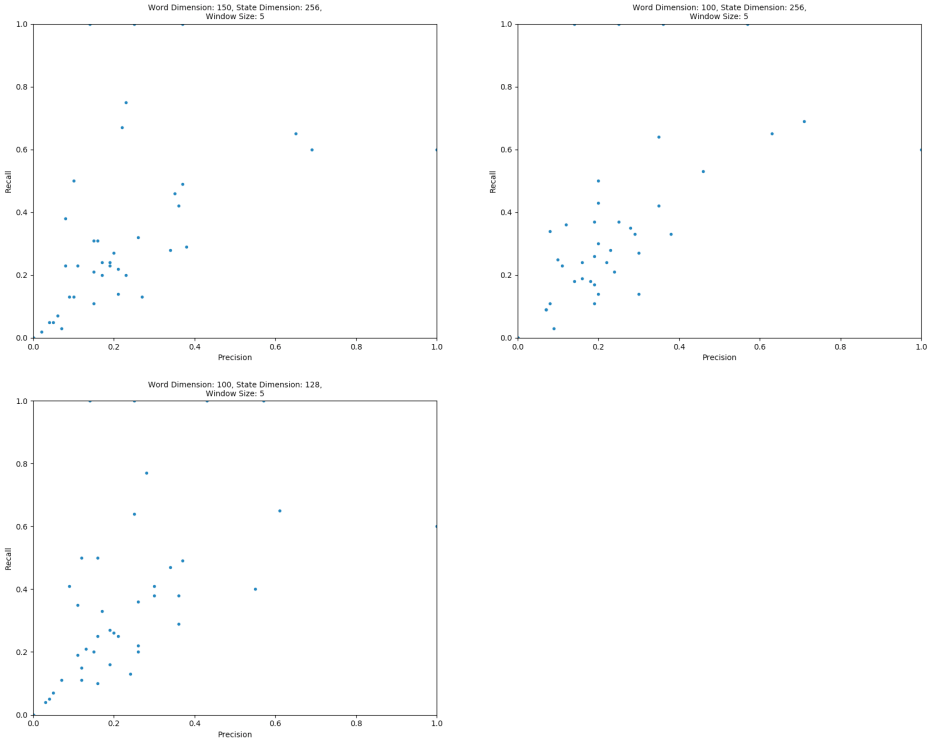

図 4.5. ニューラルネットワークのハイパーパラメータを調整した際の Precision と Recall の相関図

グスタイルの良い悪いを評価する際に考慮するトークンの数や前後の範囲がどれだけ精度に影響するのかを測定するために行った．どの実験も入出力層の次元数は 150，隠れ層の次元数は 256，閾値は 0.05 にして実験を行っている． 4.2 節で用意した検査対象のソースコード群を全て解析した総合的な結果を次元数の値とともに表 4.6 に示す．各々の値を棒グラフで比較したグラフを図 4.7 に示す．また先の実験と同じく，各ソースコードの検査結果についても解析を行い，その各数値をグラフにプロットしたグラフを図 4.8 と図 4.9 に示す．図 4.8 は Precision と Recall の相関を表したグラフであり，図 4.9 は誤りトークン数と F 値の相関を表したグラフである．ウィンドウサイズを調整した際の実験ではウィンドウサイズが 15 の際の実験結果は他のウィンドウサイズの値の実験に比べ，各指標値が高いことが表 4.6 から伺える．図 4.8 を見ても，ウィンドウサイズが 15 の際には特に Recall がソースコード全体でも大きくなっている．表 4.6 からはウィンドウサイズが小さくなるにつれ，各指標値が小さくなる傾向にあることがわかる．また図 4.9 をみると，他のウィンドウサイズでは誤りトークン数が大きくなると F 値が下がる傾向にあるが，ウィンドウサイズが 15 の時にはある程度一定の F

