東京大学
情報理工学系研究科 創造情報学専攻
修士論文

ソースコード内にコメントを挿入する際に適切なコメントを例示するシステムの開発

Development of a System that Exemplifies Appropriate Comments when Inserting Comments in Source Code

白石 誠
Shiraishi Makoto

指導教員　千葉 滋 教授

2022 年 1 月

概要

ソースコードに書くコメントの質を上げることはプロジェクトの保守性を維持するための重要な要素である．しかし, 従来のコメント支援の手法としてはルールを網羅的に明文化したり, レビューによる手動の添削をしたりすることが主に採用されており, 管理コストや作業コストがかかることが課題であった.

本研究では, コメントの書き方は周辺のソースコードの特徴に大きく依存すると仮定して, コメントを挿入したい周辺のソースコードの特徴から既存のコメントの中で最も適切なものを提示するシステムを機械学習を用いて実装した．学習データは Github からスター数の多い Ruby を主言語とするリポジトリを集めた．評価方法はシステムが提案したコメントと正解のコメントを形態素解析して主として BLEU Score 値を用いて測定し, システムの有用性を検証した.

Abstract

Improving the quality of the comments written in source code plays an important factor in maintaining the maintainability of a project. However, the main conventional method to maintain good quailty comments is to comprehensively clarify the rules or to make manual corrections by review. The problem of these method is that management costs and work costs are very high.

In this research, we hypothesized that the way you write comments largely depends on the characteristics of the surrounding source code. So we implemented a system that presents the most appropriate comments by reading the characteristics of the peripheral source code where you want to insert a comment in. We implemented this system by using machine learning. As for the learning data, we collected repositories whose main language is Ruby, and has many stars, from Github and evaluated the system. The usefulness of the system was verified by morphologically analyzing the comments proposed by the system and the correct comments and measuring them, mostly by using the BLEU Score value.

目次

第 1 章

はじめに

近年, プログラミングの義務教育化や社会的にプログラマのニーズが高まっていることから, 今後プログラマの人口が増えていくことが予想される. しかしプログラマの人口が増えることによって, ソフトウェアの開発を円滑に行うことはむしろ難しくなっていくことが予想される. これはソフトウェアの仕様や組織で定めたコーディング規約の統一と遵守の管理が難しくなったり, 他人が書いたソースコードの改良や修正及びコミュニケーションコストが増えたりすることによって招かれる. また Ruby を代表とする柔軟性の高い言語は同じ機能でも複数の書き方が存在したり, インデントなどを特に意識しなくても正常に実行されてしまったりするため, こういった書き方を許してしまうと可読性の低いソースコードが残ってしまう危険性がある. 大規模なソフトウェア開発においては, 特にコーディングスタイルを遵守させることは保守性の高いソースコードを維持する上では不可欠な要素である. そのため, これらを解決するためのコード支援の研究が盛んに行われている. 近年は GPU が安価に手に入ることや, 機械学習のライブラリを手軽に使えるようになったことによって従来では解くことができなかった様々な問題を機械学習のアプローチで取り組まれるようになってきている. ソフトウェア開発においても機械学習のアプローチが盛んになっており, 例えばソースコードの内容から疑似コードを生成する研究 [1] やコーディングスタイルのルールを学習して明文化する研究 [2], コーディング習慣を学習する研究 [3] などが挙げられる.

コード作成支援の取り組みの中でも, ソースコード中に書くコメントの質を上げることは重要な課題である. コメントはソースコードの中で唯一自然言語で書けるものであるため, わかりづらいソースコードやソースコード中では読み取れない抽象度の高い情報を書き加える際によく用いられる. これにより引き継ぎやメンテナンスをした時に, より早くより正確にソースコードを読み解くことができ, 保守性を維持する助けとなってくれる. しかし, コメントは実際のプログラムの実行に影響を及ぼさないため, 実装を重視する組織やタイトなスケジュールなどによって雑な書き方になってしまったり, 優先度が低くなったりする問題がある. これにより, よくないコメントがレビューを通過してしまい, バグの温床となるコメントがそのまま残ってしまう恐れがある.

そこで本研究ではソースコードの類似度が似ていれば, コメントの類似度も似ているという仮説を立て, コメントを挿入する周辺のソースコードの特徴から既存のコメントの中から最も

適切なコメントを算出し，具体例として提示するシステムを開発してその有用性を測った.

本手法ではソースコードの各トークンを Skip-gram によってベクトル表現化した後に, Encoder-Decoder モデルを構築して, そこに各トークンがベクトル化されたソースコード片を入力し, トークン列のベクトル表現を得た．このベクトル表現とそれに付随するコメントを一つのペアにして大きなコーパスを作成した. 未知のソースコードをシステムに入力すると, それをベクトル化してコーパスにある全てのソースコードのベクトル表現とのユークリッド距離を測り, 最も距離の近いベクトル表現に紐づいているコメントを適切なコメントとみなしてユーザに提案するようにした. 手法の具体的な内容は第 3 章で述べる.

本論文の残りの構成は第 2 章では、本研究の背景と必要となる前提知識について述べる．第 3 章では提案手法と必要な前処理及びトークン列のベクトル化手法について詳しく説明する．第 4 章では実験の概要と目的について述べた後に, 評価指標と用意したデータセットについて触れる．その後実験の実装及び結果と考察を行う．最後に第 5 章では本研究のまとめと第 4 章の結果を踏まえた今後の課題について述べる.

第 2 章

プログラミングの支援技術

ソフトウェア開発においてプログラマはソースコードの内容の理解に 59%[4] の時間を割いていると言われている．特に Ruby を代表とするような、多様なコーディングスタイルや柔軟な書き方を許容する言語では一つの機能に対して, 複数の実装の仕方が存在する．そのため大規模なソフトウェア開発においてコーディングスタイルをチーム内で統一し, 保守性を担保することは潜在的なバグや冗長な文を書くことを防ぐための大事な要素である．

コーディングスタイルの中でもソースコード中に書くコメントはプログラマにコードの内容を理解しやすくするために書くものである．そのためコメントの質をあげることはソースコードの内容を素早く正確に理解することにつながる．しかし，コメントは直接プログラムの動作には関係しないため，実装を重視する組織や納期が迫っているような場合では質の良いコメントを書くことは軽視されがちである．また，開発者が不適切なコメントを書きレビュアーにレビューしてもらうとき，コメントに対する無駄なやりとりが増えたり，不適切なコメントがそのままソースコードに反映されてしまったりする問題もある．

本章では, 上記で述べたコーディングスタイルを守らせるためのコード作成支援に対する一般的な取り組みについて説明した後に, その中でもソースコード中に書くコメントに焦点を当てて, 悪いコメントの事例やそれらを防ぐための既存手法と関連研究, さらにその問題点について説明する．

2.1 コード作成支援

本論文では、「コード作成支援」のことを統一的で保守性, 可読性の高いソースコードを書くための手法全般のことを指す．コード作成支援には大きく分けて 2 種類がある．

(a) 手動による検査
(b) ルールベースによる静的な検査

(a) とは, コーディングスタイルのルールをドキュメントや紙面に残して, 組織内のプログラマに周知させたり, 人の書いたソースコードを他の人がレビューしたり, 後で自分で見返してリファクタリングしたりするようなことである．(b) の代表的なものとして, 静的解析ツールが

ある．組織内で採用されているルールやコーディングスタイルを静的解析ツールの設定ファイルに書き込み, 自動で検査してくれるものである．ここではそれぞれの検査について紹介していく．

2.1.1　手動による検査

手動による検査の代表的なものとして, コーディングルールを明示的に記載したドキュメントの作成やコードレビューなどがある．ここではそれぞれについてより詳細に説明していく．

まずコーディングルールを明示的に記載したドキュメントの作成について説明する．組織が採用しているコーディングスタイルをドキュメントに残し, チーム内で共有することは, 多くの企業が採用しているコード作成支援の手法の一つである．実際 Ruby をメインの開発言語として扱っているクックパッド株式会社や Airbnb Japan 株式会社はコーディングガイドを記載したドキュメントを公開 [5][6] しており, チーム内で共有している．

Airbnb Japan 株式会社は Ruby 以外にも,React, CSS-in-JavaScript, CSS & Sass などのコーディングガイドも公開している．Airbnb Japan 株式会社のコーディングガイドは項目で分かれており, 必要であれば良い例と悪い例が例示されている．例えばメソッドの項目に関して, メソッドに書く引数については図 2.1 のように位置引数は非推奨で, キーワード引数が推奨されている．このような項目が 16 種類あり, それぞれの項目で例を交えながら説明している．

次にコードレビューについて説明する．コードレビューとは, 書き下ろされたばかりのソースコードに潜在的なバグや命名規則の違反などがないかを本人または他者が体系的に添削する操作のことをいう．人間の書いたソースコードには多くの場合セキュリティホールやバグの温床となるもの, 誤記や組織内のコーディング規約違反のものが意図せず書かれてしまうことが多々ある．こういったソースコードを訂正するために, 透明性が担保された場でソースコードを本人または他者に添削して査読してもらうことがある．透明性が担保された場でコードレビューすることにはいくつかの利点がある．

(a) よくあるバグやコーディング規約違反を組織内で共有することができ, コーディング規約の共通認識を作ることができる．
(b) ソースコードの品質を一定に保つことができる
(c) 添削の過程でプログラマに新たな知識を与えるなど, 育成の一助となる
(d) ソースコードの可読性の向上が期待される

コードレビューをオープンな場でできる Web サービスに Github[*1]がある．Github とは、開発プラットホームの一つでプログラムのソースコードを公開したり, プログラムの管理をしたり, コードレビューをしたりすることができる．実際公開用のプログラムで行われたコードレビューの例を図 2.2[7] で示す．

これは Ruby や Ruby on Rails のテストツールである RSpec[*2]を使ってテストコードを記

[*1] https://github.com/
[*2] https://rspec.info/

```
1  # bad
2  def obliterate(things, gently = true, except = [], at = Time.
       now)
3  ...
4  end
5
6  # good
7  def obliterate(things, gently: true, except: [], at: Time.now
       )
8  ...
9  end
10
11 # good
12 def obliterate(things, options = {})
13 options = {
14 :gently => true, # obliterate with soft-delete
15 :except => [], # skip obliterating these things
16 :at => Time.now, # don't obliterate them until later
17 }.merge(options)
18
19 ...
20 end
```

図 2.1. コーディング規約の例

述したソースコードである．背景が赤になっている部分がプログラマが書いたコードである．この書き方は実装上は問題ないが, モジュールが他のテストシナリオにもリークしまう恐れがあり, レビュアーはそれを懸念して, より保守性が保たれたソースコードに書き直すようにアドバイスしている．このように, コードレビューは第三者の目によるソースコードの査読によってセキュリティホールや誤記が本番環境にあげられないような役割を果たしており、多くの企業が採用している手法である．

2.1.2 ルールベースによる静的な検査

本節では Ruby のコーディングチェックツールである Rubocop[*3]を例にルールベースによる静的な検査について説明する．ソースコードを静的に解析するツールは一般的に, 指定され

[*3] https://docs.rubocop.org/rubocop/index.html

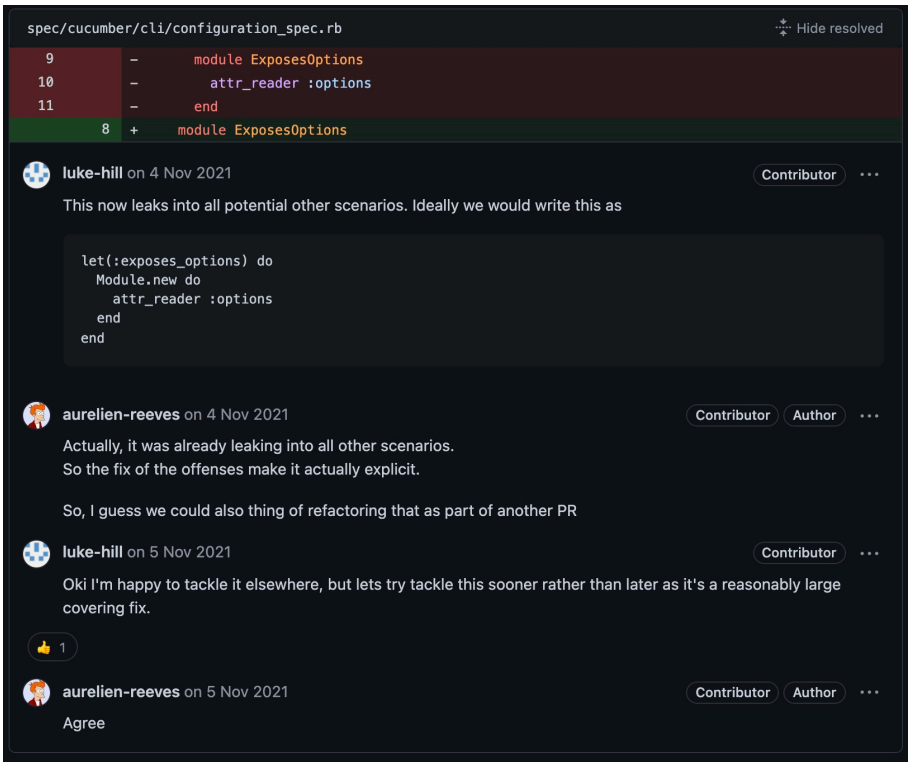

図 2.2. コードレビューの例

た設定ファイルにツールの仕様に沿ってルールを記述していき, プログラマがその解析ツールを駆動したときにルール違反がないかをプロジェクト内で捜査する. もし違反があった場合には, その違反をプログラマに通知するか, もしくは可能であれば自動修正をする. また違反にはその度合いによって, Warning, Error, Fatal などと段階が分かれており, 違反が些細なものであれば見過ごすかもしくは修正するかをプログラマに委ねることもある.

