ソースコードに対するコメント位置推定のための抽象構文木に基づいた特徴量についての考察

嘉陽 桃子[1,a)]　當間 愛晃[2]　赤嶺 有平[2]　山田 孝治[2]　遠藤 聡志[2]

概要：GitHub などオープンソースソフトウェアホスティングサービスの普及にともない、ソースコードは読まれることを想定する必要性が強まった。ソースコードの理解を助ける手段として、ドキュメントの整備や可読性の高いソースコードの作成、コメントの付与などがあるが、本研究ではとくに、記述箇所や自然言語の中に製作者の意図が反映されているコメントに焦点を当てた。コメントは、複雑なソースコードに対する解説や、ソースコードの表面上からは読み取れないような補足として使われる。本研究では、前者を目的としてソースコードの複雑さの特性を分析し、コメントに適した部分を特定を目指す。複雑さの指標として、抽象構文木を採用する。本研究では、コメントが付けられた位置を複雑な処理と仮定し、複雑さの定量化を目的とした分析を行う。構文木の構造に複雑さが表れていると予測し、ソースコード全体の抽象構文木から特徴を抽出した。結果より、構文木の深さとルートノード名からコメントの有無に関する特徴があらわれていることが分かった。

キーワード：抽象構文木，コメント, 開発支援

1. 研究背景

ソースコードは作成者だけでなく、多くの人に読まれる場面を想定し記述する必要がある。例えば、プロジェクトの運用の際には引き継ぎや保守に向けた配慮をする、ソースコードを Web 上に公開し様々な意見をもらうために可読性に配慮しレイアウトを整えたりコメントを記述する等である。そのような場面において、ソースコードを書く側と読む側ではソースコードの理解にギャップが生じる可能性がある。そのため、書き手側は多くの場合ソースコードの理解を助ける手段を用意する。理解を助けるための手段の例として、ドキュメントの整備や可読性に配慮した記述方法、コメントの付与等が考えられる。ドキュメントはシステムの要件定義やファイルの依存関係などプロジェクトの全体像を把握することができる。可読性に配慮したレイアウトや記述方法を行うことで処理を把握しやすくなるだけでなく、修正を容易にすることやバグを防ぐ効果がある。コメントは実装方法選択の理由等についてソースコード記述時の背景や記述者の意図を知ることができる。しかし、書き手のプログラミングに対する経験量や知識の差異により、このような手段を十分に行えない可能性がある。例えば、関数にコメントを記述する際に、書き手は関数の動作概要をコメントとして記述したが、読み手は実装方法選択の理由や関数内部に設定された定数の意味を知りたい場合である。本研究では、このような書き手側の経験量の不足により生じるギャップを埋めることを最終目的とし、適切なコメント記述のための支援を目指す。

事前準備として学内で 4 名にコメント記述のためのアンケートを実施した。回答者には Python で記述された三目並べのシミュレーションを行うソースコードであることのみを伝え、後輩に引き継ぐことを前提とした際に、コメントを記述するならどの箇所 (単一行〜複数行) に書くかを回答してもらった。ソースコード内にはソースコードの概要を説明するコメント、各関数を説明するコメントをそれぞれ 1 行ずつ済みのコードであり、コメント不要箇所を除外するという選択肢が含まれたアンケートとなっている。回答者はソースコードの各行に対してコメントを記述する場合はチェック印を記入し、どのような内容のコメントが必要か回答した。アンケートの結果、コメントとして記述する内容は回答者ごとに異なるが、コメントを記述する位置には傾向が見られた。そこで、本研究では「適切なコメン

[1] 琉球大学大学院理工学研究科
University of the Ryukyus, Aza Sembaru, Nishihara-cho, Nakagami-gun, Okinawa, 903-0213
[2] 琉球大学工学部工学科知能情報コース
University of the Ryukyus, Aza Sembaru, Nishihara-cho, Nakagami-gun, Okinawa, 903-0213
[a)] k178585@ie.u-ryukyu.ac.jp

ⓒ 2020 Information Processing Society of Japan

ト」は適切な内容と適切な位置に分けられると考え、コメント記述に適した位置の特定を目指す。アンケートでは if 文や定数が存在する位置にコメントが記述されやすい傾向にあったことから、コメントの記述されやすさは処理内容と相関があると考えた。コメントが記述されやすい処理を特定することでその位置へコメント記述を促すことが可能となり、書き手側への支援に繋がると考えられる。本研究ではコメント記述が必要な処理を定量的に評価し、コメント記述に適した位置を特定することを目的とする。