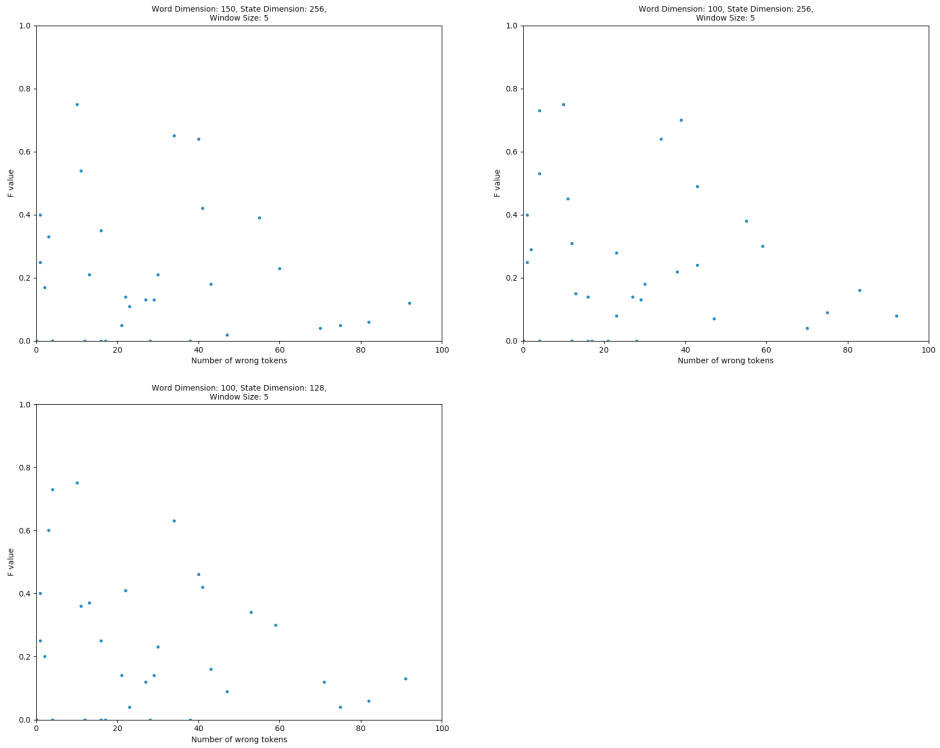

図 4.6. ニューラルネットワークのハイパーパラメータを調整した際の誤りトークン数と F 値の相関図

値を保てていることが確認できる．

最後に今回我々が実装したシステムの閾値を調整する実験を行った．この実験ではシステムが間違いトークンと判別する尤度の閾値を変更することでシステムの判定の厳しさを調整するもので，このパラメータの調整によってトレードオフ関係にある Precision と Recall の値が変わり，コーディングスタイルの検査システムとして最適な精度が得られると考えた．どの実験も入出力層の次元数は 150，隠れ層の次元数は 256，ウィンドウサイズは 15 にして実験を行っている．こちらの実験でも各ソースコードの検査結果についても解析を行い，各ソースコードの結果から得られた総合的な指標値を閾値とともに表 4.7 に示す．また表 4.7 の各値を棒グラフで表したものを図 4.10 に示す．その各数値をグラフにプロットしたグラフを図 4.11 と図 4.12 に示す．図 4.11 は Precision と Recall の相関を表したグラフであり，図 4.12 は誤りトークン数と F 値の相関を表したグラフである．図 4.11 及び図 4.12 を見るとそこまで二つの結果に差異はないように見える．一方で表 4.7 をみると Recall が下がる代わりに Precision が上がり，F 値についても改善が見られた．

表 4.6. ウィンドウサイズを調整した際の総合的な指標値

ウィンドウサイズ	Precision	Recall	F 値
15	0.31	0.95	0.47
13	0.23	0.42	0.30
10	0.26	0.40	0.32
8	0.24	0.34	0.28
5	0.19	0.24	0.21
3	0.19	0.26	0.22