Ruby のコーディングチェックツールに Rubocop が存在するが, 他の言語にも類似するツールや IDE 内の拡張機能として備わっているものもある. Rubocop は Ruby の柔軟な言語に対応するために YAML 形式の設定ファイルにルールを記述することでカスタマイズを可能としている. デフォルトの状態だとコーディング規約が厳しい上に依存関係のある他のライブラリも捜査対象になってしまうため, ユーザは設定ファイルからどのファイルを対象として捜査するかやどのルールを適用するかを決める.

図 2.3 には Rubocop の設定ファイルの記述例を示す. この例では 3 から 6 行目に Rubocop の捜査対象外のファイル群を指定している. デフォルトの状態だと, ライブラリの依存関係をまとめたフォルダも捜査対象となってしまう. ライブラリの依存関係をまとめたフォルダはほとんどの場合外部からのソースファイルで構成されており, 量も膨大であるため冗長な検査となってしまう. そのようなファイルは捜査対象の範囲に入れないように設定して捜査及び修正コストを下げる. また 7 から 8 行目では Rails 向けの cop を実行している. 10 から 12 行目は括弧や中括弧に空白を入れるかどうかを設定している. EnforcedStyle を no_space にするこ

```
1  AllCops:
2    Exclude:
3      - 'bin/*'
4      - 'lib/*'
5      - 'node_modules/**/*'
6      - 'test/**'
7    Rails:
8      enabled: true
9
10   SpaceInsideHashLiteralBraces:
11     EnforcedStyle: no_space
12     EnforcedStyleForEmptyBraces: space
13
14   LineLength:
15     Max: 80
```

図 2.3. Rubocop の設定ファイルの記述例

とによって半角スペースが不要になり, EnforcedStyleForEmptyBraces を space にすることによって空ブレースに半角スペースが必要となる. 最後に 14 から 15 行目は一行に書ける文字量の上限についてである. LineLength の Max を 80 に記述することによって一行に書くことができる文字量の上限を 80 に設定している. このように Rubocop は数多くの設定が用意されており, ユーザが自由にカスタマイズできるようになっている. この設定で Rubocop を実行すると対象ファイル内を捜査し, 違反がログに記載されてユーザに知らされる.

2.2 コメントの支援

コード作成支援の中でも, ソースコード中に書くコメントの質を上げることは重要な課題である. 良いコメントは人間が理解しやすい自然言語で書かれるため, ソースコードの内容を素早く理解する大きな役割を担っている [9].

しかし、コメントは実際のプログラムの実行に影響を及ぼさないため, 時間にゆとりがない時などはプログラマもレビュアーもコメントの優先度が低くなり, よくないコメントがそのままレビューを通過し, バグの温床となるような書き方がそのまま残ってしまう恐れがある. 本節ではよくないコメントの具体例を示した後にそのようなコメントを書かないようにするための既存の手法について説明する.

```
1  // set product to "base"
2  product = base;
3
4  // loop from 2 to "num"
5  for (int = 2; i<=num; i++) {
6    // multiply "base" by "product"
7    product = product * base;
8  }
```

図 2.4. コードに書かれていることの繰り返しの具体例

```
1  def postal_code
2    primary_address.present? ? primary_address.postal_code :
       nil
3  end
4
5  # def default_client_locale_to_use
6  #   if supported_locales.count == 1
7  #     locale = supported_locales.first.split("-")[0]
8  #   else
9  #     locale = default_client_locale || :en
10 #   end
11 #   locale
12 # end
```

図 2.5. ソースコード自体をコメントアウトしているの具体例

2.2.1　よくないコメントの具体例

よくないコメントの具体例 [8] として以下のようなものがある.

(a) ソースコードに書かれていることの繰り返し.
(b) ソースコード自体をコメントアウトしている.
(c) 具体的に書かれていない.
(d) 自己満足なコメント.

それぞれの具体例について説明する.

```
1  // BAD
2
3  /// <summary>
4  /// Determines whether the given file name fits one of
       multiple masks.
5  /// </summary>
6  /// <param name="fileName">Name of the file</param>
7  /// <returns>Boolean</return>
8  public bool FitsOneOfMultipleMasks(string fileName)
9  { . . .}
10
11 // GOOD
12
13 /// <summary>
14 /// Determines whether the given file name matches
15 /// one or more of the specified masks.
16 /// </summary>
17 ///
18 /// <param name="fileName">Name of the file.
19 /// Multiple masks may be included (in the <see cref="Mask"/>
       property)
20 /// by separating them with lines, commas, spaces, or
       vertical bars.</param>
21 ///
22 /// <returns>Boolean indicating whether the file name matches
23 /// one or more masks.</returns>
24 public bool FitsOneOfMultipleMasks(String fileName)
25 { . . .}
```

図 2.6. 具体的に書かれていないの具体例

```
1
2  // BAD
3
4  rescue StandardError => e
5    # I hate catching StandardError. Blech!
6    Rails.logger.error do
7      "ERROR::␣FAILED␣TO␣PURGE␣#{key}␣,␣#{e.class}:␣#{e}"
8    end
9  end
10
11 // GOOD
12
13 rescue StandardError => e
14   # I hate catching StandardError, ideally we'd be more
       specific.
15   # However, there doesn't seem to be a safe way to identify
16   # all network-based exceptions. See:
17   # https://stackoverflow.com/questions/5370697/
18   # what-s-the-best-way-to-handle-exceptions-from-nethttp
19   # For example, this does NOT WORK:
20   # rescue HTTParty::Error, Net::OpenTimeout, IOError => e
21   Rails.logger.error do
22     "ERROR::␣FAILED␣TO␣PURGE␣#{key}␣,␣#{e.class}:␣#{e}"
23   end
24 end
```

図 2.7. 自己満足なコメントの具体例

図 2.4 は「ソースコードに書かれていることの繰り返し」の具体例 [10] である．これは各行を逐一コメントで説明しており, その内容もただソースコードを翻訳しているだけである．複雑なロジックが組まれている場合はその必要性も理解されるが, 図のようなとても単純なソースコードにまでコメントを書くとかえって煩雑なものとなってしまう。こういったコメントは不要であるため削除するか, どうしても必要である場合は「なぜ」そのようなソースコードを書いたかを説明するべきであると考えられる．

次に, 図 2.5 は「ソースコード自体をコメントアウトしている」の具体例 [11] である．これは試験的に動作を試してみたり, バグを見つけるためにデバッグ用のコードを書いてみたり, ディレクトリ構造が大きく改変されたために一時的にコメントアウトしているようなときに多

く見られる．しかし, こういったソースコードは冗長であり, 全体を見づらくしてしまう．また「いつかのために残しておく」といった心理で残してしまう場合もあるが,Github などのプロジェクト管理ツールでソースコードの履歴を残すことができるため, 余程の理由がない限りはソースコード上には残さない方が良いと考えられる．どうしても残す必要がある場合には, 何のためにソースコードをコメントアウトしているかを明記する必要があると考えられる．

次に, 図 2.6 は「具体的に書かれていない」の具体例 [8] である．図ではドキュメントコメントを例示している.BAD のコメントは必要最低限の要約, パラメータ, 返り値しか書かれていない．こういった情報は利用者に付加価値を与えず, どのように API を利用するかが明確ではない．そのため, この場合はより具体的にコメントを書くことが必要だと考えられる．GOOD に書かれているコメントは BAD を修正したものである. 例えば返り値については Boolean だけに留まらず, より具体的にどのような値が返ってくるかがわかるように書かれている．このようにコメントは読み手がどのような層が多いのかを考えながら書くことが大事である．

最後に, 図 2.7 は「自己満足なコメント」の具体例 [12] である．これはユーザが思うような実装ができなかったり, 原因不明の挙動が起きてしまったときに, ストレス発散の場として書いてしまうようなコメントである．ユーモアはあるが, ソースコードはストレス発散の場ではない．うまく実装できなかった場合は代替案として記載したコードをなぜ書いたか, なぜうまくいかなかったかなどを明記したほうが後の開発プロセスでスムーズに移行できるようになる．そのため, 図の GOOD のように理想状態やユーザが試みたことなどを明記した上で実施した対策を記載した方が開発を引き継いだ人が読んだときに親切であると考えられる．

2.2.2 既存のコメント支援の手法

前節の 2.2.1 で述べたようなコメントを書かない、もしくは書いたとしても訂正できるようにする手法として以下のような手法が考えられる．

(a) コメントのコーディング規約の遵守
(b) 手動による添削
(c) 静的解析ツールや IDE の拡張機能の利用
(d) 機械学習などを用いた統計ベースなアプローチ

(a) は各言語ごとのコーディング規約をチーム内で統一させ, 従うことを指す．例えば「Eclipse で学ぶ初めての Java」[13] では良いコメントの条件としてコードの意図を説明したもの, 書籍の見出しや目次のようにコードの要約になっているもの, 著作権の告知やバージョン情報といった, コードでは表していない情報などをあげている．

(b) は 2.1.1 で述べたように, コードレビューを代表とする人間の手による添削のことを指す．組織内で定められているコメントのフォーマットに違反があったり, 誤字脱字があった場合などにレビュアーがコメントの訂正をプログラマに指摘する．

(c) は 2.1.2 で述べたような, プログラミング言語のコーディングチェックツールの利用や IDE に備わっているコメントの補助をする拡張機能のことを指す．例えば IDE の拡張機能と

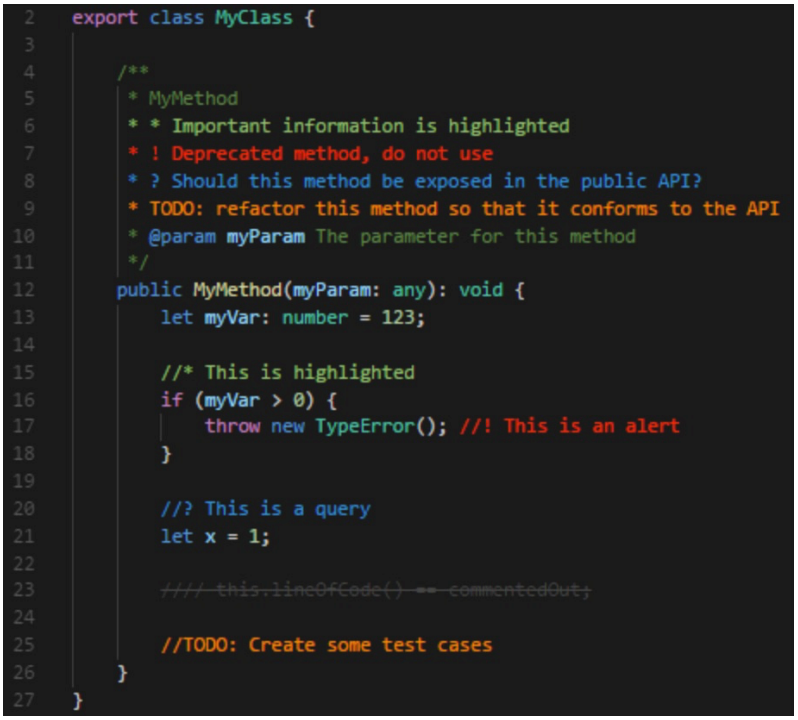

図 2.8. commentBetter の例

して VSCode[*4]の Better Comments[*5]がある．図 2.8 のように Better Comments はコメントを用途によってカテゴリ化して, 色別でユーザに識別させる．このカテゴリの種類はユーザがカスタマイズすることもできる．また Better Comments は多くのプログラミング言語をサポートしている．Better Comments はコメントの内容自体を補助するものではないが, コメントを書く側に対して書く目的や意図を意識づけさせたり, 読む側にも事前知識を与えてくれるといったメリットがある．

(d) について近年 Keras[*6]や Pytorch[*7]など機械学習に特化したライブラリの登場や GPU を安価に手に入れることができるようになったことによりソフトウェア工学において機械学習を用いた研究が盛んに行われている．そのため, 今までルールベースが中心で行われてきた研究も統計ベースの手法を適用することにより, より良い精度を出すことができるようになっている．ここではソースコード中に記載するコメント支援に関する研究をいくつか紹介する．

Annie *et al.*(2020)[14] らはコメントを書くときの適切な場所を提示するシステムを提案した．複数行のソースコードを一つの塊としてニューラルネットワークに入力して, それぞれの行にコメントを入れることが適切かどうかを 2 値分類で学習させる．モデルは RNN をベースに実装されており, 各行が時系列順に入力され評価される．ポイントは, ある行を評価すると

[*4] https://code.visualstudio.com/
[*5] https://marketplace.visualstudio.com/items?itemName=aaron-bond.better-comments
[*6] https://keras.io/
[*7] https://pytorch.org/

きに, その行だけではなく前の複数行も判断材料として加味していることである. 結果として Precision 値が 74%,Recall 値が 13% であった.

