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

白石 誠　千葉 滋

本研究では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開発を試みた．コメントの質を上げることはコードの保守性を上げるための重要な要素である．しかし，現在存在するリファクタリングツールやソフトウェア解析ツールはコメントの添削においてまだ未成熟な点が多い．本研究ではコメントが周辺のコード片によって特徴付けられるという仮説を立て，Ruby を主言語としたリポジトリでコメントと周辺のコード片の特徴の関連づけを学習させた．現在コメントを挿入したい場所のコード片の特徴に最も合致する既存のコメントを適切なコメントの具体例として示すシステムの開発を目指している．

1　はじめに

ソフトウェア開発においてソースコード中に書くコメントはプロジェクトのタイトなスケジュールなどによりわかりづらく書かれているものが多い．近年コードの要約やコーディングスタイルのルール化などの研究が盛んに行われるようになってきているが，実用上でまだ課題が多く見られる．

そこで本研究ではコメントは周辺のソースコードによって決定づけられ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計算し，具体例として示すシステムの開発を目指している．実験計画としては 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コメントのある周辺のソースコードを LSTM によってベクトル化する．そのベクトル化されたソースコードを全結合層に入力して，得られる出力をコメントの分散表現とみなして，事前に学習したコメントの分散表現とコサイン類似度で最も距離の近いコメントを計算する．そのコメントを最終的な出力値として適切なコメントとみなす．

2　コメント記述の支援

ソフトウェア開発において開発者はソースコードの内容の理解に 59%の時間 [1] を割いていると言われている．そのためプロジェクトの開発効率をあげるためにはソースコードの内容の理解に割く時間を減らすことが大事である．

ソースコード中に書くコメントは開発者に理解しやすいように書くためのものであり，コメントの質をあげることはソースコードの内容を理解するためには必要な要素である．

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，実装を重視するところや納期が迫っているような場合では質のい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．ここでいう不適切なコメントというのは例えば以下のようなものが考えられる．

```
1  # if isOpen is true
2  if(isOpen)
3    --- inside algorithm ---
```


白石 誠, 東京大学, The Unviersity of Tokyo.
千葉 滋, 東京大学, The Unviersity of Tokyo.

```
4   end
```

これは if 文の条件式を説明したコメントであるが，コードに書かれていることをそのまま言語化しているだけで良いコメントであるとは言えない．本来であれば，より粒度を高く何をしている条件分岐なのかを説明することが望ましい．そのため以下のようなコメントが望まれる．

```
1   # if modal is opened
2   if(isOpen)
3     --- inside algorithm ---
4   end
```

またクラスに書くコメントに関しても以下のような例があげられる．

```
1
2   # . Magic Do not touch
3   # will be executed when cache error .
        occurs
4   class Image::CacheError
5     --- inside algorithm ---
6   end
```

これはクラスの説明をしたコメントとなるが，その「クラスの責務」や「ソースコードをどう利用するべきか」などの情報がなく，後にメンテナンスしたり，引き継ぎしたりする際に，ソースコードの意図がわからず，無駄な工数が増えてしまう．そのため例えば以下のようなコメントが望まれる．

```
1   # This class allows you to configure
        how to treat errors when the cache
         is not .
        available
2   # In order to use it use it like . this
3   # ~~~/~~~/. imagerb
4   # . Imageconfigure(
5   # :cache => Image::. CacheErrornew()
6   # )
7
8   class Image::CacheError
9     --- inside algorithm ---
10  end
```

このようにコードからは読み取ることが難しいが，コメントを読むだけで何をしているかわかるような書き方が適切だと考えられる．

3 似たような文脈に登場するコメントを例として提示する

本研究ではコード内にコメントを挿入する際に，周辺のソースコードの特徴量に最も合致する既存のコメントを，適切なコメントの具体例として示すシステム開発を目指す．将来的には総合開発環境やエディタに組み込まれて使われることを想定している．

3.1 機械学習モデルによる実現

上記のシステムを開発するアプローチとして機械学習を使った手法を提案する．我々は，コメントは周辺のソースコードによって特徴づけられるという仮説を立て，その特徴を機械学習によって学習できることを期待する．