2. 関連研究

Dustin らはリーダブルコード [1] において、情報を伝えるためのコメントの書き方を指南している。自分の考えを記録する必要があると感じた場合やソースコードよりもコメントを読んだほうが早く理解できる場合にはコメントが必要であると述べている。つまり、コメントの役割は、記述時の背景などのソースコードの表面上からは読み取れない情報の提供や、ソースコードの理解を容易にするための情報の補完であり、それらを記述する際には読み手の立場に立つことが重要であることがわかった。しかし、どのようなコメントが読み手に必要な情報を与えるか、読み手の立場に立ったときにどのような箇所に疑問を持つかなどを想像することは書き手の経験への依存が大きいと考えられる。そのため、ソースコードを読む際に円滑な理解の助けとなるコメントを記述するための支援を行うことが本研究の立ち位置である。

CODE COMPLETE[2] ではコメント内容の分類が行われている。

(1) コードの繰り返し:ソースコードが行うことを別の言葉に言い換えたコメント

(2) コードの説明:複雑なソースコード、細心な注意が必要なソースコードを説明するために使用されるコメント

(3) コードの目印:作業が途中であることを開発者に思い出させるためのマーカーの役割を持ったコメント

(4) コードの概要:数行のコードを 1 つか 2 つの文にまとめたコメント

(5) コードの意図の説明:ソースコードの一部分の目的を説明するためのコメント

(6) コード自体では表せない情報:著作権の告示、オンラインリファレンスの参照先等のコメント

本研究では適切なコメント記述位置の特定を目指すが、「適切なコメント」について内容と位置は完全に切り離せるものではないと考えている。そのため、CODE COMPLETE におけるコメント内容の分類から書き手の経験量に左右されやすいコメントを抽出することで、本研究の目的達成のための位置を特定すべきコメントを詳細に設定できると考えた。(1)、(4) は、例えばソースコード全体や関数などの処理ブロック全体に対する概要説明等の翻訳にあたるため経験量に依存しないコメントであると考えた。(2)、(3)、(5)、(6) はリーダブルコードにおけるソースコードの表面上からは読み取れない情報の補完となるコメントであると言える。ソースコードの表面上からは読み取れない情報には、書き手がソースコードを記述する際の背景や意図が含まれており、このようなコメントを記述するためには書き手側にある程度の経験量が必要であると考えられる。本研究では (2)、(3)、(5)、(6) を経験量の必要なコメントであるとし、そのようなコメントが記述されやすい処理の特定を目指す。

3. 提案手法

アンケートからコメントが記述されやすい処理の傾向が確認でき、関連研究から記述するべきコメントの内容を設定した。本研究ではコメント記述に適した位置を定量的に評価するため、抽象構文木を用いた分析を行う。

3.1 取得するデータについて

1 ソースコードとコメントの条件

- a Python で記述されている
- b GitHub にて最もスターが多くつけられたリポジトリ (2019 年 9 月)
- c コメント記号「#」の前にスペースかタブが 1 文字以上入力されている

2 ソースコード内の処理とコメントの扱い

- a コメントはその次の行の処理へ言及しているとする
- b ソースコードは 1 行単位に分割する

1-b は読み手側から最も評価されたソースコードを取得する意図がある。スターには様々な意味合いの評価が含まれるが、評価を行うユーザーの中にはソースコードを読むユーザーが存在する。そのユーザーがソースコードを読む際に理解の補助となったコメントが存在すると考えた。1-c は [2] にて分類された (2),(3),(5),(6) のコメントを抽出するためのものである。Python は記述上、インデントが重要な意味を持つプログラミング言語であるため、コメントは言及する処理と同じ位置までインデントされている可能性が高いと考えられる。また、ソースコード全体に対するコメントや関数全体に対するコメントは [2] の (4) にあたるため除外したい。以上より、関数内部の処理等に対するコメントはインデントされている可能性が高いと考え、このような条件を設定した。

2 は本実験におけるデータの扱いである。コメントは複

数行の処理に対して言及する場合があるが、その範囲を特定することは容易ではない。そのため、まずはソースコード1行とコメント1行をセットとして扱い、分析を行いコメントの付きやすい処理とそうでない処理の特徴の解明を目指す。

3.2 ラベルの作成

ソースコードを1行単位に分割し、それぞれの行に対してコメントされているか否かラベルを付与する。処理が記述されている上の行に3.1節の1-cの条件と一致するコメントが記述されていた場合はコメントありのラベル、記述されていない場合はコメントなしのラベルが付与される。この処理はソースコードの全行に対して行われるので、構文木として抽出できたすべての行にはいずれかのラベルが付与される。

3.3 特徴ベクトルの作成

本研究では、コメントに適した位置はソースコードの構造から判別できる予測し、抽象構文木を用いて構造を表現する特徴を抽出する。抽象構文木とは、トークンの並びから解析木を構築する処理である [5]。字句解析において文字列をトークン列に分解した際には意味解析には括弧等の必要のないトークンが存在する。そのようなトークンを排除した構文木をとくに抽象構文木と呼ぶ。本研究では、抽象構文木 (以下、構文木) の各ノードからソースコードの構造をあらわす3つの特徴を抽出し、ベクトルに変換する (以下、ノードベクトル)。ノードベクトルを接続することで構文木のベクトルとして表現する (以下、構文木ベクトル)。各ノードから抽出した3つの特徴を以下に示す。