Xing *et al.*(2018)[15] らは,Java プログラムのメソッドに限定してコメントを自動生成する「DeepCom」を提案した. これはコメントの自動生成タスクをソースコードからコメントに翻訳しているとみなして, 機械翻訳タスクによく用いられる Seq2Seq モデルを適用した研究である. ソースコードから JavaDoc にコメントが記載されているメソッドを抽出して, 学習を行った. 実験では機械翻訳タスクで利用される BLEU-4 値を使って定量的に評価された. BLEU-4 値による評価では平均で 38.17% という結果になった. また個別の結果としては, Java のメソッドが単純で一般的なものに対しては比較的良い結果が得られ, 独自性の高いソースコードや複雑なソースコードに対してはあまり良い結果が得られなかった. またコメントの自動生成は壊れた文章を出力したり, ソースコードを逐一翻訳してしまったりすることも多く課題が残っている.

2.3 既存のコメント支援の問題点

2.2.2 で示したようにコメント支援にはいくつかの手法が存在する. ここではそれぞれの手法の問題点を述べる.

「コメントのコーディング規約の遵守」については教育コストがかかってしまい人手が足りてない組織ではコメントの規約を遵守させるような仕組み化が未整備な場合がある. またコメントの規約は全ての場合について明文化することは現実的に難しく, 一般論に留めているものも多い. そのため具体的なコメントの書き方は現場のプログラマに委任されてしまい, 同じ組織でありながらスタイルが統一されないといった問題がある. また規約を管理する運用コストも高く, ドキュメント作成と保守に人員が割かれてしまう.

「手動による添削」についてはレビューできる人材が少ない場合, 添削コストの負担は少ないメンバーに集中してしまう. また, レビュー依頼を出してもすぐにレビューが返ってくることは少なく通常は 1,2 日後に返ってくることが多い. そのため待ってる間は開発の進捗が滞ってしまう. またヒューマンエラーにより規約違反があったとしても見過ごされてしまう可能性もあり, 完璧な対応とは言えない.

「静的解析ツールや IDE の拡張機能の利用」については運用コストが高いことが挙げられる. 手動で全てのルールを細く設定する必要があり, 規約の変更や言語仕様そのものが変わってしまった場合は, 設定ファイルを書き直す必要が出てくる. また全てのルールを明文化する必要があるため, そもそもツールに該当する設定がない場合は直接カスタマイズする必要が出てきたり, 明文化することが難しいスタイルは適用できなかったりする問題もある.

「機械学習などを用いた統計ベースなアプローチ」は上記の手法の問題点を解決するために近年盛んに行われるようになっている. しかし, 既存の研究ではソースコードの翻訳や簡単な自動生成しかできておらず, まだ発展段階にある. またコメントを自動生成することによって完全に補完することを目的とした研究が多いが, ほとんどはソースコードの翻訳にとどまってしまうものやそもそも文として成り立たないものを出力してしまうことが多い. コメントは

ソースコードの翻訳を書くのではなく, 例えば「なぜそのようなソースコードを書いたのか」といった, ソースコードだけでは読み取れない抽象度の高い内容を書くことが求められる.

そこで本研究では, 既存のコメントの中にも質の高いコメントがたくさんあることに着目して, それらのコメントを提示することによってコメント記入の補助をする手法を提案する. これは今までの研究のようなコメントの自動生成とは違い, すでに存在するコメントの中から参考になるものを選んで提示することによりコメント記入の手助けをすることを目的としている. またこの手法は壊れた文を出力することもない. これにより, まだプログラミング歴の浅いプログラマにとってはコメント記入の助けになるだけではなく, ベテランプログラマにとっても新たな知見を発見する動機付けになることが期待される. 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, 既存のコメントの中から類似するコメントを提案する研究はなされていない.

2.4　機械学習の前提知識

前節で, 機械学習などの統計ベースによるアプローチがコメント支援手法として, 近年注目されていることを述べた. 本研究でも, 既存のコメントの中から適切なコメントを提案するモデルを機械学習を用いて実装する. 本節では, コメント支援の研究において多く利用されている機械学習の基礎知識を具体的に説明する. 初めに機械学習において基礎となる Neural Network の仕組みについて説明する. 次に Neural Network の考え方を応用して単語の埋め込みベクトル表現が得られる Skip-gram モデルについて説明する. 次に Neural Network を拡張して時系列データを扱えるようにした RNN(Recurrent Neural Network) とその発展系である LSTM(Long Short Term Memory) について説明する. 最後に翻訳タスクなどによく用いられ,LSTM から構成される Encoder-Decoder モデルについて詳しく述べる.

2.4.1　Neural Network

図 2.9 に最も単純なニューラルネットワークを示す. ニューラルネットワークは入力層, 中間層 (隠れ層), 出力層に分けられる. 各層には重みが定義されており, それぞれのユニットは活性化関数を持っている. またニューロン全体に対するバイアス b を持っている.

各ユニットを図 2.10 に示す. ユニットは入力値とそれに対応する重みをかけ合わせ, その和を取ることによって計算される. 一般的な数式で表すと (2.1) のようになる.

$$\sum_{k=1}^{n} w_i x_i + b \tag{2.1}$$

この計算結果を活性化関数に入力して得られた値を次の層に出力する. ニューラルネットワークによく用いられる活性化関数の一つにシグモイド関数がある. これは以下の数式で表される関数である.

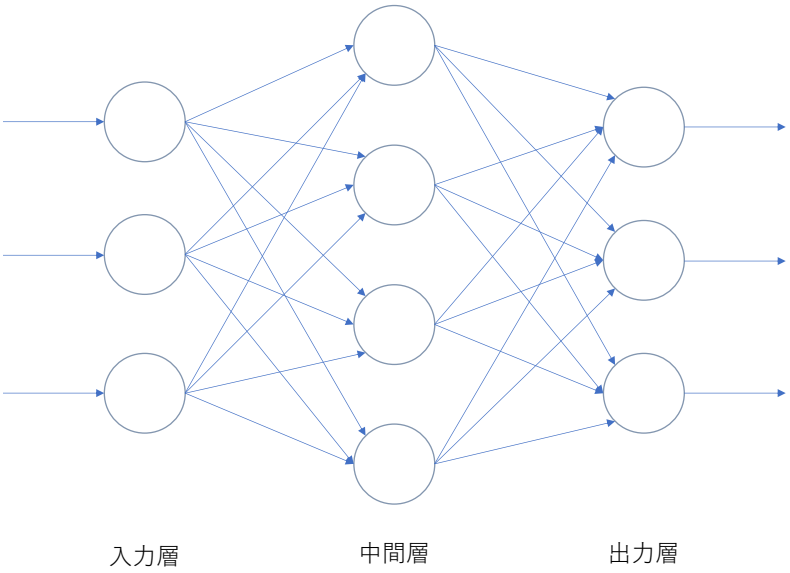

図 2.9. ニューラルネットワークの構造

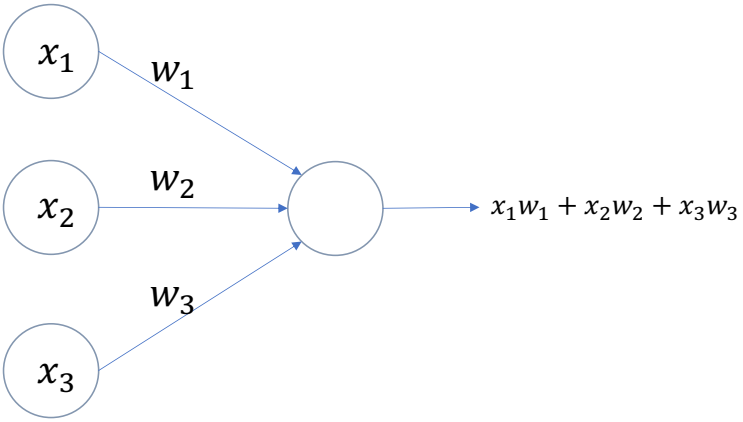

図 2.10. 各ユニットの計算

$$h(x) = \frac{1}{1 + exp(-x)} \tag{2.2}$$

x には式 (2.1) が入る. 活性化関数には他にも種類があり, 例えばステップ関数や Relu 関数などが挙げられる. それぞれ式 (2.3),(2.4) で表される.

$$h(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases} \tag{2.3}$$

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases} \tag{2.4}$$

図 2.11. Skip-gram モデルが扱う問題

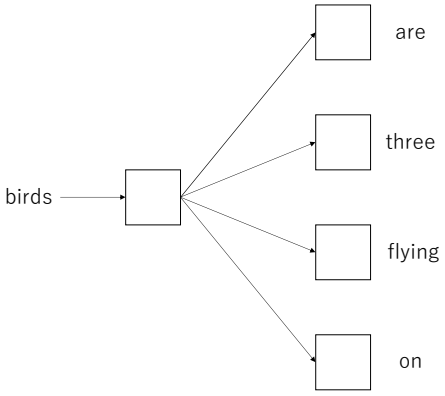

図 2.12. Skip-gram モデルの概観図

ニューラルネットワークは入力層から入力値を得て, 各ユニットで計算を行い, 出力層で最終的な出力値を得る．学習時は出力層で得られた値と実際に期待される答えとなる値の距離が最小化するように最適化される．最適化するための手法は確立勾配降下法が用いられることが多く, これにより各層の重みとバイアスの値が更新されていく．

2.4.2　Skip-gram モデル

単語をベクトル表現で表すことはソースコードの特徴を表す上では重要なタスクである．単語をベクトル表現に表す代表的な手法として CBOW モデル [16] と Skip-gram モデル [16] がある．本節ではより優れた結果が得られる傾向 [17] にある Skip-gram モデルについて説明する．

Skip-gram モデルは「ある単語は周辺の単語によって意味づけされる」という仮説に基づいて計算される推論ベースのモデルである．実際にモデルが学習することは「ある単語の周辺に出現する単語の確率」である．図 2.11 に Skip-gram モデルが扱う問題について示す．これは”birds”という単語に着目して, 周囲にある単語を予測させているときのものである．周囲の単語のことをコンテキストと呼ぶ．モデルの概観図を図 2.12 に示す．学習する時は, 注目する単語を入力層に入力して, 出力層はコンテキストの数だけ用意する．それぞれの出力層には損失関数が定義され, 損失関数の平均値を最小化するように重みを更新していく．この操作を注目する単語を一つずつずらしながら繰り返していく．最終的にニューラルネットワークで用いた重みの各行が単語のベクトル表現として得られる．

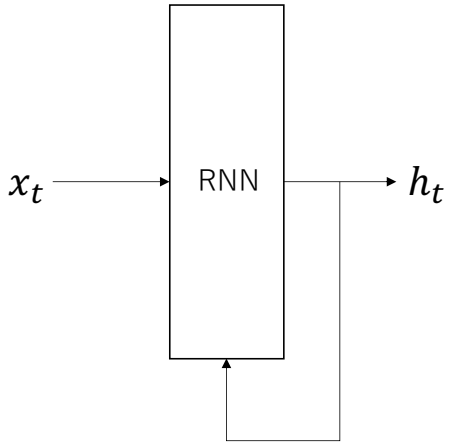

図 2.13. RNN レイヤ

2.4.3 RNN と LSTM

前節で説明した内容は全て単一方向のネットワークである．つまり, 入力値が与えられるとその情報は次の層に渡され, その次の層に渡され・・・, と一つの方向だけの伝達となる．この問題点として自然言語などの時系列データを上手く扱えないというものがある．RNN(Recurrent Neural Network) はこの問題を解決するために考えられたニューラルネットワークである．本節では RNN の構造と仕組みを説明した後に RNN の問題点を指摘して,RNN をさらに応用させた LSTM(Long Short Time Memory) モデルについて説明する．

RNN レイヤを図 2.13 で示す．図からわかるように RNN レイヤは閉じた経路を持ち, これにより前の情報が次のレイヤに引き渡され, 過去の情報を保持することができるようになる．レイヤの内部ではレイヤへの入力 x_t と前の情報を保持している h_{t-1} を受け取り h_t を出力する．この時 h_t を算出するときの計算は式 (2.5) である．

$$h_t = tanh(h_{t-1}W_h + x_tW_x + b) \tag{2.5}$$

ここで W_x は入力 x を h に出力するための重みであり, W_h は一つ前の出力を次時刻の出力に変換するための重みである．RNN の重みの更新は誤差逆伝播法という手法を用いる．時系列を考慮したニューラルネットワークの誤差逆伝播法のことを BPTT(Backpropagation Through Time) と呼ぶ．しかし BPTT は各時刻の中間データをメモリに保存する必要があるため, 入力する時系列データが長くなると, 消費するコンピュータリソースが膨大となってしまう．これにより, 逆伝播時の勾配が不安定になる．この問題を解決するための手法として Truncated BPTT がある．これは時系列データを適当なサイズに切って消費データを抑えようとする手法である．