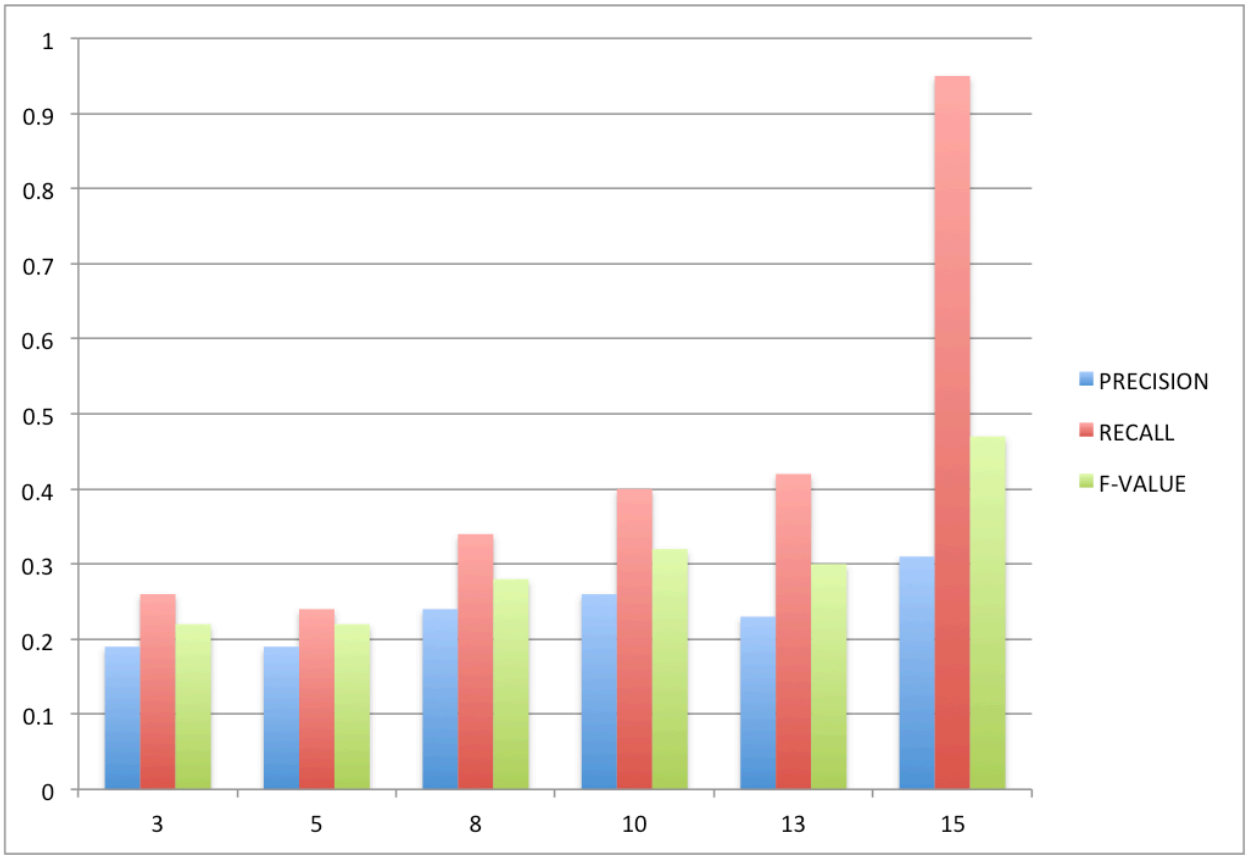

図 4.7. ウィンドウサイズを調整した際の各指標値

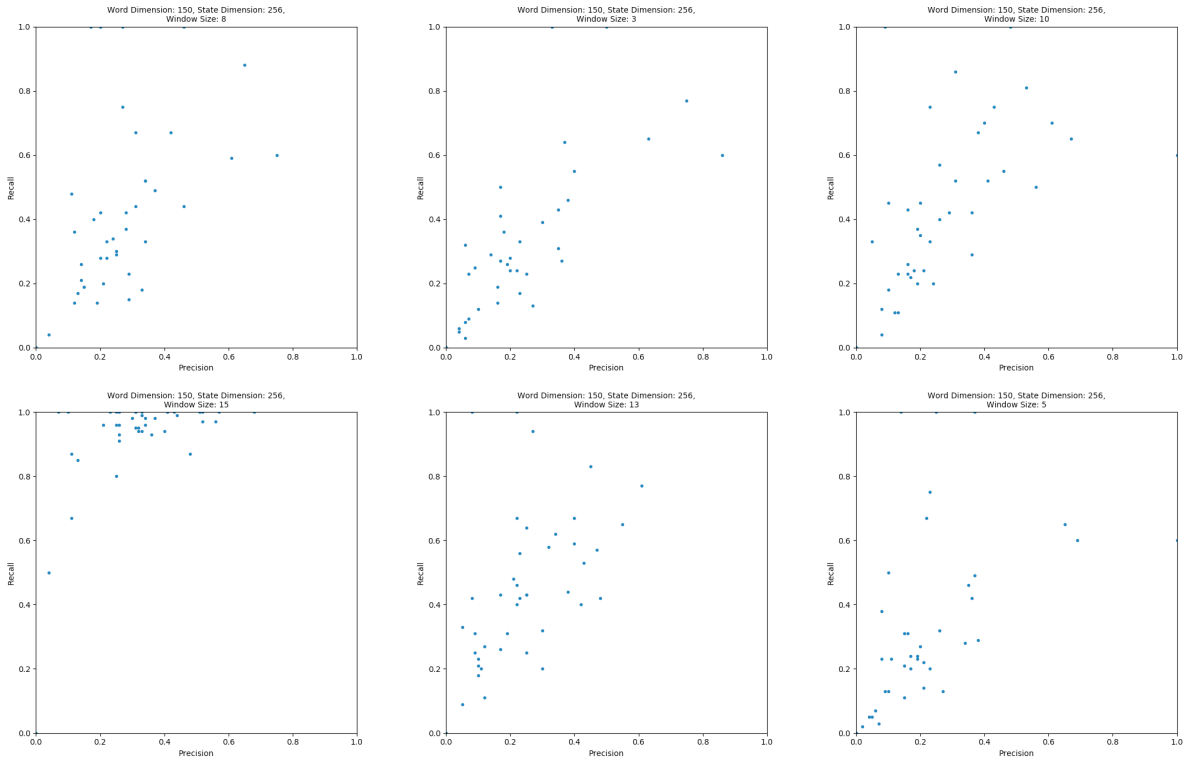

図 4.8. ウィンドウサイズを調整した際の Precision と Recall の相関図

結果として実験で選択したパラメータの中では 表 4.8 が最も最適なパラメータとなった．またそのパラメータを設定した際に得られる指標値は 表 4.9 である．全実験を通して Precision と Recall の相関は正の相関関係があると考えられる．これはつまり検査の結果，Precision が高かったソースコードでは Recall の値も高いということになる．この結果から考えられることとして今回我々が実装したシステムにはコーディングスタイルの検査を得意とするソースコードとそうでないソースコードがあると考えられる．またどの実験結果も Precision よりも Recall の方が大きくなる傾向にあることがわかる．