- ノードが存在する位置の深さ
- オブジェクト名 (以下、ノード名)
- 親ノードの Index

3.4 評価方法

作成したラベルと構文木ベクトルからコメントの有無を判別することができるか、単純ベイズ分類器を用いて評価を行う。単純ベイズ分類器は分類対象のデータがそれぞれ独立であると仮定し、確率モデルの性質にもとづいて教師あり学習を行う学習モデルである。本研究においては、構文木ベクトルとコメントの有無をあらわすラベルが同時に出現する確率を学習することで、コメントの付きやすい位置の特定を目指す。

分類成功率が高い場合は設計した構文木ベクトルからコメントの有無を判別できているとし、低い場合は失敗分析を行うと同時に構文木ベクトルやラベルの設計、作成方法等の改善についての考察を行う。

3.5 学習データの作成

Python の標準ライブラリである AST モジュールを用いる。

1. 深さ優先探索を行い、3.3節の3つの特徴を各ノードに渡す。
2. それぞれの特徴を以下の表現でベクトルに変換する。
   - ノードが存在する位置の深さ: 離散値
   - ノード名: カテゴリ値
   - 親ノードの Index: カテゴリ値

   ノードが存在する位置の深さは数値の大きさに意味があると考えたため連続値とした。それ以外はカテゴリとしての意味を持たせるため one-hot 表現とした。
3. 3.1節の1-cの条件に一致するコメントと、コメントが記述されている行番号を抽出する。
4. 1行単位に分割された処理をデータとするため、構文木を行番号ごとに分割する。
5. 3、4を3.1節の2-aの条件に基づき紐付ける。
6. 不均衡なデータ数による学習の偏りを避けるため、コメントありのラベルが付与された構文木ベクトルのデータ数を上限とし、コメントなしのラベルが付与された構文ベクトルのデータをランダムで抽出した。

4. データセット

4.1 データセットの内訳

GiHub にてもっともスターが多く付けられた Tensorflow のリポジトリを取得しデータセットとした。データセットの内訳を以下に示す。なお、コメントなしのラベルが付与されたデータ数は16506件だったが、学習の際にデータ数の偏りを考慮しコメントありのラベルが付与されたデータ数に合わせてランダムで644件を抽出した。

- 取得したファイル数: 49件
- ファイルの合計行数: 17150行
- コメントなしのラベルが付与されたデータ数: 644件
- コメントありのラベルが付与されたデータ数: 644件

4.2 データセットの観察

本研究ではコメントに適した位置を探るため、コメントが記述されたソースコードの構造を分析し、コメントが記述されやすい処理にどのような特徴があるか探る。ソースコードの構造をあらわすため、抽象構文木から3つの要素を抽出しベクトル化した。コメントが記述されやすい処理はひと目で見て内容が把握できない処理という仮定のもと、まずは構文木の深さを分析する。構文木が深いほど式が複雑であったり、設定する条件が多い処理である可能性

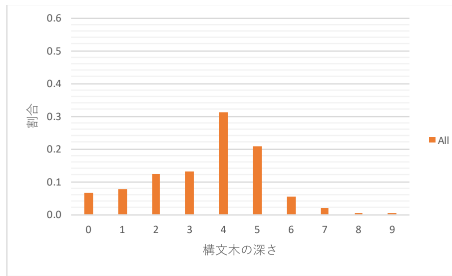

図 1　データセット全体の深さの分布

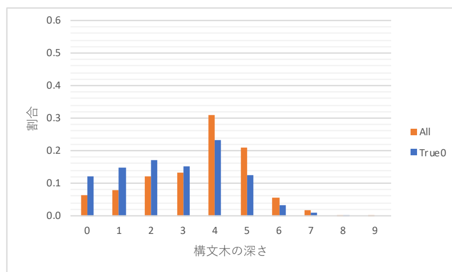

図 2　データセット全体の深さの分布とコメントなし正解データの深さの分布の比較

があると考えた。次に、処理内容とコメントの記述されやすさにどのような関係があるか探るためノード名の分析を行う。各ノード名は記述された処理をあらわす最小の要素であるため、単体のノード名やノード名の組み合わせからラベルごとに特徴が出ている可能性がある。また、コメント記述者は構文木の深さではなく、処理の内容からコメントの可否を決定していると考えられる。そのことから、ノード名の分析は構文木の深さとは異なる特徴を探るためにも有効である。本研究の位置づけとして、構文木の深さの分析は処理の複雑さとコメントの有無の関係を探るものであり、ノード名の分析は処理内容とコメントの有無を探るものである。

4.2.1　構文木の深さ

ノードが存在する位置の深さの最高値を構文木の深さとして扱い、ラベルごとの構文木の深さの割合を確認する。以下の図はラベルごとの構文木の深さの割合を示した図である。本研究で使用したデータセットでは構文木の深さは 0 から 9 である。データセット全体の深さの分布を平均とし、ラベルごとの深さの分布と比較してコメントの有無と構文木の深さの関係を分析する。割合はすべて小数点以下 2 桁で切り捨てて表示する。