このようにして様々な工夫がなされている RNN だが, 時系列データが長いと上手く学習できないという問題がある．これは RNN は直近のデータ情報の影響を強く受け, 離れている

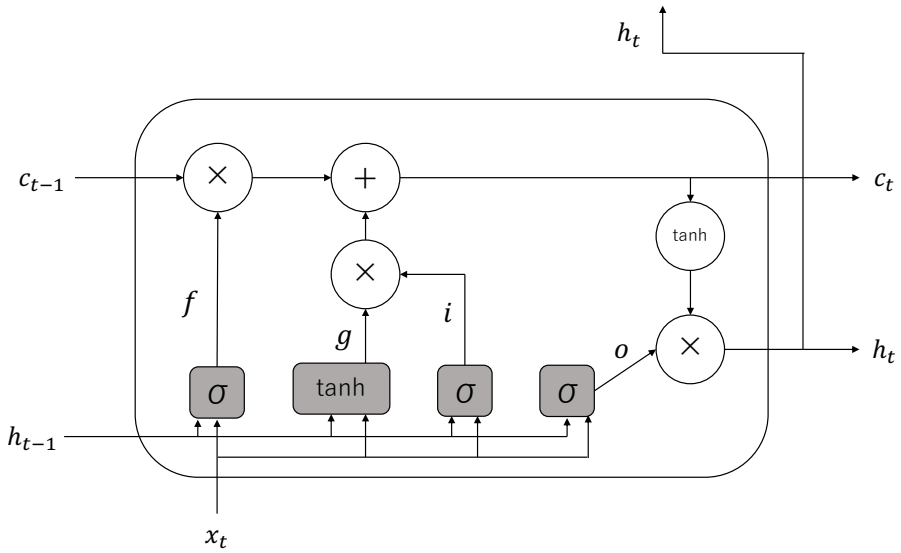

図 2.14. LSTM モデルの構造

データ情報の影響はほとんど受けなくなるという特徴があるからである. 長い時系列データを入力すると, 重みの更新の過程で勾配が弱くなってしまい, 重みパラメータは更新されなくなってしまう. 時間を遡るにつれて勾配が小さくなる勾配消失か大きくなる勾配爆発のどちらかが起こってしまう. これらを解決するために RNN をさらに応用した LSTM(Long short-term memory)[18] が使われる.

図 2.14 に LSTM モデルの概略図を示す.

RNN は単一の tanh 層だけがあったのに対して,LSTM 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持っており, より複雑になっている. ここでは LSTM の仕組みについて説明する. RNN との違いは LSTM には c という経路があることである. これは記憶セルと呼ばれ,LSTM 内部だけで保持する記憶である. 記憶セルでは新たな情報を更新したり, 不要となった情報を削除したりしている. この操作は LSTM レイヤにあるゲートと呼ばれる構造によって制御している. 図 2.14 の f ,i ,o がゲートの役割を担っており, それぞれシグモイド関数に通すことによって演算している. 例えば f は式 (2.6) の演算を行なっている.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.6}$$

i,o についても同様の演算を行なっている. またそれぞれのゲートの役割は以下のようになっている.

- forget ゲート (f): 過去の情報の中で不要なものがどれだけあるかを計算する.
- input ゲート (i): 新たに加えられた情報がどれだけ重要かを計算する.
- output ゲート (o): 次時刻の隠れ状態としてどれだけ重要かを計算する.

また図 2.14 にある g は新しく追加する情報を取捨選択して記憶セルに追加するための演算

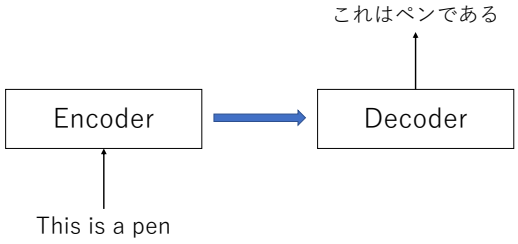

図 2.15. Encoder-Decoder モデルの例

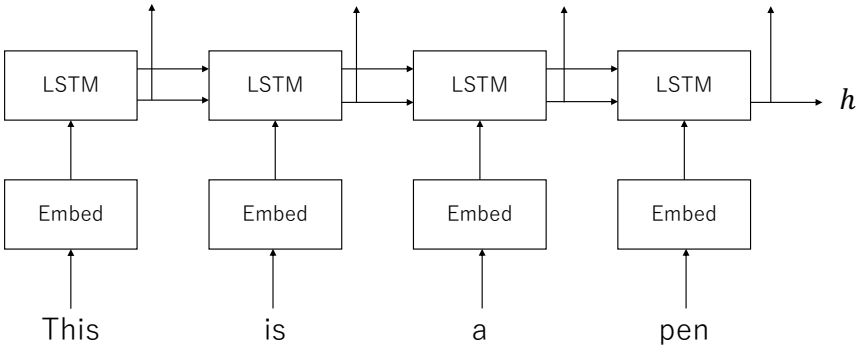

図 2.16. Encoder の概略図

である．これらのゲートがあることにより, 効率よく重要な情報を保持して, 不要な情報を削除している．それによって RNN の問題点であった, 勾配消失を起こさずに値の更新を行うことが可能となる．近年は LSTM をさらに改良したモデル [19][20] が考案されており, 盛んに研究が行われている．

2.4.4 Encoder-Decoder モデル

Encoder-Decoder モデルは時系列データを別の時系列データに変換させたい場合に用いられるモデルであり seq2seq とも呼ばれる．モデルは Encoder と Decoder によって構成され, Encoder によって与えられた時系列データが埋め込みベクトルに変換される．その埋め込みベクトルを Decoder に渡し, 目的となるデータに変換して出力する．このモデルを使用する代表例として翻訳タスクがある．図 2.15 はモデルを通して翻訳を行なっている例である．

Encoder-Decoder の中身は前節で説明した LSTM によって構成されることが多い．まず Encoder について説明する．図 2.16 は Encoder の中身を展開して示した図である．入力する文字列は Skip-gram モデルなどを用いてベクトル化 (図の Embed) した後に, LSTM モデルに入力される．計算を時系列順に行い, 最後に得られた隠れ状態 h を時系列データ全体を表すベクトル表現と見なす．隠れ状態 h は Decoder の入力値として渡される．また h は固定長ベクトルである．

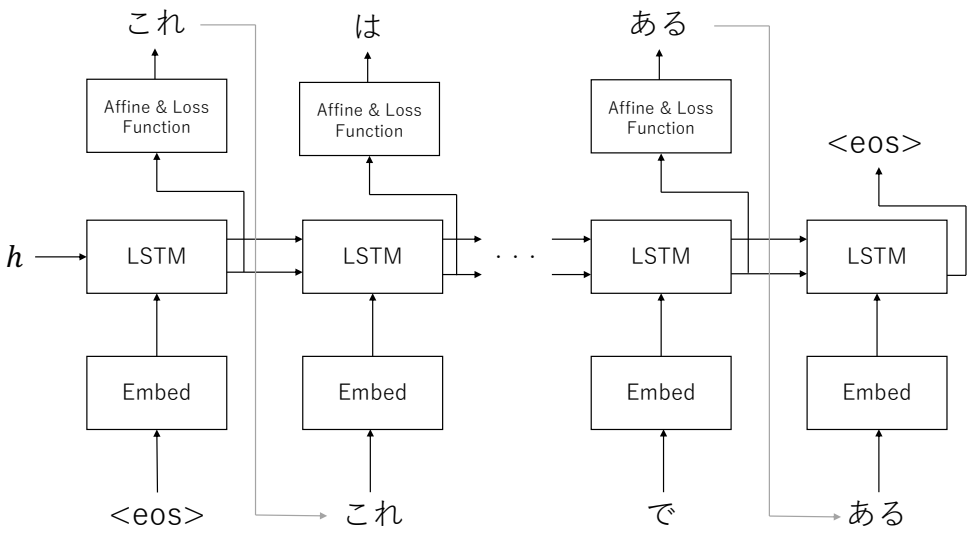

図 2.17. Decoder の概略図

次に Decoder について説明する. 図 2.17 は Decoder の中身を展開した図である．図中の eos は区切り文字であり, 文章生成の開始と終了を知らせるものである．Encoder から受け取った h と開始タグ eos を入力値として LSTM レイヤーに入れる．得られた隠れ状態と Affine 層の重みで重み付き信号の総和を計算して, 推論された単語を次の LSTM の入力値とする. これを終了タグ eos が出力されるまで繰り返す．これによって時系列順に得られたデータが出力値となる．またモデルの学習時では答えとなる時系列データが事前にわかっているため, 推論された単語を次時刻の入力値とはせず, 答えのデータが入力される．このように Encoder-Decoder モデルは二つの LSTM を繋ぎ合わせることによって構成される．現在はこのモデルをさらに改良するための研究 [21] も盛んであり, タスクによっては強力な力を発揮しているものもある.

第 3 章

コメントの探索

本論文では, 第 2 章で説明したコメントに対するよりよい支援手法として、既存のコメントの中から最も適切なコメントを提示するシステムを提案する．この手法は, 似たような構造を持つソースコードに記載するコメントも似たような構造になるという仮説に基づいている．ここでいう似たようなコメントとは、スタイルやカテゴリが一致するようなコメントのことを指す．スタイルとは例えばコメントの最後に署名を書くフォーマット,ToDo の宣言のフォーマットやコメントの文字量などが一致していることである．カテゴリとは例えばコメントの内容が処理の説明や, 入出力の条件についてのもの, 不変条件についての記載などそのコメントの目的が一致していることである．本手法は周辺のソースコードからスタイルやカテゴリが似ているコメントを提示することを目指す．このような仮説を立てた根拠としては, コメントの内容は主にソースコードの構造や特徴に大きく依存していると考えられるからである．例えば図 3.1 にはソースコードの特徴が似ていて, かつコメントのスタイルとカテゴリが似ている例を示す [22]．このようにコメントはソースコードの内容によって特徴づけられることがわかるため, 上記の仮説に基づいてシステムを開発した．本手法により, プログラマがソースコード中にコメントを記入する際に, 過去の良質なコメントを参考にしながら書くことができるようになる．それにより特にプログラミング歴が浅いプログラマによる低品質なコメントが減り, レビューの工数が減ることなどが期待される．また本研究は Ruby を主言語とするプロジェクトを対象としており,Ruby 全体で暗黙的に了承されているコメントの書き方やプロジェクト間のコメントスタイルの違いについて新たな知見を得ることも期待される．本章では本手法の概要と実験の前処理及びモデルの概要について説明する．

3.1 概要

本手法ではソースコードのベクトル表現とコメントがペアになっている大きなコーパスを用意する必要がある．ここではまずソースコードのベクトル表現の取得方法についての概要を説明した後に未知のソースコードに対して適切なコメントをコーパスから抽出する方法について説明する．

まずはソースコードのベクトル表現の取得方法について説明する．コメントのペアとなって

```
1   ##
2   # Look up the coordinates of the given street or IP address
        .
3   #
4   def self.coordinates(address, options = {})
5     if (results = search(address, options)).size > 0
6       results.first.coordinates
7     end
8   end
9
10  ##
11  # Look up the address of the given coordinates ([lat,lon])
12  # or IP address (string).
13  #
14  def self.address(query, options = {})
15    if (results = search(query, options)).size > 0
16      results.first.address
17    end
18  end
```

図 3.1. ソースコードとコメントの特徴が似ている例

いるソースコードに前処理を行いトークン列を得る. 得られたトークン列の各トークンをベクトル表現に変換して, そのベクトル表現を用いてニューラルネットワークからトークン列のベクトル表現を得る. このトークン列のベクトル表現をソースコードのベクトル表現とみなして, コメントのペアとさせる. ニューラルネットワークには RNN を応用した LSTM を用いて Encoder-Decoder モデルを構築した.

次に未知のソースコードから最も類似するコメントを取得する方法について説明する. 未知のソースコードについても前処理を行いトークン列を得る. 図 3.2 のようにそのトークン列を学習済みの Encoder-Decoder モデルに入力してベクトル表現を得た後に, コーパスにある全てのベクトル表現の中で最も距離が近いものを算出する. 距離の計算にはユークリッド距離を用いる. 距離が最も近かったベクトル表現のペアとなるコメントが最終的な値として出力される.