本研究で提案するモデルは，入力値としてコメントを挿入したい場所のソースコードを受け取り，その特徴量から類推される最も適切なコメントを事前に学習した既存のコメントの中から計算し，出力するものである．

学習データとしては Github から集めたスター数の多い Ruby のリポジトリを使う．ソースコードの中のコメントのある部分を抽出し，コメントの分散表現を作成する．またコメントの周辺にあるソースコードも分散表現に変換する．ソースコードの分散表現を入力値，それに対応するコメントの分散表現を正解ラベルとして学習を進める．

周辺のソースコードは LSTM によって得られる分散表現を用いて，コメントは Keras の Doc2Vec[2] を用いてベクトル化する．LSTM によってベクトル化されたソースコードは全結合層を通してコメントの分散表現を出力するようなモデルである．

モデルは図 1 のように入力層，中間層，出力層からなる．

(a) 入力層

Github から集めたスター数の多い Ruby のリポジトリを集め，ソースコードの各トークンを Keras の Word2Vec[3] に通してトークンの分散表現を作成する．keras の Word2Vec は単語

のベクトル化の手法として CBOW(continuous Bag-of-Words Model) と Continuous Skip-gram Model(以下 skip-gram) の 2 種類がある．CBOW モデルとはターゲットの単語を予測するために，その前後の単語を与えて学習する手法であり，その副産物として単語の分散表現が得られる．skip-gram モデルとはある単語が与えられた時にその前後の単語を予測するようなものであり，これも副産物として単語の分散表現が得られる．今回は skip-gram を用いることした．skip-gram により得られたトークンの分散表現を LSTM モデルの入力値として，最後の隠れ層の出力を文全体の分散表現として取り出す．LSTM は与えられたソースコードで次に来るトークンを予想させるタスクを与えて各パラメータを最適化する．

(b) 中間層

入力層で得られた文の分散表現を全結合層の入力値として入力し，計算する．訓練時においては出力されたベクトルがコメントの分散表現と合致するように，入力層で入力したソースコードに対応するコメントの分散表現を正解ラベルとして確率勾配降下法を用いてベクトルを最適化させる．中間層の出力で得られたベクトルを出力層の入力値として値を渡す．

(c) 出力層

Github から集めたコメントを事前に Keras の Doc2Vec を用いてコメント文の分散表現を得る．keras の Doc2Vec は PV-DM(分散記憶モデル) を採用している．これは訓練時に与えられた文書ベクトルを連番で番号を振っていき，その番号と文書ベクトルを入力値として，出力を文書の次の単語となるように学習するモデルである．中間層で得られたベクトルと事前に得たコメント文の分散表現をコサイン類似度で計算し，最も距離が近いコメント文を「適切なコメント」とみなし，最終的な出力値とする．

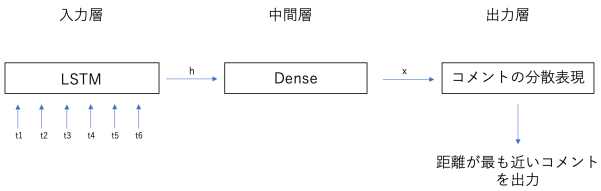

図 1　モデルの概要図

3.2　本システムが良いと考えられる理由

ソースコードの似たような文脈に登場するコメントを例として提示することが良いと考えられる理由として以下のよう点があげられる．

(a) 既に存在するプロジェクトファイルの中には経験則的に良いとされるコメントが数多く存在する．例えば，Github のスターの数が多いリポジトリでは多くの人が開発に携わっており，コメントも多くの人が理解しやすいように書かれている．そのためこれらのコメントを学習することにより，良いコメントの特徴をうまく抽出できると考えられる．

(b) 周辺のソースコードから類推されるコメントにはパターンがあると考えられるため既存のコメントを例示するだけでも有用である考えられる．例えば以下のような例を示す．

```
1   ##
2   # Look up the coordinates of the
        givien street or IP .
        address
3   #
4   def . selfcoordinates(,
        address options = {})
5     if(results = search(,
          address options)). size > 0.
          .
6       resultsfirstcoordinates
7     end
8   end
9
10  ##
11  # Look up the address of the given
          coordinates([,
        latlon])
12  # or IP adress(string).
13  #
14  def . selfaddress(,
        query options = {})
```

```
15      if(results = search(,
            address options)). size > 0.
            .
16        resultsfirstaddress
17      end
18    end
```