- データセット全体の深さの割合

  図 1 について、本実験で使用したデータセットには、深さ 4 の構文木が 31.1%と最も多く含まれていた。深さから特徴を探るため、最頻値である 4 を中心として、深さ 0 から 3 を浅い構文木、5 から 9 を深い構文木とする。データセット全体では浅い構文木は 39.8%、深い構文木は 29%含まれていた。

- コメントなしのラベルが付与されたデータの深さの分布

  コメントなしのラベルが付与された構文木 (以下、コメントなし正解データ) は深さ 4 の構文木が 23.2%と最も多く含まれていた。浅い構文木は 59.6%、深い構文木は 17%含まれていた。図 2 について、データセット全体の深さの分布を基準としたとき、コメントなし正解データは深さ 0 から 3 が上回っているため、浅い構文木にコメントなし正解データの特徴が現れている可能性がある。

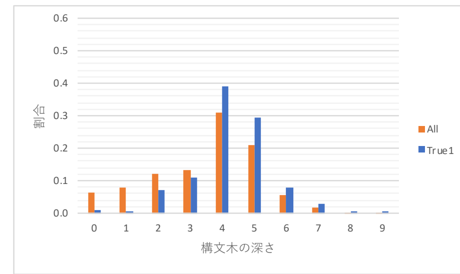

図 3　データセット全体の深さの分布とコメントあり正解データの深さの分布の比較

- コメントありのラベルが付与されたデータの深さの分布

  コメントありのラベルが付与された構文木 (以下、コメントあり正解データ) は深さ 4 の構文木が 38.9%と最も多く含まれていた。浅い構文木は 20%、深い構文木は 40%含まれていた。図 3 について、データセット全体の深さの分布を基準としたとき、コメントあり正解データは深さ 4 から 9 が上回っているため、深い構文木にコメントあり正解データの特徴が現れている可能性がある。

4.2.2　ノード名

処理内容とコメントの記述されやすさにどのような関係があるか探るため、ラベルごとに構文木の各ノード名の出現傾向の分析を行う。今回は構文木のルートに位置するノードのノード名 (ルートノード名) の傾向を確認する。また、ノード名は Meet the Nodes[6] において大まかな処理内容ごとに分類されているため、ラベルごとに出現するノード名をさらにカテゴリー名でまとめることによってコメントの有無と処理内容の傾向が見えると考えた。

一方のラベルにのみ出現するルートノード名を抽出し、そのラベルの特徴ルートノード名とする。また、それぞれの特徴ルートノード名が分類されるカテゴリー名も列挙する。今回は両ラベルに存在するカテゴリー名に分類される特徴ルートノード名の分析を行う。カテゴリー名は大まかな処理内容をあらわすので、両ラベルに同じカテゴリー名が存在することは、似た処理を行うにも関わらずコメントが記述される処理とされない処理に分かれたと考えられる。そのようなコメントの有無を分けた要因から、構文木の構造から見た処理内容とコメント記述者から見た処理内

ⓒ 2020 Information Processing Society of Japan

CategoryName	Feature_RootNodeName	Number
Literals	Dict	7
Literals	List	5
Literals	NameConstant	21
Function and class definitions	arg	16
Function and class definitions	ClassDef	5
Function and class definitions	Lambda	1
Variables	Name	33
Variables	Starred	1
Expressions	BinOp	2
Statements	Import	2

表 1　コメントなしラベルが付与された構文木の特徴ルートノード名

CategoryName	Feature_RootNodeName	Number
Expressions	BoolOp	2
Control flow	Try	2

表 2　コメントありラベルが付与された構文木の特徴ルートノード名

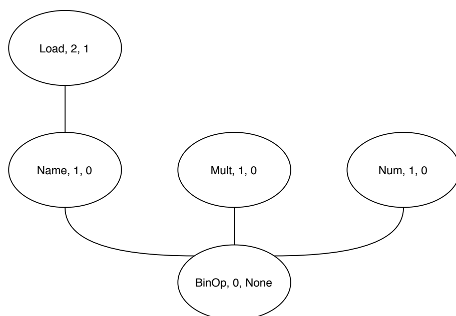

図 4　図 6 の 957 行目から作成された BinOp をルートノードに持つ構文木

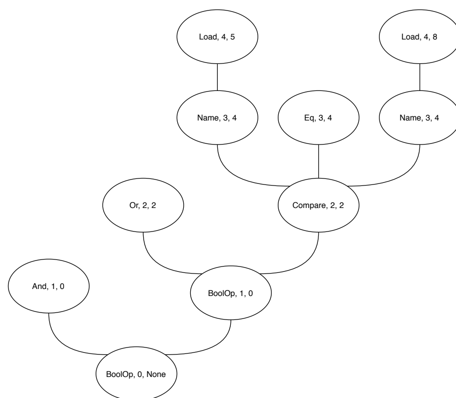

図 5　図 7 の 392 行目から作成された BoolOp をルートノードに持つ構文木

```
956 |  |  logging.info('Expect the performance of LSTM V2 is within 80% of '
957 |  |  | | | | | | | 'CuDNN LSTM, got {0:.2f}%'.format(cudnn_vs_v2 * 100))
```