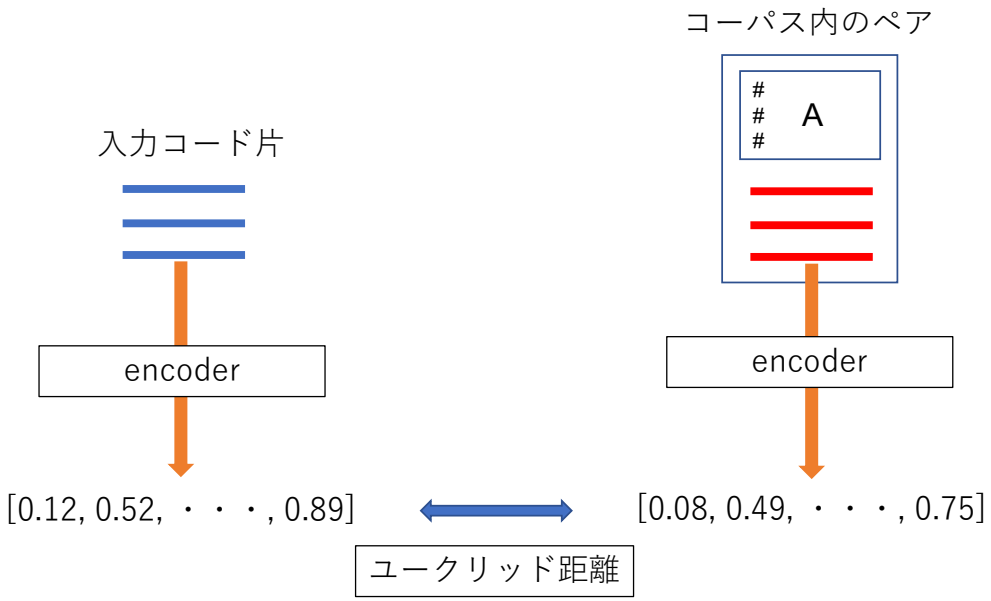

図 3.2. 未知のソースコードから最も類似するコメントの取得方法

3.2 前処理

本手法ではプログラムのソースコードから似たようなコメントを抽出するために, ソースコードに前処理を行う. これにより, 必要な情報だけを抽出してニューラルネットワークに学習させることができるようになる. 本節ではその具体的な処理方法について述べる.

3.2.1 ソースコードのトークン列化

Ruby のソースコードをトークン化するとき, Ruby の標準ライブラリである Ripper[*1]を用いて字句解析して, 改良したものを利用する. コメントの内容自体は, ソースコード中の空白や改行によって影響することはないと考えられるため字句解析で得られた空白情報や改行情報は削除した. また識別子はプログラマが自由に名付けられることができるが, 一般的に同じ識別子が何回も使いまわされたり違うプロジェクトで全く同じ命名がされたりすることは少ないため, 全識別子を区別することは非頻出トークンを考慮しすぎて過学習が行われると考えられる. しかし中には慣習的によく用いられる識別子も存在する. そういった識別子は区別して学習させた方が有用であると考えられる. そのためプロジェクトの中で 5 回以上使用された識別子については区別して,5 回未満のものはそれぞれ同じトークンに変換するようにする. 字句解析すると, 変数名は”on_ident”, 定数名は”on_const”にトークン化される. 区別する際は, 表 3.1 のように字句解析したトークンの後ろに”—-”と識別子の定義名を加えるようにする. また 5 回

[*1] https://ruby-doc.org/stdlib-2.5.1/libdoc/ripper/rdoc/Ripper.html

条件	変換前の例	トークンに変換後
5 回以上使われた変数名	require	on_ident—-require
5 回未満使われた変数名	create_action	on_ident
5 回以上使われた定数名	CONFIG	on_const—-config
5 回未満使われた定数名	FIGURE_X_SIZE	on_const

表 3.1. 変数名・定数名をトークンに変換するときの例

データの種類	トークンの表記
文字列リテラル	on_tstring_content
整数リテラル	on_int
小数点リテラル	on_float
シンボルの開始タグ	on_symbeg
演算子	on_op
文字列を開始するクオテーション	on_tstring_beg
文字列を終了するクオテーション	on_tstring_end
グローバル変数	on_gvar
ピリオド	on_period
カンマ	on_comma
開き括弧	on_lparen
閉じ括弧	on_rparen
開き中括弧	on_lbrace
閉じ中括弧	on_rbrace
開きブラケット	on_lbracket
閉じブラケット	on_rbracket
正規表現の開始タグ	on_regexp_beg
正規表現の終了タグ	on_regexp_end
式展開の開始タグ	on_embexpr_beg
式展開の終了タグ	on_embexpr_end

表 3.2. データの種類とトークンの対応表

未満の場合は字句解析した結果がそのままトークンとなる．その他一部の型と字句解析の結果の対応表を表 3.2 に示す．

Ruby には事前に決まった意味を持ち識別子として宣言できない予約語がある．これらの予約語は重要な意味を持ち, コメントの内容にも影響を及ぼすと考えられるため, 全て区別するようにする．予約語は字句解析でトークン化するとき”on_kw”に変換される．この後ろに”—-”と予約語を付け足すことによってトークン化処理をする．また識別子以外で出現回数が 5 回未

満のトークンは全て”UNK”トークンに置き換える．

3.2.2 コメントのペアとなるソースコードの抽出

ソースコード中にコメントを記入するとき, コメントは対象となるソースコードの周辺に書くことが多い．しかし,「周辺」がどこを指すかはプログラマに委ねられているため, 自明ではない．本節ではソースコードとコメントのペアを抽出するときに対象となるソースコードの抽出方法について説明する．

コメントを書く場所として考えられるのは以下のような場所である．

(a) 対象となるソースコードの前
(b) 対象となるソースコードの後
(c) 対象となる行の行末
(d) 対象となる行の行頭

(a) に書くことは最も直感的な場所であると考えられる．一般的にソースコードを読み進めていくときは, 上から下に向かって読んでいく．そのためソースコードの要旨やメタ情報を含むコメントはソースコードよりも前に書くことが妥当であると考えられる．

(a) として書かれたコメントの例は図 3.1 で示されている．これは実際に Github 上に上がっているリポジトリ [22] に書かれているコメントであり, 関数の前に説明を施しているコメントであることがわかる．このような事例は他にもたくさんあり, コメントは直下のソースコードの説明をしていることがほとんどである．

(b) として書かれたコメントの例を図 3.3 に示す．7 行目に書かれているコメントは 4 から 6 行目の範囲の Rubocop による警告を無効化したいときに書くものであり, コメントは対象となるソースコードの後に書かれている．また 15 行目に書かれているコメントは 12 から 14 行目の範囲をステートメントガバレッジの範囲外としたいときに書くものであり, コメントは対象となるコードの後に書かれいている．このようなコメントは割合としてはそこまで多くない．

(c) として書かれたコメントの例を図 3.4 に示す．これも (a) の例と同様のリポジトリに書かれているコメントであり, それぞれのコメントは行末に書かれいてることがわかる．このようなコメントは (a) のように書くこともできるが行数を多く使ってしまうため, 行数を抑えるためにこのように書かれることが多い．

また (d) で書かれるようなコメントはほとんど存在しない．そのため, 本研究では全てのコメントが (a) または (c) に属すると考えてコメントのペアとなるソースコードを抽出した．本来 (b) に属するようなコメントも (a) または (c) として抽出することになる．これは (b) のコメントは割合としては微々たるものでそこまで結果に影響しないと考えたからである．抽出するときの条件としてはソースコードが書かれている行と同じ行にコメントが書かれている場合は (c) とみなして同じ行のソースコードとコメントをペアとして抽出する．それ以外のコメントは直下のソースコードの塊を抽出する．もし直下のソースコードがキーワード”def”から始まる場合, コメントはブロック全体を対象として書かれていると考えられるため, ブロックが

```
1  # First Example
2
3  # rubocop:disable Metrics/AbcSize
4  def create_required_gen_vals_from(name)
5    --- inside algorithm ---
6  end
7  # rubocop:enable Metrics/AbcSize
8
9  # Second Example
10
11 # :nocov:
12 module Sail
13   --- inside algorithm ---
14 end
15 # :nocov:
```

図 3.3. 対象となるクラス・関数・モジュール・行の前にコメントが書かれた例

```
1
2  @data[:timeout]      = 3           # geocoding service
       timeout ・・・
3  @data[:lookup]       = :nominatim  # name of street address
       ・・・
4  @data[:ip_lookup]    = :ipinfo_io  # name of IP address ・・・
5  @data[:language]     = :en         # ISO-639 language code
6  @data[:http_headers] = {}          # HTTP headers for lookup
7  @data[:use_https]    = false       # use HTTPS for lookup ・・・
8  @data[:http_proxy]   = nil         # HTTP proxy server ・・・
9  @data[:https_proxy]  = nil         # HTTPS proxy server ・・・
10 @data[:api_key]      = nil         # API key for geocoding
       ・・・
```

図 3.4. 対象となる行の行末にコメントが書かれた例

```
1 on_kw----def on_kw----self on_period on_ident----address
    on_lparen on_ident----query on_comma on_ident----options
    on_op on_lbrace on_rbrace on_rparen on_kw----if on_lparen
    on_ident----results on_op on_ident----search on_lparen
    on_ident----query on_comma on_ident----options on_rparen
    on_rparen on_period on_ident----size on_op on_int on_ident
    ----results on_period on_ident----first
```

図 3.5. 抽出されたトークン化列の例

終わるまでのソースコードを抽出する．抽出はどれもトークン列数を 30 の固定長とする．キーワード”def”から始まりブロック全体を抽出したときにトークン数が 30 を超える場合は 30 になるように後ろのトークンを削る．また 30 に満たない場合は後ろに”EMP”トークンをトークン数が 30 になるまで加える．このようにして全プロジェクトを捜査していき, ソースコードとコメントのペアを抽出していく．図 3.5 は図 3.1 にある上のソースコードを抽出した結果得られるトークン化列である．

3.3 トークンのベクトル表現化

図 3.5 で抽出したトークン列を一つのベクトル表現に表すための前段階として, それぞれのトークンをベクトル表現化する必要がある．トークン化する手法は Word2Vec を利用する．Word2Vec は CBOW モデルと Skip-gram モデルを採用しているが, 本研究では Skip-gram モデルを使用する．学習データはプロジェクト内の Ruby で書かれている全てのソースコードとする．これはトークンの特徴量はコメント周囲のソースコードだけではなく, プロジェクト全体におけるトークンの分布を学習することによってより正確に特徴を汲み取れると考えたからである．

3.4 トークン列のベクトル表現化

トークン列のベクトル表現を得ることによって, ベクトル表現同士の類似度をユークリッド距離やコサイン類似度で簡単に測ることができるようになる．よって本節ではトークン列のベクトル化の手法について説明する．ソースコードの中に存在する互いに一致した, もしくは類似するコード片のことをコードクローンと呼ぶ．本手法はコードクローンに付随するコメントを提示するタスクとして見ることができる．コードクローンの研究は多くの場合教師あり学習で行われることが多く, 一昨年も教師あり学習を使ったコードクローンを検出する論文が二つでた [23][24]．しかし, 教師あり学習はデータセットに依存するため良いデータセットがない場合は性能が期待できないといった問題がある．そこで Liu *et al.*[25] は教師データを必要

図 3.6. トークン列のベクトル表現を得るための概略図

としない教師なし学習のコードクローンの検出を試みた．本手法ではこの研究を参考にして Seq2Seq モデルによってソースコードのベクトル表現化を得る．

3.4.1　Seq2Seq モデル

トークン列のベクトル表現を得るために LSTM を使った Encoder-Decoder モデル (Seq2Seq モデル) を使用した．入力はコメントのペアであるソースコードのトークン列で, 正解データも入力値と同じトークン列とする．図 3.6 に概略図を示す．図のように Encoder の最後の隠れ状態をソースコードを表すベクトル表現とみなすことによって未知のソースコードを入力したときにそのベクトル表現も得られるようになる．また Seq2Seq モデルの学習後, 既存のソースコードを入力して得られたベクトル表現と 3.2.2 で抽出したコメントを紐づけて大きなコーパスを作る．これによって未知のソースコードのベクトル表現から最も類似するベクトル表現をコーパスから算出することができ, それに紐づけられたコメントを提示するようにすることができる．

第 4 章

実験

本章では 2.3 節で提案した手法の有用性を測るための実験及び結果について述べる．まず初めに実験の目的について説明する．次に類似するコメントの定義について説明して, 実験で用いた評価指標とデータセットについて紹介する．次に前処理を行うプログラムの実装とモデルの詳しい設定について述べる．最後に実験結果を示し, 考察を行う．

4.1 実験の目的

本実験は 2.3 節で述べた提案手法が提示したコメントと実際の正解コメントとがどれだけ類似しているかを測定することによってその有用性を示すことが目的である．適切なコメントを書くことはコードリーディングを円滑に進め, 保守性を維持することにおいて大事である．既存の手法では 2.2.2 で述べたようなものがあるが, これらは人手のコストがかかったり, ソースコードの翻訳に留まってしまったりすることが多くまだ不十分な面が多い．本手法によってモデルが正解データと類似するコメントを提示することができれば, 良質なコメントを書くために必要な教育コストを抑えたり, コメントに対するコードレビューのやりとりが少なくなったりして, 本手法の有用性を示せると考えた．