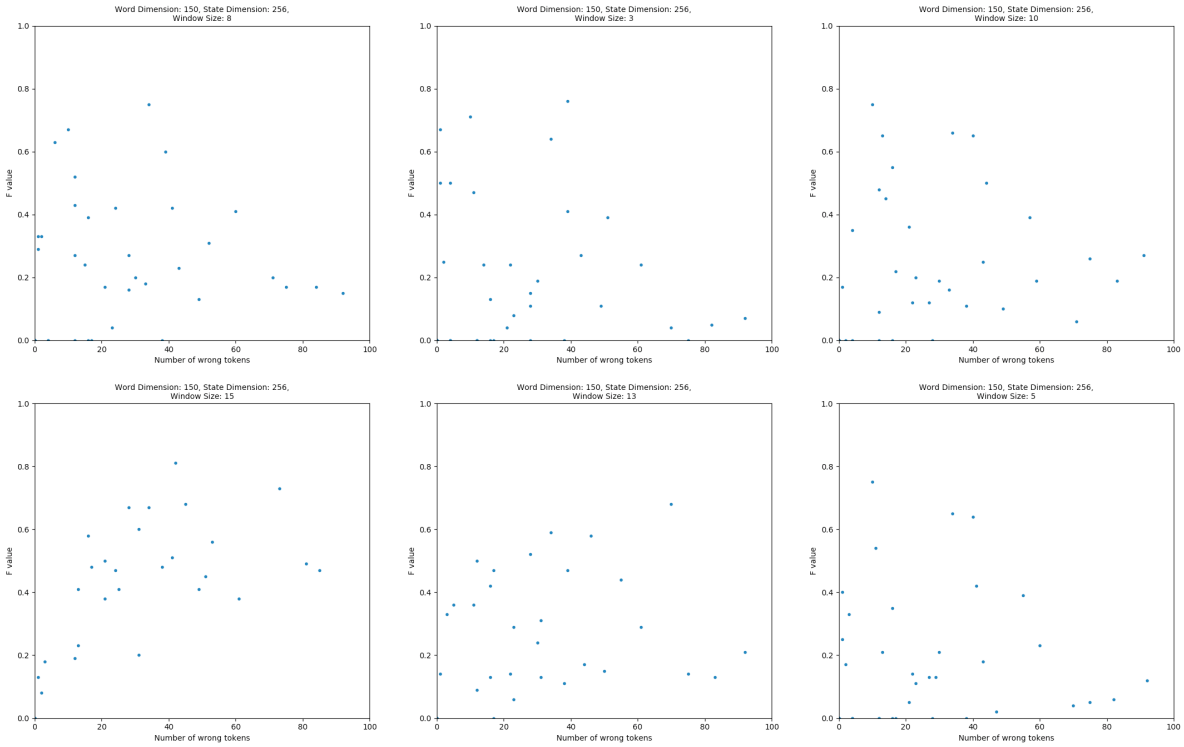

図 4.9. ウィンドウサイズを調整した際の誤りトークン数と F 値の相関図

表 4.7. 閾値を調整した際の総合的な指標値

閾値	Precision	Recall	F 値
0.0000005	0.48	0.3	0.37
0.000005	0.45	0.44	0.44
0.00005	0.43	0.58	0.49
0.0005	0.4	0.72	0.51
0.005	0.36	0.9	0.51
0.01	0.34	0.92	0.5
0.05	0.31	0.95	0.47