図 6　BinOp をルートに持つ構文木のもととなるソースコード

```
391 | | | | |  # Either user specified GPU or unspecified but GPU is available.
392 | | | | |  (device_type == _GPU_DEVICE_NAME
393 | | | | |  | or (device_type is None and context.num_gpus() > 0))
394 | | | | |  and
395 | | | | |  (mask is None or is_sequence_right_padded(mask, self.time_major)))
```

図 7　BoolOp をルートに持つ構文木のもととなるソースコード

容にギャップが生じている可能性があると考えた。

表 1、表 2 のラベル名の概要を以下に示す。CategoryName より、両ラベルに出現するカテゴリー名は Expressions である。コメントなし正解データの特徴ルートノード名は BinOp、コメントあり正解データの特徴ルートノード名は BoolOp であった。BinOp は二項演算を行う際に現れるノードである。使用したデータセットにおいて BinOp をルートに持つ構文木の例を図 4 に、構文木の元となったソースコードを図 6 に示す。図 4 より、変数と数値の単純な掛け算を行っていることが分かる。BinOp の子ノードは 3 つだが、変数については値を読み込む処理のみなのでシンプルな構造であることが分かった。図 6 の 957 行目より、文字列と同時に出力される計算を行っていることが分かる。出力される文字列や cudnn という単語が含まれる変数名から処理速度などに関するメッセージの出力であることが分かる。また、Num ノードにあたる 100 はマジックナンバーだが、文字列や変数名からパフォーマンスの出力のための値であると予想できる。BinOp をルートに持つ構文木について構造がシンプルであること、ソースコードについて自然言語情報が多く含まれることから処理内容の把握が容易であることが分かった。

BoolOp は or か and のブール演算を行う際に現れるノードである。使用したデータセットにおいて BoolOp をルートに持つ構文木の例を図 5 に、構文木の元となったソースコードを図 7 に示す。図 5 より、BoolOp は BinOp より構造が複雑であることが分かる。BinOp では子ノードに変数や値があったが、BoolOp では BoolOp、And、さらに孫ノードでは Compare などの条件の設定のために値同士を比較するノード名が多く出現する。そのため複雑な構造になっていると考えられる。図 7 の 392 行目より、デバイスと GPU を比較していることが分かる。コメントを読むことで、この処理ブロックはユーザーが GPU を指定したか否かを確認し動作を行う処理であることが分かる。コメントが記述されていない場合、変数名や条件から処理の内容を把握する必要があるが、ひと目でその判断を行うことは難しいと言える。BoolOp をルートに持つ構文木は条件の設定などで構造が複雑であること、ソースコードについて処理内容を把握するためには変数名と条件の組み合わせから判断しなければならないことが分かった。

以上より、値を計算するのみのシンプルな構文木コメントが記述されにくく、条件設定のある複雑な構文木はコメントが記述されやすい傾向にあると考えられる。また、構文木とソースコードでは読み取れる情報に違いはあるが、どちらもコメントの有無について同様の傾向が現れていることが分かった。このことから、コメント記述者から見たコメント記述の可否を決定する条件は構文木からも判別できる可能性がある。

ⓒ 2020 Information Processing Society of Japan

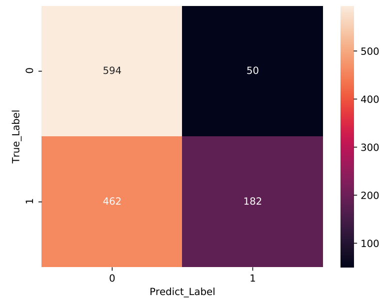

図 8 confusion matrix

	precision	recall	support
0	0.56	0.92	0.70
1	0.78	0.28	0.42
accuracy			0.60
macro avg	0.67	0.60	0.56
weighted avg	0.67	0.60	0.56