提案されたコメントと正解のコメントの類似度合いを測るために三つの評価指標を用いた．二つの文の BLEU Score 値,Jaro Winkler 距離, そして長さの比較である．それぞれの評価指標については 4.3 で詳しく説明する．

4.2 類似するコメントの定義

本節では類似するコメントの定義ついて説明する．類似するコメントとはコメントを形態素解析して比較したときに 4.3 で述べる評価指標が高い値の時である．本手法は既存のコメントの中から参考になるコメントを提案するモデルである．ここで参考になるコメントとは, コメント自体の内容よりも書き方のパターンが共通していることの方が重要であると考える．コメントを形態素解析して品詞に変換させた上で構造の類似度を測ることによって, より適切に測定することができると考えた．これによりコメントの内容自体は全く異なったものでも, モデ

図 4.1. 形態素解析の例

ルがスタイルやカテゴリが似ているものを出力できているかを検証することができるようになる．例えば図 4.1 は二つの文の内容は全く異なるが形態素解析した結果同じ文になるため, 二つの文は「似ている」と評価される．本手法は形態素解析した結果の類似度が高ければ, コメントとして提案することは有用であると考える．

4.3　評価指標

前節で述べたとおり, 実験で用いる評価指標は BLEU Score 値,Jaro Winkler 距離, コメントの長さである．ここではそれぞれの評価値について述べる．

4.3.1　BLEU Score 値

BLEU Score 値は, 機械翻訳タスクなどで最もよく用いられる評価指標である．これは「モデルが, 人間の翻訳者が翻訳した文に近いもしくは一致する文を出力すれば翻訳精度が高い」という前提のもとで測られる指標である．翻訳者がある文を別の言語に翻訳する仕方は一通りではないため, 複数文を用意して測ることもできる．本実験は正解ラベルが一つしかないため複数文用意することはない．値は 0 から 1 の間の数字で表現され,1 に近いほど高品質な翻訳だと解釈できる．

BLEU Score の計算方法[*1]を以下に示す．

$$BLEU = BP_{BLEU} * exp(\sum_{n=1}^{N} w_n \log(p_n)) \tag{4.1}$$

$$p_n = \frac{\sum i \text{ 翻訳文 } i \text{ と参照訳 } i \text{ で一致した } n-gram \text{ 数}}{\sum i \text{ 翻訳文 } i \text{ 中の全 } n-gram \text{ 数}} \tag{4.2}$$

BP_{BLEU} は, 翻訳文が参照文よりも長い場合のペナルティである. もし参照文の方が長い場合はペナルティはなく, 値が 1 となる．また N-gram とは隣接する連続した N 文字のことであり,N は任意の数字を与える．一般的に多量に存在する翻訳文の BLEU Score 値の平均を算出したい場合は, 個々の BLEU Score は N を 1 から 4 で測り, その平均をとる．これによって算出されたそれぞれの BLEU Score の平均値を全体の平均値と見なす．これは N が小さい値だ

[*1] https://kakedashi-engineer.appspot.com/2020/06/06/bleu/

と, 長い文に対して不正確な値を算出してしまい, 逆に N が大きな値だと短い文に対して不正確な値を算出してしまうため N の値を変えて平均値を出すことによって値の揺らぎを抑えることが目的である．BLEU Score の値は 0.3 を超えると比較的良い翻訳とみなされる．しかし, この指標の欠点としては類義語やちょっとした活用系の変化に対しても違う単語と評価してしまい単純な字面しか考慮していないことと, 文の語順が考慮されていないことがあげられる．

4.3.2 Jaro Winkler 距離

Jaro Winkler 距離とは二つの文字列の類似度を文字列の一致度合いで測る指標である．一致度合いとは, 一致する文字数と置換の要不要から算出する．この値は 0 から 1 の間の数字で表現され, 完全一致する文字列に対しては 0, 完全に異なる文字列に対しては 1 を出力する．しかし Jaro Winkler 距離は厳密には距離ではなく類似度と定義されているため,1 が完全一致,0 が完全不一致となるように計算を改変している．Jaro Winkler 距離の計算方法[*2]を以下に示す.

$$\phi = W_1 * \frac{c}{d} + W_2 * \frac{c}{r} + W_t + \frac{(c-\tau)}{c} \tag{4.3}$$

$$\phi_n = \phi + i * 0.1 * (1-\phi)(i = 1, 2, 3, 4) \tag{4.4}$$

ここで W_1,W_2 はそれぞれ重みであり,W_t は置換する際にかかる重みを調整するための定数である．また d, r は文字列の長さであり,c は文字列の中で一致する文字の数,τ は文字列を編集する際に置換が必要な文字数を 2 で割った数値である．式 (4.4) は (4.3) から算出された値を用いて計算される Jaro Winkler 距離である．i は先頭の文字列を比較して一致する文字数 (最大で 4) に設定して計算すると距離が出る．これにより, 単語同士の類似度を数値化して比較することができる.

4.3.3 コメントの長さ

最後の指標はコメントの長さを比較することである．これはコメントの内容には触れず, コメントの文字列の数だけを比較したものである．もしソースコードの内容とコメントの長さに相関があるとすれば, 少なくともプログラマが書くべきコメントの分量がわかることになる．これは言い換えると具体的なコメントを書くべきか, 大まかなコメントを書くべきか, といった大きな方向性をユーザに示すことにつながる．このような情報がわかることによって Ruby を主言語としたプロジェクトに内在している暗黙的な慣習を発見する新たな知見につながると考えられる.

[*2] https://mieruca-ai.com/ai/levenshtein_jaro-winkler_distance/

表 4.1. 学習データのトークン数とペアの数

トークン数	ペアの数
9890310	329677

4.4　データセット

実験ではコメントの質が担保された学習データが必要となる．本実験では Github から Ruby を主言語としたリポジトリの中からスター数が 300 以上のリポジトリを 554 個抽出し, 学習データとテストデータに分ける．スター数の多いリポジトリはそれだけ世に評価されていると解釈でき, 監視の目も多くあるため, コメントの質もある程度維持されいていると考えることができる．またスター数で抽出することは比較的容易であり, 良質なコメントが含まれるデータを数多く集めることができる．上記のように抽出したコメントを良いコメントとみなして実験を行う．

学習データとテストデータの分け方はリポジトリ毎に分ける．テストデータは 10 個のリポジトリをランダムに抽出し, 残りを学習データとする．ユーザが本モデルを使用するときに学習済みのコーパスの中には作業しているリポジトリのコメントはコーパスの学習データに含まれていない．そのため, ユーザが作業しているリポジトリの中で検証したいソースコードとコメントのペア以外のペアを学習済みコーパスに含めて検証を行うことにより, より実用に則した実験を行うようにする．本実験ではこのように作成したデータセットで実用性を検証する．表 4.1 に学習データの総トークン数とコメントとソースコードのペアの数を示す．

4.5　コーパスの実装

提案手法では既存のコメントの中から最も類似するものを算出するために, ソースコードとコメントをペアにした大きなコーパスを用意することが必要である．4.4 で説明したように学習データとテストデータを分けた場合, 同じプロジェクトにある他のコメントはコーパスの中には存在しない．そのため学習済み seq2seq モデルを用いてコーパスを作成した後に, 検証するソースコードとコメントのペア以外の同じプロジェクトにあるソースコードとコメントのペアを seq2seq モデルに流して, ソースコードのベクトル表現を得る．得られたベクトル表現とコメントを紐づけて, コーパスに付け加えることによって同じプロジェクトファイルにある他のコメントも考慮できるようにする．これによって, もし適切なコメントが同じプロジェクト内にあったとしたら, それを提示することが可能になるようにする．

表 4.2. 利用したモジュールのバージョン情報

モジュール名	バージョン
Python	3.5.2
Ruby	2.3.1
Tensorflow	2.3.0
Keras	2.4.0
Gensim	3.8.0
Numpy	1.18.1

4.6 前処理の実装

3.2 で述べたように本実験ではデータセットを用意するためにソースコードのトークン化とコメントに付随しているソースコードの抽出が必要となる．これは Ruby の標準ライブラリである Ripper を用いてソースコードを字句解析した後に 3.2.1 に沿ってトークンを変換させた．

コメントのペアとなるトークン列の抽出は 3.2.2 に沿って実装した．”def”から始まるブロックの抽出方法は後ろの行が”def”のインデント数と一致するかで判断した．Ruby は柔軟な言語であるため, インデントを特に意識しなくてもプログラムは正常に作動する．しかしブロックの中はインデントして書くことが暗黙的な慣習となっている. 今回抽出したデータセットは全てこの慣習に則っているものとしてブロックの抽出を行なった．つまり”def”が記載されている行のインデント数と一致する初めての行をブロックの終わりだとみなして、一つの塊として抽出した．

4.7 モデルの実装

近年はニューラルネットワークが簡単に実装できるような質の高いライブラリが数多く公開されており, 機械学習による研究の敷居が低くなっている．今回は Keras という Tensorflow[*3] 上で動くニューラルネットワークライブラリを使用する．これは LSTM モデルや Skip-gram モデルを簡易的なコードで実装することができて, かつ GPU による計算も可能としている．各トークンのベクトル化はオープンソースライブラリである gensim が提供する Word2Vec を利用する．トークンの次元数は 100 として学習するときのコンテキストサイズは 20 とする．トークン列のベクトル表現を得るために,Keras が提供する LSTM を用いて Encoder-Decoder モデルを実装する．学習する時の LSTM の隠れ層の数は 256 とする. これによりトークン列のベクトル表現の次元数も 256 となる．またそれぞれの入力値として Python の拡張モジュールである Numpy を用いる．表 4.2 に利用したモジュールのバージョン情報を示す．

[*3] https://www.tensorflow.org/

4.8　実験結果

本節では行った実験とその結果を示す．まず初めに学習済みの Seq2seq モデルにテストデータを入力し, 算出されたコメントとテストデータの正解ラベルの BLEU Score 値を示した図を 4.2 に示す．テストデータの数は 144 個である．全体で見ると結果は左に寄っており, 良いコメントを出力できることが少ないことがみてとれる．しかし,BLEU Score が 0.8 を超えたあたりから正解データに近いコメントを選ぶ個数が増えていることもわかる．ここで正解に近いコメントを算出できているときの個々の結果を目視で確認したところ, その全てが入力コード片と同じリポジトリにあるコメントであることがわかった．そのため次に図 4.2 をシステムが出力したコメントが入力コード片と同じリポジトリであったか異なるリポジトリであったかで区別して内訳を可視化した．結果を図 4.3 に示す．違うリポジトリからコメントが算出された数は 91 個, 同じリポジトリからコメントが算出された数は 53 個であった．また, 一般的に BLEU Score 値が 0.4 を超えると, 高品質な精度が保障されていると言われる.[*4] この値を超えた数は 28 個であり, 全体の 19.4% であった. またその全ては同じリポジトリから選ばれたものであった.

モデルが違うリポジトリからコメントを提示するときは, ほとんどのスコアは 0.15 以下となり, 最も良いスコアでも 0.45 を超えないことがわかる．その一方で同じリポジトリからコメントが例示されたときは,BLEU Score の値が高くなる傾向にあることがわかる．全体で見ると, 高いスコアでコメントを算出できるものも存在するが, 低いスコアがほとんどでありこの段階では良い精度とみなすことはできない．またモデルが同じリポジトリからのコメントを出力した場合はその 52.8% が BLEU Score 値 0.4 を超えていた．ここで, 完全に一致したコメントの例と違うリポジトリから抽出されたコメントの中で最もスコアがよかったコメントの例, スコアが 0.10 を下回ったコメントの例, そして BLEU Score 値が 0.6 ほどだった時のコメントの例を図 4.4, 4.5, 4.6, 4.7 に示す．図 4.4 のようにコメントが完全に一致した時は, ソースコードもほぼ一致していることがわかる．また図 4.5 のように異なるリポジトリから抽出されたコメントの中で最もスコアがよかったコメントの BLEU Score 値は 0.36 程だったが, 結果を目視で確認すると出力値の書き方のフォーマットが一致していることがわかりスタイルが似ているコメントをうまく選んできていることが観測できる．図 4.6 のようにスコアが最も低い例では, ソースコードは似ていることが感じ取れるが, コメントは全く異なるものとなった．また図 4.7 は BLEU Score が 0.6 ほどであった時のコメント例であり, 返り値の書き方のフォーマットや False の設定の仕方などが似ていることがわかる. このように BLEU Score が高いと構造が似ているコメントを抽出できていることがわかった.