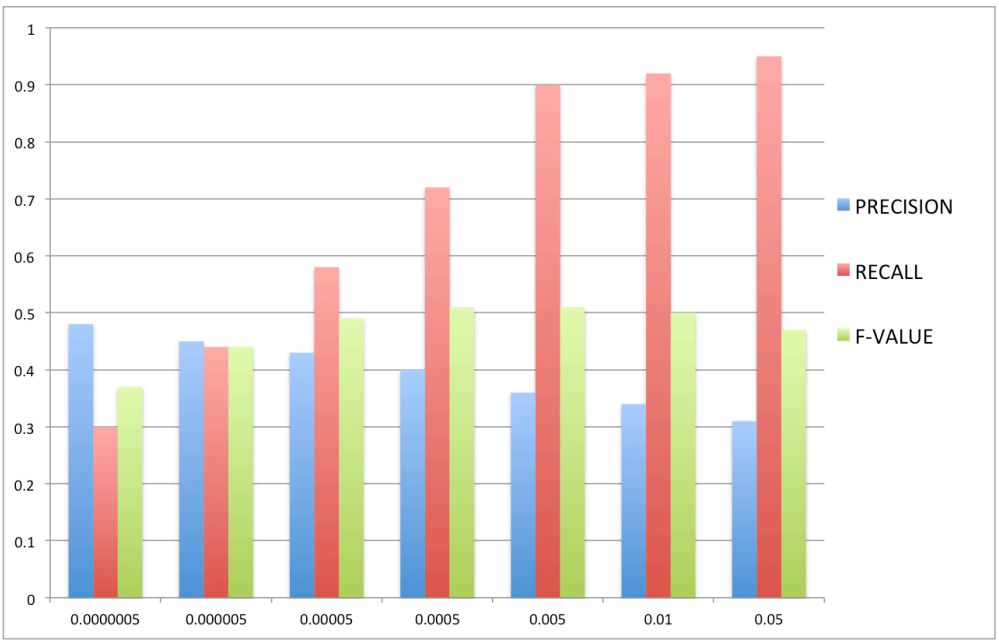

図 4.10. 閾値を調整した際の各指標値

表 4.8. 実験中に得られた最適なパラメータ

埋め込み層	隠れ層	ウィンドウサイズ	閾値
150	256	15	0.0005

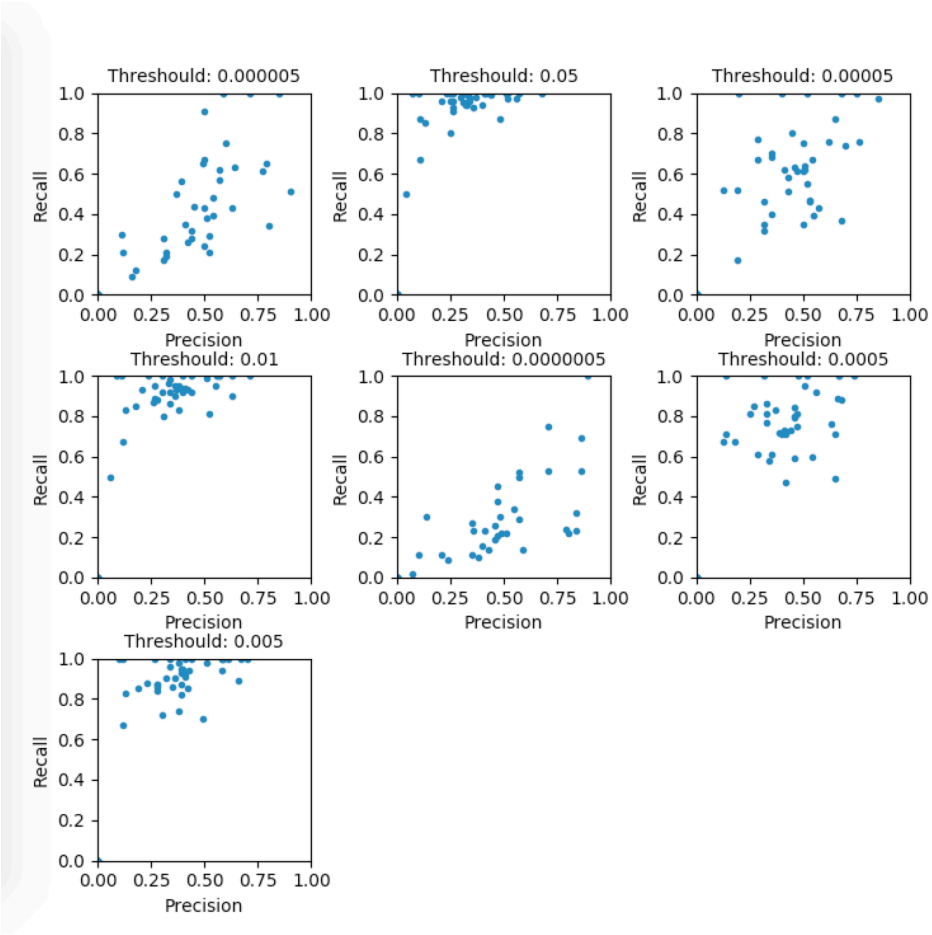

図 4.11. 閾値を調整した際の Precision と Recall の相関図

表 4.9. 実験中に得られた最大の指標値

Precision	Recall	F 値
0.40	0.72	0.51

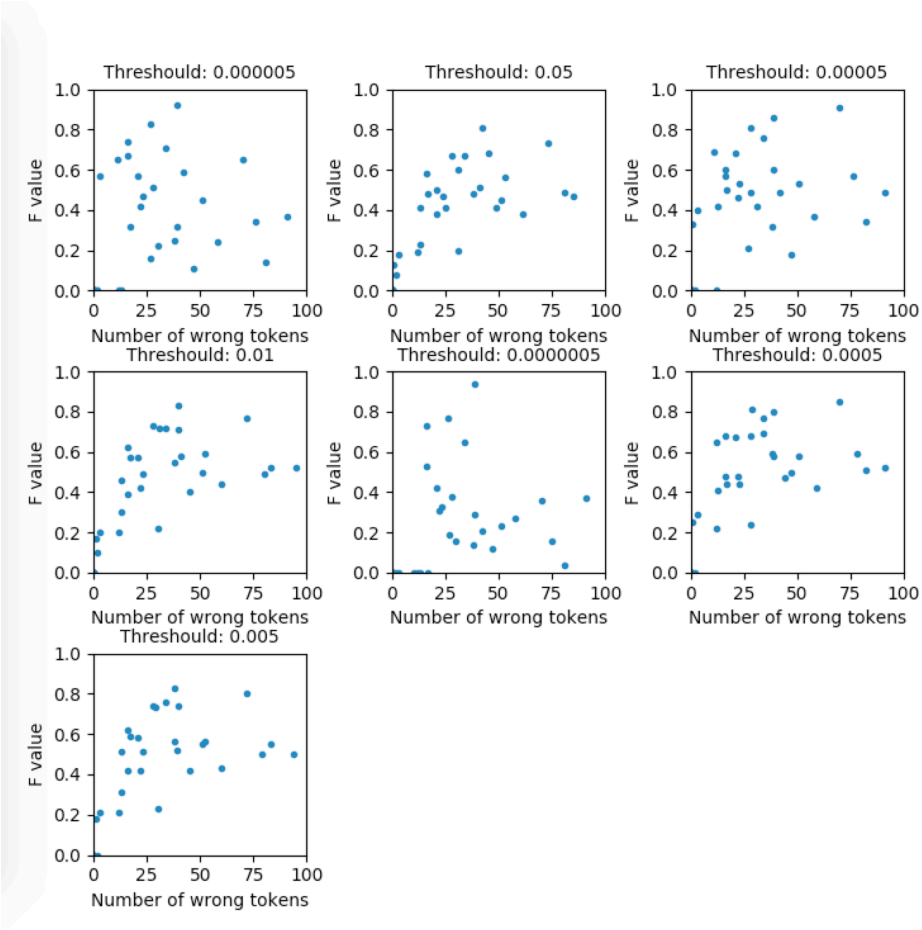

図 4.12. 閾値を調整した際の誤りトークン数と F 値の相関図

第 5 章

まとめと今後の課題

5.1 まとめ

本研究では，機械学習のアプローチを応用してソースコード内のコーディングスタイルを自動で検査する手法を提案した．近年，多くの企業や組織が複数人で大規模なソフトウェア開発を行うことが増え，それに伴い，開発効率を上げるためにコーディングスタイルを組織内で統一することが重要な課題の一つになった．しかし従来までコーディングスタイルの検査は手動で行われていたか，もしくはルールベースの静的解析ツールを用いて行われていたが，ドキュメントの作成や設定ファイルの作成，新しいメンバーの教育など運用面での開発チームのコストが高くなってしまう．一方で近年は機械学習分野での研究が著しい発展を遂げ，実用に耐えうるような学習精度でタスクをこなせるニューラルネットワークが続々と登場してきた．