表 3 classification report

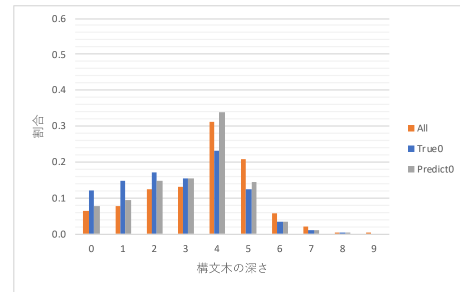

図 9 データセット全体の深さの分布とコメントなし予測データの深さの分布の比較

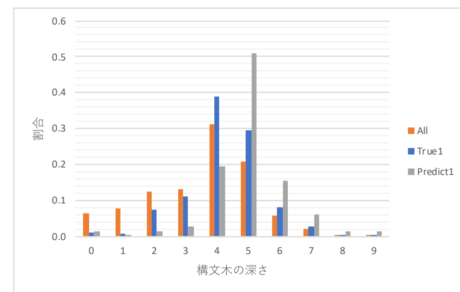

図 10 データセット全体の深さの分布とコメントあり予測データの深さの分布の比較

5. 実験

5.1 実験結果

混合行列と再現率、適合率の値から学習の傾向を観察する。表 3 より、分類正解率は 60%だった。本研究において設計した特徴ベクトルは、単純ベイズ分類器において偏った学習をしたことが分かる。

- コメントなしラベルについて
  図 8 より、コメントなしと予測されたデータは 1056 件、そのうち、正しく分類されたデータは 594 件だった。また、表 3 の recall の値が高い理由として、使用したデータのほとんどがコメントなしと予測されたためである。データセットにおいてはコメントあり・なしサンプル数を揃えていたにも関わらず「コメントなし」と誤識別しやすい傾向となったことから、設計した特徴ベクトルにはコメントなしと判断するための特徴が多く含まれていたと考えられる。
- コメントありラベルについて
  図 8 より、コメントありと予測されたデータは 232 件、そのうち、正しく分類されたデータは 182 件だった。また、表 3 の precision の値が高いことから、コメントありと予測されたデータの分類成功率が高いことがわかる。

5.2 考察

5.2.1 構文木の深さについて

各ラベル予測データの構文木の深さ分布をデータセット全体の構文木の深さの分布と比較し正解データの特徴との差異を分析する。また、予測データの中から分類に失敗した構文木の深さの分布から、構文木の深さが学習にどの程度寄与したか考察を行う。分布同士の比較には JS ダイバージェンスを用いる。JS ダイバージェンスは 2 つの確率分布の距離を測る手法である。値が小さいほど 2 つの分布は似ていると言え、まったく同じ分布同士の比較では JS ダイバージェンスの値は 0 となる。また、すべての JS ダイバージェンスの値は小数点以下 5 桁で切り捨てて表示する。

- ラベルごとの予測データの観察

  – コメントなしと予測されたデータの深さの分布
    図 9 より、コメントなしと予測された構文木 (以下、コメントなし予測データ) は深さ 4 の構文木が 33.7% と最も多く含まれていた。浅い構文木は 47.3%、深い構文木は 18.9%含まれていた。また、コメントなし予測データは深さ 0 から 4 がデータセット全体の深さの割合を上回っている。図 9 において、コメントなし正解データは深さ 0 から 3 に特徴があらわれていたことから、コメントなしと判別される深さの範囲が広くなったと考えられる。
  – コメントありと予測されたデータの深さの分布
    図 10 より、コメントありと予測された構文木 (以下、コメントあり予測データ) は深さ 5 の構文木が 50.8%と最も多く含まれていた。浅い構文木は 5.6%、深い構文木は 75%含まれていた。また、コメントあり予測データは深さ 5 から 9 がデータセット全体の深さの割合を上回っている。図 10 において、コメン

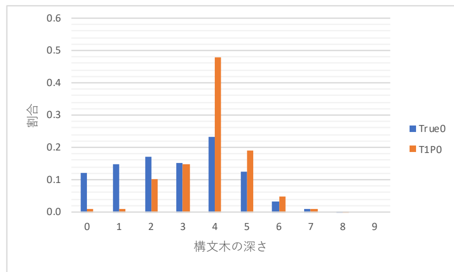

図 11 コメントなし正解データの深さの分布と誤ってコメントなしと予測されたデータの深さの分布の比較

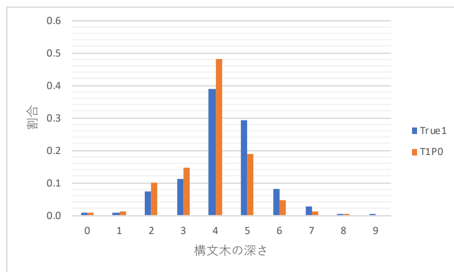

図 12 コメントあり正解データの深さの分布と誤ってコメントなしと予測されたデータの深さの分布の比較

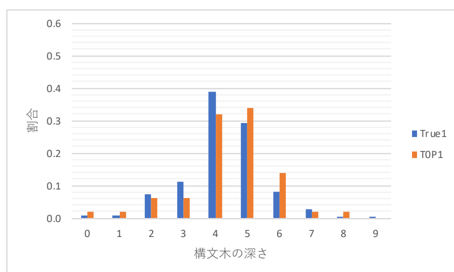

図 13 コメントなし正解データの深さの分布と誤ってコメントありと予測されたデータの深さの分布の比較

トあり正解データは深さ 4 から 9 に特徴があらわれていたため、それよりも深い位置に構文木が集中している。

深さ 0 から 4 にコメントなしと判別された構文木が多く、深さ 5 から 9 にコメントありと判別された構文木が多く含まれていたことが分かった。コメントあり予測データの深い構文木が 75 %を占めていたことから考え、深い構文木はコメントあり、それ以外はコメントなしと学習した可能性が考えられる。