次に JaroWinkler で測った時の図を 4.8 に示す．図からわかるように両者とも比較的高い値を示している．しかし,JaroWinkler は単語の類似度を測ることに特化しており, 文の類似度を測ることには向いていない可能性がある. これは 4.3.2 の式からわかるように, 初めの数文字が

[*4] https://kakedashi-engineer.appspot.com/2020/06/06/bleu/

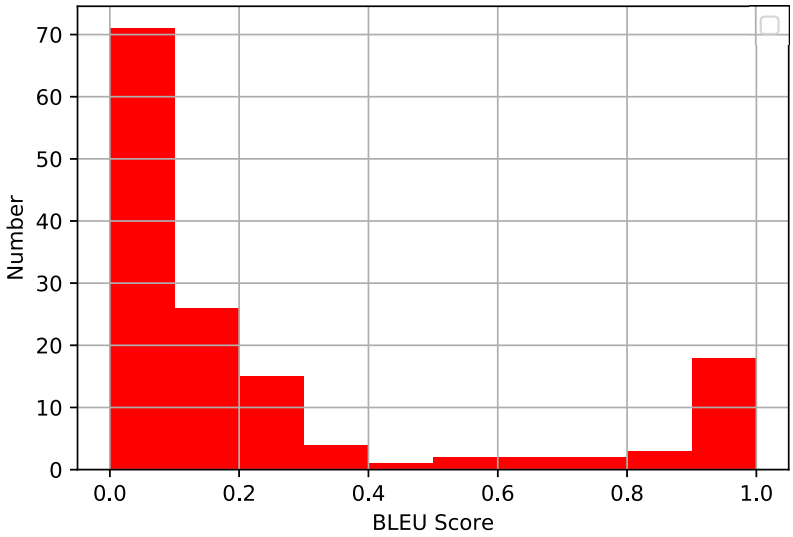

図 4.2. モデルが出力したコメントと正解データのコメントの BLEU Score 値

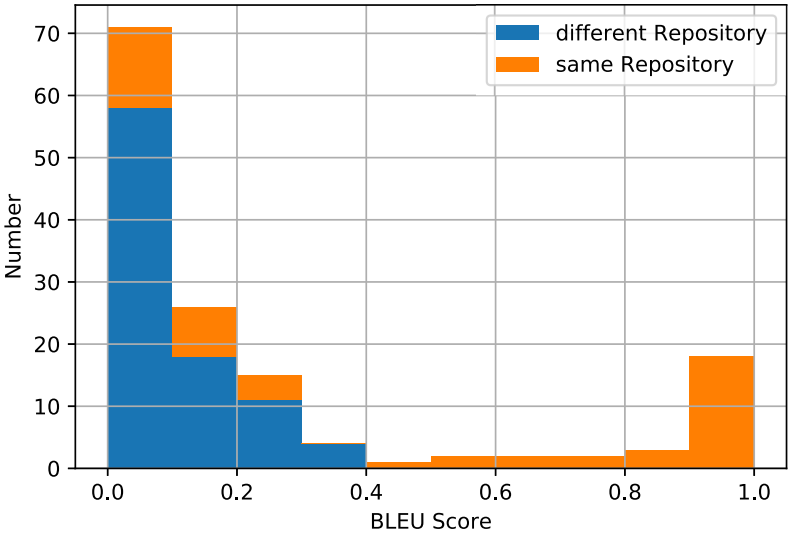

図 4.3. モデルが出力したコメントが同じリポジトリか異なるリポジトリかの内訳

```
# TEST DATA

# Tap the delete button
view('Just another deletable blank row').superview.superview.
    subviews.each do |subview|
  if subview.class.to_s == confirmation_class
    tap subview
    wait 0.01 do

==================================================
# ANSWER DATA

# Tap the delete button
view('A non-deletable blank row').superview.superview.
    subviews.each do |subview|
  if subview.class == confirmation_class
    tap subview
    wait 0.01 do
```

図 4.4. モデルの出力と正解ラベルで完全一致した例

一致しているだけで高い値を算出することになるからである. 例えば図 4.5 で示したコメントの場合 BLEU Score 値は 0.42 だったが,JaroWinkler で測ったところ, 結果は 1.0 となる. 初めの数文字が一致しているだけで, 文全体が完全一致したものとしてみなされてしまう. よって JaroWinkler による文の評価は信憑性に欠けてしまう.

次にコメントの長さを比較した結果を図 4.9 に示す. 図からわかるように, 同じリポジトリから提示されたコメントはある程度相関がある. 実際, 同じリポジトリだけに限定して相関係数を測ったところ 0.34 とやや相関があった. しかし, 異なるリポジトリから抽出されるコメントの相関係数は 0.13 と小さく, また予想されたコメントも正解データと外れているものも散見される.

以上のようにそれぞれの指標で測ったが, この段階では本モデルの性能が十分とは言えない. ここからは BLEU Score に着目してさらに結果を分析する.

BLEU Score の結果からは良いコメントを提示するときもあるが, 約 8 割は参考にならないコメントを提示することがわかった. ここで, その原因として 2 つの理由が考えられる.

(a) コーパスに良いコメントが含まれていない
(b) 本実験で定義したソースコードの類似度とコメントの類似度には相関がない

```
# TEST DATA

# @return [Boolean] Returns true if current engine is `:
    creole`.
def creole?
  @current_engine == :creole
end

==================================================
# ANSWER DATA

# @return [Boolean] True if connected to NgameTV
def ngametv?
  @name == :ngametv
end
```

図 4.5. 異なるリポジトリから提案されたコメントの中で最も高い値の例

```
# TEST DATA

# Test using secure cookies
def test_cookie_with_secure
  url = URI.parse('http://rubygems.org/')

==================================================
# ANSWER DATA

# If no path was given, use the one from the URL
def test_cookie_using_url_path
  url = URI.parse('http://rubygems.org/login.php')
```

図 4.6. モデルの出力と正解ラベルのスコアが低い

```
# TEST DATA

# Whether or not to require a password confirmation. If you don't want your users
# to confirm their password just set this to false.
#
# * <tt>Default:</tt> true
# * <tt>Accepts:</tt> Boolean
def require_password_confirmation(value = nil)
  rw_config(:require_password_confirmation, value, true)
end

==========================================================
# ANSWER DATA

# In order to turn off automatic maintenance of sessions when updating
# the password, just set this to false.
#
# * <tt>Default:</tt> true
# * <tt>Accepts:</tt> Boolean
def log_in_after_password_change(value = nil)
  rw_config(:log_in_after_password_change, value, true)
end
```

図 4.7. BLEU Score が 0.6 ほどであった時の例

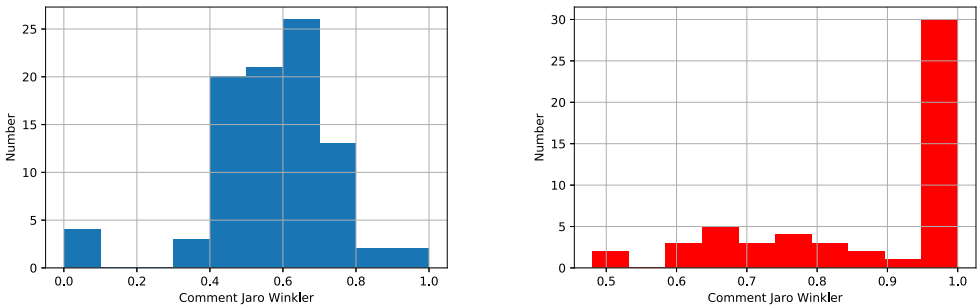

図 4.8. JaroWinkler で測ったとき. 右は異なるリポジトリからコメントを提示, 左の図は同じリポジトリからコメントを提示した時

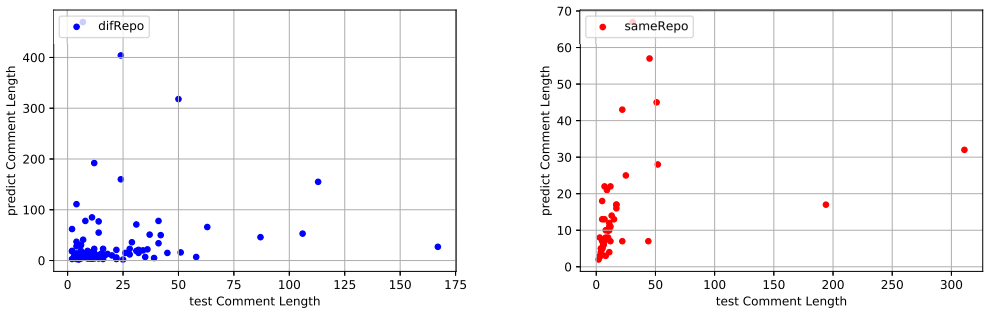

図 4.9. 長さで比較した時の結果. 右は異なるリポジトリからコメントを提示, 左の図は同じリポジトリからコメントを提示した時

(a) について, もしコーパスの中に正解コメントに近いコメントがそもそもない場合, モデルがどれだけ精度の良いものになったとしても良いコメントを提示することはできない. そこで, コーパスの中に正解コメントに近いコメントが存在するかを検証するために, テストデータの正解のコメントを利用する. 正解コメントとコーパスの中にある全てのコメントとの BLEU Score 値を計算して最も高かったコメントを抽出する. BLEU Score 値をプロットすることによってコーパスの中に参考になるコメントがそもそも存在するかを確認する.

(b) について, そもそもソースコードの類似度とコメントの類似度に相関がない場合は, ユークリッド距離が最も近いコメントを提示してもそれが最も参考になるコメントである保証がなくなる. これによりシステムのコメントの選び方に問題があると考えることができるようになる. ここではそれぞれについて検証するための追加実験を行なった.

まず (a) の実験結果をヒストグラムにした図を 4.10 に示す. この図はシステムが原理的に提案できる最良のコメントのスコアである. コーパスにある最良のコメントとの BLEU Score はモデルが予想したコメントとの BLEU Score よりも大きいことがわかる. 違うリポジトリから予測したコメントに関して, モデルは最大で 0.35 程のものしか見つけることができなかったが, 実際は 1.0 付近のコメントがコーパスの中に存在していた. また同じリポジトリのコメントのスコアが 0.2 以下になるものはほとんどない. 全体で見ると BLEU Score が 1.0 付近にあるものが 40 個ほどあり正解コメントに近いコメントがコーパスの中にある程度存在することがわかる. これにより (a) の可能性はあまりないことが考えられる.

次に (b) についての実験結果を図 4.11,4.12 に示す. 図 4.11 の左図は最も距離の近かったベクトル表現に紐づいているコメントが完全一致したものである. モデルは一致したコメントを提示できたことになる. しかし他の点はほとんどが BLEU Score の値が 0.15 を超えることはなく, コーパスにあるほとんどのコメントは類似していないことがわかる. 右図は最も類似しているコメントが中央にあり, それだけが突出している様子がわかる. モデルは最も左にあるコメントを提示するため正しいコメントを提示できていない. またこの図からもユークリッド距離とコメントの間に相関がないことがわかる. 図 4.12 の左図はそもそも正解データがコーパスの中にない場合のプロットである. この時は BLEU Score が低い位置に満遍なく散らばっており, 図 4.11 の右図と同様に相関は見られない. また右図は複数のコメントが正解付近にあ

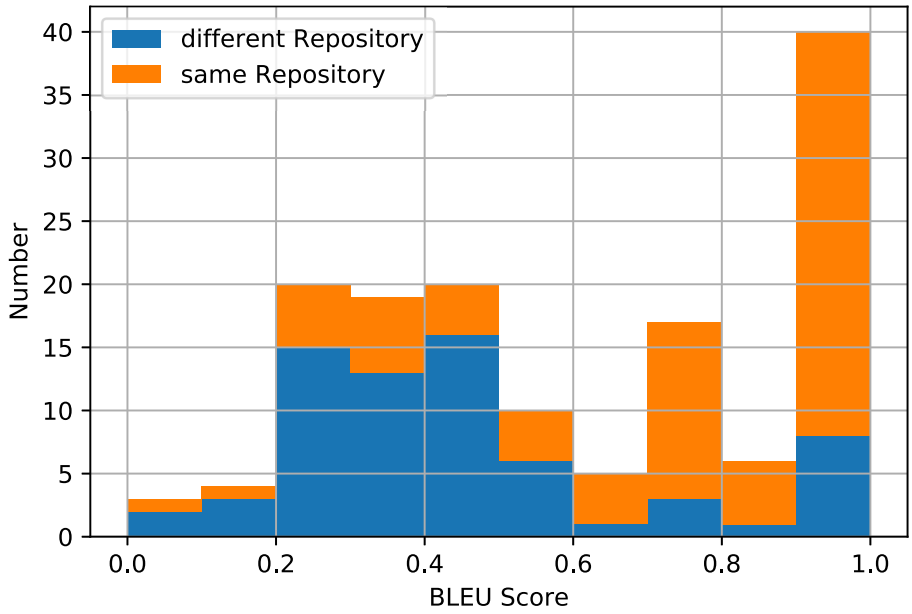

図 4.10. コーパスの中にある最も参考になるコメントの BLEU Score をヒストグラムにしたグラフ

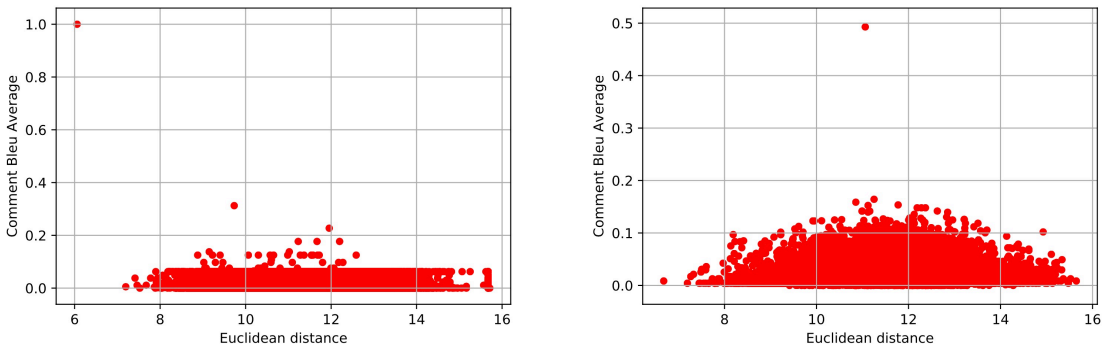

図 4.11. 一つのテストデータに着目してコーパスにあるユークリッド距離と BLEU Score の遷移をプロットした図. それぞれの図で一つのテストデータに着目している

るが, いずれもユークリッド距離が近い時には分布していない.