そんな中，これまでルールベースでのアプローチが基本的であったソフトウェア工学の研究分野でも機械学習のアプローチを使って，これまでの研究課題を解決する研究が増え始めてきている．

そこで我々は機械学習のアプローチを用いて，組織が保有する過去のソースコードからその組織や企業のソースコードのコーディングスタイルを学習し，尤度として表現することで開発チームに運用コストのかからないコーディングスタイルの自動検査を実現することができる手法を提案した．本手法はコーディングスタイルのルールの明文化や目視による検査など従来まで手動でやられていた検査や静的解析ツールの設定を行っていた運用コストを減らしつつ各組織にあったコーディングスタイル検査を行うことが可能となる．コーディングスタイルの尤度を計算する手法は自然言語処理の手法を応用し，プログラム中のトークンの並びから尤度を計算できるようなニューラルネットワークを利用している．また字句解析器を改良してコーディングスタイルの検査に必要な情報をソースコードから取得できるような前処理の方法についても提案した．

更に我々は Python のライブラリとして提供されている Chainer を利用して本提案手法を基にしたシステムを設計，構築した．またその実用性を検証するためにクックパッド株式会社の協力を得て，クックパッド株式会社が提供するオープンソースプロジェクトを学習データにし，内部で保有するソースコードを検査データとして実験を行った．この実験の結果によって