- 分類失敗データ

分類に失敗した構文木ベクトルの深さの分布を正解データと比較することで、失敗の要因が構文木の深さかそれ以外の特徴かを分析する。

– コメントなし予測データ

図 11 と図 12 は、コメントなし予測データの中の誤って分類されたデータの構文木の深さの分布と、正解データの両ラベルの構文木の深さの分布と比較した図である。図 11 の JSD は 0.05427 であり、図 12 の JSD は 0.00419 だった。JSD の値より、図 12 の分布同士が近いことが分かる。誤ってコメントなしに分類されたデータは、コメントあり正解データの深さの分布と似ているにも関わらずコメントなしへ分類された。このことから、誤ってコメントなしに分類されたデータは構文木の深さとは異なる特徴を強く学習していることが分かる。

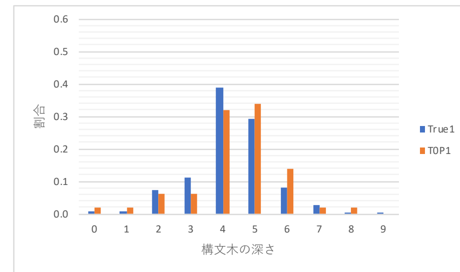

図 14 コメントあり正解データの深さの分布と誤ってコメントありと予測されたデータの深さの分布の比較

Feature_RootNodeName	GraphDepth	AccuracyRate
arg	0	100%
ClassDef	2∼3	100%
Lambda	4	100%
Attribute	2∼3	84%
BinOp	1∼2	100%
Dict	0∼4	86%
List	2∼4	80%
NameConstant	0	95%
Num	0	73%
Str	0	83%
Import	1	100%
ImportFrom	1	87%
Raise	3	75%
Name	1	91%
For	2∼3	67%
Attribute	2∼3	84%
For	2∼3	67%
ImportFrom	1	87%
Num	0	73%
Raise	3	75%
Str	0	83%

表 4 コメントなしと予測された構文木の特徴ルートノード名

– コメントあり予測データ

図 13 と図 14 は、コメントなし予測データの中の誤って分類されたデータの構文木の深さの分布と、正解データの両ラベルの構文木の深さの分布と比較した図である。図 13 の JSD は 0.01760 であり、図 14 の JSD は 0.00126 だった。JSD の値より、図 14 の分布同士が近いことが分かる。誤ってコメントありに分類されたデータは構文木の深さの分布が似ているためにコメントありに分類されたことが分かる。

Feature_RootNodeName	GraphDepth	AccuracyRate
Assert	4~7	67%
BoolOp	4	100%
Try	0	100%
Starred	4	missing

表 5 コメントありと予測された構文木の特徴ルートノード名

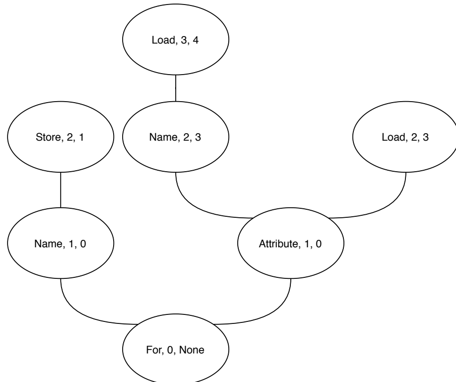

図 15 分類に失敗した For をルートノードに持つ構文木

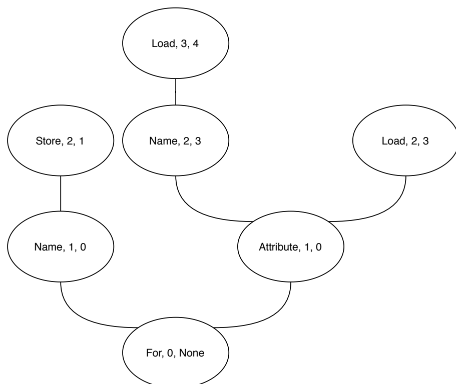

図 16 分類に成功した For をルートノードに持つ構文木

```
1005        broadcast_shape = [1] * ndims
1006        for dim in self.axis:
1007          broadcast_shape[dim] = input_shape.dims[dim].value
```

図 17 分類に失敗した For をルートノードに持つ構文木のもととなるソースコード

```
961        # Validate axes
962        for x in self.axis:
963          if x < 0 or x >= ndims:
964            raise ValueError('Invalid axis: %d' % x)
```