これらはいずれもメソッドの説明をしているコメントであり，周囲の書き方も類似している．またコメントも「○○が与えられたら△△を検索する」という構造が一致している．これより周囲のソースコードの類似性はコメントを書く際には参考になり得ると考えることができる．

4 実験計画

本節では今後の実験計画について述べる．

4.1 データセット

学習データは Github からスター数が 1000 以上の Ruby のリポジトリを 554 個集める．集めたリポジトリの Ruby ファイルだけを抽出し，Ruby のプログラムパーサである Ripper を使って各トークンとコメントを取り出す．各トークンの内リポジトリに 5 回以下しか出現しないようなトークン “UNK” トークンに置き換える．トークンがリテラルだった場合は，その型に置き換える．空白や改行などは事前に削除した．またコメントが周辺コードのどこにかかっているかは自明でないため，全てのコメントはその直後のソースコードにかかっているかもしくは前後のトークンによって特徴づけられるとして場合分けをして実験する．周辺のソースコードとして取り出すトークンの長さは 20，30，40，50 で実験する．554 個集めたリポジトリの内ランダムに 80%を取り出して学習データとする．

5 関連研究

ソースコード中のコメントの改善に関するコメントはいくつかある．

Louis *et*, *al.* (2020)[4] はコメントを挿入するべき場所を提案するモデルを作成した．これはあるコード片が与えられた時にそのコードに対してコメントを挿入するべきかの有無を課題とした研究である．手法としては，多層パーセプトロンによる学習，シーケンシャルモデル (LSTM) による学習，そして Hierarchical Sequence(Hier) モデルによる学習を行いその性能を比較した．結果としては Hier の精度が最も高く，Precision が 74%， recall が 13%であった．しかしこの研究はコメントの場所にフォーカスした研究であり，コメントのテキスト自体は提案しない．

また Hu *et*, *al.* (2018)[5] は DeepCom という Java のメソッドに限定して適切なコメントを自動生成する手法の提案，開発を行った．評価指標としては BLEU-4 を使っており，結果は 38．17%となった．しかし，これはソースコード中のメソッドに限定したものであり，かつコメントというよりはコード要約としての側面が大きい．

6 まとめと今後の課題

本論文ではソースコード中にコメントを挿入する際に周辺のソースコードから既存のコメントのうち最も適切なものを提案するモデルの手法についてのと実験計画について述べた．しかしまだ考慮しなければならない点がいくつかある．まずソースコードの分散表現を獲得する手法がとして STM ではない適切ではない可能性がある．これは LSTM モデルは最後のトークンの情報が最も強く保持されてしまうため，先頭に近いトークンの記憶の保持が難しい特徴があるからである．また非頻出単語についての扱いにも検討の余地がある．ソースコードを分散表現に変換するとき，自然言語と大きく異なる点が，変数名など単語の出現頻度として低いものが大量に存在することである．ただ闇雲に”UNK”と置き換えただけではほとんどのトークンが”UNK”に変換されてしまい，結果に大きく作用することが考えられる．これは変数名をその型情報に変換することによって改善されることが考えられる．また評価指標についても考慮しなければならない．現在用いる予定である BLEU-4 は予測されたコメントと正解ラベルの類似度を計算する．しかし，トークンの順序が考慮されていなかったり，字面しか評価されず類義語が使われても評価ば悪くなってしまったりと決して評価指標として最適ではな

いと考えられる．これらの改善の検討が必要である．

参 考 文 献

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

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word represetations in vetor space. In Proceedings of Workshop at ICLR(2013).

[3] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In NIPS. 3111–3119(2013).

[4] Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. Where should I comment my code?: a dataset and model for predictingg locations that need comments. Proceedings of the ASM/IEEE 42nd international Conference on Software Engineering: New Ideas and Emerging Results(2020). pp. 21-24.

[5] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. Proceedings of the 26th Conference on Program Comprehension(2018). pp. 200-210.