我々の手法にはまだ性能上の課題があることがわかった．各パラメータの調整を行った結果，ウィンドウサイズがシステムの精度に与える影響が大きいことがわかった，全実験において Precision と Recall には正の相関があることがわかった．この結果から今回実装したシステムに検出を得意とするコーディングスタイルとそうでないコーディングスタイルが存在すると考えられる．また測定の結果として Precision よりも Recall のほうが高くなる傾向があることがわかった．

5.2　今後の課題

今後の課題としては，提案手法がより実用に耐えうる指標値が出せるように改良を進めていく必要があると考えられる．指標値を改善する上でウィンドウサイズの調整は精度に大きく影響を与えると考えられる．また今回のこのシステムが目的とする「コーディングスタイルの誤りを検知する」というタスクを考えると Precision の数値をあげることが何より重要であると考えられる．今回の実験では全体を通して Precision よりも Recall の方が大きくなってしまい，Precision の改善方法として閾値が重要なパラメータになると考えられる．以上より今後はより最適なウィンドウサイズと閾値のパラメータの組み合わせを探索する必要があると考えられる．

発表文献と研究活動

(1) 小林佑樹，千葉滋，再帰型ニューラルネットワークを用いたコーディングスタイルの自動検査手法の提案，日本ソフトウェア科学会第 34 回大会，2017.9.19-21 (発表申し込み中).

参考文献

[1] Cookpad styleguide. `https://github.com/cookpad/styleguide`, 2015.

[2] `https://github.com/bbatsov/rubocop`.

[3] Srinivas SS Kruthiventi, Kumar Ayush, and Radhakrishnan Venkatesh Babu. Deepfix: A fully convolutional neural network for predicting human eye fixations. *IEEE Transactions on Image Processing*, 2017.

[4] 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*, pp. 87–98. ACM, 2016.

[5] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, and et al. Learning to generate pseudo-code from source code using statistical machine translation. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference*, pp. 574–584. IEEE, 2015.

[6] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 5, pp. 306–315, September 2005.

[7] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, Vol. 12, No. 1, pp. 43–60, 2002.

[8] Moneyforward styleguide. `https://github.com/moneyforward/ruby-style-guide`, 2014.

[9] Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[10] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, Vol. 5, No. 2, pp. 157–166, 1994.

[11] Thomas Weinlein. Diplomarbeit im fach informatik.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Lstm can solve hard long time lag problems. In *Advances in neural information processing systems*, pp. 473–479, 1997.

[13] Kaisheng Yao, Trevor Cohn, Katerina Vylomova, Kevin Duh, and Chris Dyer. Depth-gated recurrent neural networks. *arXiv preprint*, 2015.

[14] Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. In *International Conference on Machine Learning*, pp. 1863–1871, 2014.

[15] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional lstm. In *CoNLL*, pp. 51–61, 2016.

[16] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354*, 2016.

[17] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[18] `https://github.com/redmine/redmine`.

[19] Cookpad github. `https://github.com/cookpad`.

[20] `http://pygments.org/docs/lexers/`.

[21] `https://chainer.org/`.

[22] `http://scikit-learn.org/stable/`.

謝辞

本研究を進めるにあたり，研究方針や論文構成，発表手法，研究アイディアなど多岐にわたって手厚くご指導賜りました指導教員の千葉滋教授に心より感謝致します．加えて，研究室同期であり，機械学習分野とソフトウェア工学の双方の視点から多くのアドバイスとサポートをくれた Daniel Perez 氏にも感謝致します．

日々本研究の議論にお付き合い頂き，様々な示唆を与えてくださった中山研究室の西田典紀氏に感謝の意を表します．西田氏は違う研究室に所属しているにも関わらず，本研究をすすめる上で多くのアドバイスを頂き，本研究の推進に大きくご協力いただきました．また，本研究で行った実験に協力してくださったクックパッド株式会社に感謝します．特に実験を進めるにあたりソースコードの提供や実験環境の準備など実験の協力に尽力してくださったクックパッド株式会社の庄司嘉織さん，笹田耕一さんに感謝致します．

最後に，日頃から研究室生活やミーティングを通じてご支援くださった千葉研究室の皆様に心より感謝致します．