このように全体としてソースコードの類似度とコメントの類似度には相関があまり見られないことがわかる. よって本手法の前提となっていた「ソースコードの類似度が似ていれば, コメントの類似度も似ている」という前提が本実験手法においては成り立たないことがわかる.

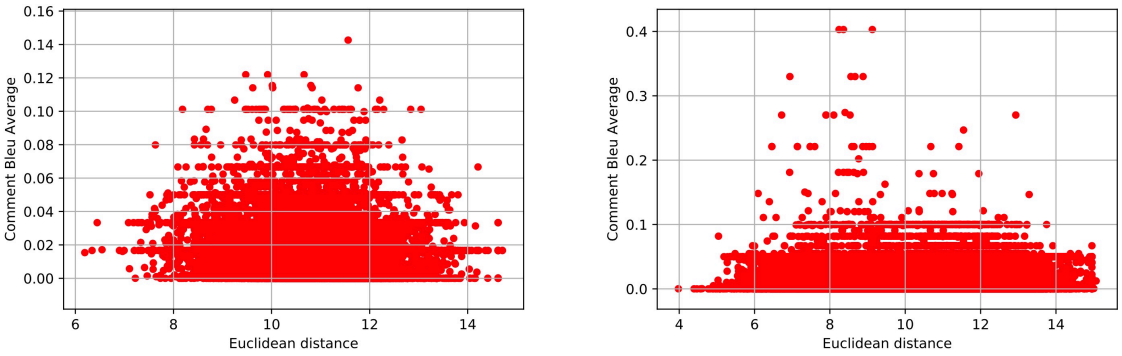

図 4.12. 一つのテストデータに着目してコーパスにあるユークリッド距離と BLEU Score の遷移をプロットした図. それぞれの図で一つのテストデータに着目している

4.8.1 結果のまとめ

本実験結果についてまとめると, モデルが入力コード片と同じリポジトリにあるコメントを出力した場合その 52.8% は BLEU Score 値が高品質な精度が保障されていると言われる 0.4 を超えることがわかった. これによりモデルが入力コード片と同じリポジトリにあるコメントを出力した場合, その約半分は参考になる可能性が高いことを示唆している. また個々のデータを目視で確認したところ BLEU Score が 0.3 を超えたあたりからスタイルやカテゴリが似ているものが多く確認された. ここから BLEU Score が 0.3 を超えたものは参考になるコメントであると判断できる可能性が高いことが考えられる. 最後に, 入力コード片とコーパス中のコード片とのユークリッド距離とコメントの BLEU Score の間には相関はみられなかった. ここから本手法では参考になるコメントをうまく算出できていないことがわかった. 実際 BLEU Score が 0.3 を下回るテストデータは全体の 80.5% であり, ほとんどが参考にならない結果となった. さらにその後の実験からコーパスの中には参考になるコメントが存在することが確認できたためアプローチ自体は問題はないように思われる. よってコーパスの中に存在する参考になるコメントの算出方法に改良の余地が残されていることがわかった. コメントの算出方法として本実験ではコードのベクトル表現を,Seq2Seq モデルというよく知られた手法を用いてベクトル化して単純なユークリッド距離でコード間の類似度を定義したが, ここに改善の余地があると思われる. 改善手法の一つとしてコーパスの中に存在する最も参考になるコメントのペアとなるソースコードとテストデータのソースコードの間に相関を持つように教師あり学習をさせてベクトル表現を得るようにすることが考えられる. こうすることによって図 4.11,4.12 がより相関を持つような分布に学習されると考えられる.

第 5 章

まとめと今後の課題

5.1 まとめ

本研究ではソースコードの類似度が似ていればそれに付随するコメントも類似するという仮説に基づいて, 既存のコメントの中から最も適切なコメントを提示するシステムの開発を試みた. 近年, プログラミングの義務教育化や, プログラマのニーズが高まっていることから, 今後プログラマの人口は増えていくことが予想される. それにより, 特に Ruby といった柔軟性の高いプログラミング言語では統一されたコーディング規約を遵守するとともに, 他のプログラマとの連携を容易にするために質の高いコメントを書くことが重要になってくる. しかし既存の手法ではまだ質の高いコメントを書くには多くの課題が残されている. そこで本研究では機械学習のアプローチを用いて, 既存のコメントの中から適切なコメントを提示するシステムの開発及びその有用性を検証した. 本手法では, 初めに Skip-gram モデルによって得られたトークンの分散表現をもとに, トークン列のベクトル表現を Encoder-Decoder モデルから計算した. 算出されたベクトル表現とペアになっているコメントを紐づけて大きなコーパスを作った. その後, 未知なソースコードを入力したときに学習済みの Encoder-Decoder モデルが算出したソースコードのベクトル表現とコーパスにある全てのベクトル表現とのユークリッド距離を計算して, 最も距離の近いものに紐づいているコメントを提案するシステムを開発した. 結果として BLEU Score の値は精度として高品質と言われる 0.4 を超えるテストデータは, 出力が同じリポジトリのコメントに限定すると,52.8% あることがわかった. またコメントは同じリポジトリにあるものの方がより類似するコメントが存在する傾向にあることもわかった. しかし, 実用上にはまだ課題が多く, さらに検査を進めたところ, ユークリッド距離とコメントの類似度の間には相関がないことがわかり, そもそもの仮説である「ソースコードの類似度が近ければコメントの類似度も近くなる」が本実験手法および検証方法においては成り立たないことがわかった.

5.2 今後の課題

本手法では完全に一致する定型文などに対してはある程度の結果が出ることがわかったが, 全体で見ると改良する点がいくつかある. 今後の課題としては, まず BLEU Score がコメントの類似度を上手く表しているかを検証する必要があると考える. BLEU Score は語順が考慮されていなかったり, 人間の評価と大きく異なったりすることがあるのでこれだけでは一概に似ていると判断することは難しい. そのため, 例えばプログラマを数人集め本手法が提案したコメントが参考になったかのアンケートを行うといった主観的な評価を交えることも有用であると考える.

次にデータセットの見直しが必要であると考える. 本手法では主言語が Ruby のリポジトリでスター数が多いものに存在するコメントは質が高いとみなし学習データとして学習させた. しかし, これは自明なことではなく, かつプロジェクトの更新頻度も考慮されていないため, 古いコメントが混ざっている可能性はある. それにより誤った学習を行い結果に反映されてしまったことが考えられる. これは学習データを改良することによって解決されると考えられる. しかし「質の良いコメント」だけを集めたデータセットは存在しないため, 手作業で作ることが必要であると考える.

次にソースコード及びコメントの「類似度」の定義の再考が必要である. 本実験ではソースコードの類似度は Seq2Seq モデルでベクトル表現化した後にユークリッド距離を測り, 距離が近いものを似ていると評価した. またコメントは形態素解析した後に BLEU Score を測り値が大きいものを似ていると評価した. それぞれの手法でうまく似ているものを測定できることは自明ではなく, また二つの指標の間には相関がないこともわかった. 前者の改善手法としてはコーパスの中にある最も参考になるコメントのペアとなるコードとテストデータのコードが相関を持つように教師ありで学習させることがことが考えられる. 後者についてはそもそもこの評価の仕方でスタイルやカテゴリが似ているコメントをうまく測定できているかを検証する必要がある. 目視で評価を確認することや事前に正解データを手動で振り分けてうまく類似するコメントを計算できているかを確認する必要がある.

次に, コメントの内容に起因する要素は何かを調査することが必要であると考える. コメントの内容はソースコードの内容に大きく依存すると考えられたが, 単純な字面だけでは内容は決まらない可能性がある. 例えばもっと大きな範囲を考慮に入れ, どこのディレクトリ構造に位置するかといった情報や抽象構文木を利用してよりセマンティクスな特徴を捉えたデータに整形し直して検証することが必要であると考えられる.

前述のようなアプローチを検討することによって, より実用可能な精度まで向上させることが必要であると考える.

発表文献と研究活動

(1) 白石誠, 千葉滋. コード内にコメントを入れる時に適切なコメントを例示するシステムの開発. 日本ソフトウェア科学会第 38 回大会, 2021.9.1-3.

参考文献

[1] 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.

[2] V. Markovtsev, W. Long, H. Mougard, K. Slavnov and E. Bulychev, "Style-Analyzer: Fixing Code Style Inconsistencies with Interpretable Unsupervised Algorithms," *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019*, pp. 468-478.

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 281–293.

[4] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shan- ping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* (2017).

[5] Cookpad styleguide. `https://github.com/cookpad/styleguide/blob/master/ruby.ja.md`

[6] Airbnb styleguide. `https://github.com/airbnb/ruby`

[7] cumcumber/cucumber-ruby's pull request. `https://github.com/cucumber/cucumber-ruby/pull/1566`

[8] Fighting Evil in Your Code: Comments on Comments. `https://www.red-gate.com/simple-talk/opinion/opinion-pieces/fighting-evil-code-comments-comments/`

[9] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10)*. Association for Computing Machinery, New York, NY, USA, 43–52.

[10] Steve McConnell, CODE COMPLETE, Microsoft Press, second edition, section

32.4.(2004)

[11] etewiah/property_web_builder. https://github.com/etewiah/property_web_builder

[12] coreingrastructure/best-practices-badge https://github.com/coreinfrastructure/best-practices-badge

[13] Kimura Satoshi. *Eclipse* **で学ぶはじめての** *Java (***初版***)*. SB クリエイティブ, pp. 203–224, 2008.

[14] Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. 2020. Where should I comment my code? a dataset and model for predicting locations that need comments. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 21–24.

[15] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension (ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 200–210.

[16] Tomas Mikolov, et al. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*.2013.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. 1st International Conference on Learning Representations, ICLR, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings.

[18] Sepp Hochreiter and J ¨ urgen Schmidhuber. Long short-term memory. *Neural computation*, Vol. 9, pp. 1735–80, 12 1997.

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

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

[21] Cho, Kyunghyun, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint*, 2014

[22] alexreisner/geocoder. https://github.com/alexreisner/geocoder

[23] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. pp. 518–528, 05 2019.

[24] Kawser Nafi, Tonny Kar, Banani Roy, Chanchal Roy, and Kevin Schneider. Clcdsa: Cross language code clone detection using syntactical features and api documentation. pp. 1026–1037, 11 2019.

[25] Liu Yuze and Shigeru Chiba. Attempts on detecting cross-language code clones using only monolingual training data without labels. The University of Tokyo. 2020.

謝辞

本論文は, 東京大学情報理工学系研究科創造情報学専攻千葉研究室において行った研究の成果をまとめたものである.

本研究の遂行にあたり, コロナ禍による難しい環境でも, 毎週のオンラインミーティングなどでご指導ご鞭撻いただきました本学千葉滋教授に心から感謝いたします. 研究方針や発表練習, 社会勉強, メンタルの維持など多くの面で支えられました.

本研究の遂行にあたり, ご多忙な中毎週のミーティングで多くのアドバイスや的確な助言をいただきました本学鵜川始陽准教授に深く感謝いたします.

日ごろより親身になってご相談に乗っていただき, 研究の進め方や考え方をご指導いただきました本学山崎徹郎特任助教に深く御礼申し上げます.

毎週のミーティングで素朴な疑問から鋭いコメントまで多くの意見とご指導をいただきました本学穐山空道創造情報学専攻助教に深く感謝いたします.

2 年間の研究生活でなかなか会うことはできませんでしたが, オンラインツールなどを用いてコミュニケーションをとり励まし合ってきた同期の依田和樹君, 三富秀和君, 横井駿平君に感謝いたします.

最後に, 研究を進める上でご相談をさせていただきました先輩方, 同期, そして研究室のスタッフの皆様に深く感謝いたします. 本当にありがとうございました.