図 18 分類に成功した For をルートノードに持つ構文木のもととなるソースコード

5.2.2 ルートノード名について

特徴ルートノード名について正解データと予測データの比較、各ラベル同士の比較を通して学習後の特徴の変化について、特徴ルートノード名、構文木の深さ、分類成功率から分析を行う。

表 4、表 5 は予測ラベルごとの特徴ルートノード名を列挙しており、一方のラベルにのみ存在する特徴ルートノード名 (Feature_RootNodeName)、構文木の深さ (GraphDepth)、分類成功率 (AccuracyRate) を列挙した。データセットの分析に用いた特徴抽出の方法と同様に、コメントなし予測データのルートノード名とコメントあり予測データのルートノード名を比較し、一方のラベルに存在するルートノード名を特徴ルートノード名とする。

- コメントなし予測データ

  表 4、表 1 より、コメントなし正解データの特徴ルートノード名のほぼすべてがコメントなし予測データの特徴ルートノード名となっていることが分かる。しかし、ルートノード名 Starred は含まれておらずコメントあり予測データに分類された。Starred をルートに持つ構文木はデータセット内に 1 件のみである。本研究では Leave-One-Out 交差検証を用いて学習を行っているため、Starred をルートに持つ構文木を学習する際にはルートノード名以外から構文木の特徴を学習する。本研究ではノード名、ノードが存在する深さ、親ノードの Index を用いて構文木の特徴ベクトルを作成しているため、Starred は少なくとも学習データからルートノードの特徴は学習できないことが分かる。また、Starred をルートに持つ構文木の深さは 4 と「浅い構文木」であるにも関わらずコメントあり予測データに分類されたことから、深さだけの特徴も考慮した判断を行ったと考えられる。表 4 の特徴ルートノード名の中でも、arg、BinOp、ClassDef、Dict、Import、Lambda は分類成功率が 100%だった。これらの構文木はルートノード名からも特徴を学習していると言える。ただし、Lambda はデータセットに 1 件しか存在しないため、Starred と同様に、少なくともルートノード名から特徴を学習することができないため除外する。Lambda は深さが 4 であるためコメントなしと判別されたと考えられる。それ以外のルートノード名についても分類成功率は 80%を超えているが、異なるラベルに分類された構文木も存在する。構文木の深さはすべて「浅い構文木」であるためルートノード名、異なるラベルに分類された構文木は深さ、ルートノード名とは異なる特徴を学習していると考えられる。

  コメントなし正解データと比較し新たに加えられたカテゴリー名は Control flow である。データセットにおいて Control flow に分類される For は両ラベルに出現するルートノード名だった。学習後に For をルートに持つ構文木がコメントなしに分類されたため、表 4 の AccracyRate も低くなっている。For をルートノードに持つ構文木を図 15、図 16、ソースコードを図 17、図 18 に示す。図より、2 つの構文木がまったく同じ構造をしていることが分かる。しかし、図 15 の分類に失敗した構文木にはコメントが記述されており、図 16 の分類に成功した構文木には記述されていない。図

17、図 18 からも似た処理が行われていることが分かる。処理の構成上似ているにも関わらず一方にコメントが記述された理由として、本研究において設計した特徴ベクトルで表したソースコードの構造とコメント記述者から見たソースコードの構造のギャップが考えられる。コメントを記述するのは人間であり、多くの場合、コメントを記述する際には構文木ではなく処理全体の流れや処理内容からコメントの要否を判断する。ソースコードの構造からある程度コメントが記述されやすい処理を判別することは可能だが、コメントを記述の要否を人間が判断する際には異なる基準が存在する可能性が示唆された。

- コメントあり予測データ
  表 5、表 2 より、コメントあり正解データの特徴ルートノード名はすべてがコメントあり予測データの特徴ルートノード名となっていることが分かる。BoolOp と Try をルートノードに持つ構文木はコメントが記述されやすいと言える。コメントが記述されやすい構文木は「深い構文木」だが、これらの構文木は「浅い構文木」であるため、深さとは異なる有用な特徴となる可能性が示唆された。

6. 結論

6.1 まとめ

コメントの有無を判別するために設計した特徴から、構文木深さとルートノード名の分析と考察を行った。構文木の深さについて、深さ 0 から 4 の構文木にコメントが付きにくく、深さ 5 から 9 の構文木にコメントが付きやすいことが分かった。そのことから、深さによる複雑さの定量化の可能性が示唆された。しかし、そのような特徴を強く学習したために、コメントが付与されていない深い構文木や、コメントが付与された浅い構文木の判定が難しいという問題も見られた。

ルートノード名では、正解データと予測データの比較からコメントの有無についての特徴を分析した。正解データからは構文木とソースコードにコメント記述の可否の決定に関わる似た特徴が見られた。また予測データでは、arg、BinOp、ClassDef、Dict、Import、BoolOp、Try をルートに持つ構文木は分類成功率が 100%だった。とくに BoolOp と Try は「浅い構文木」であるため構文木の深さに影響されない特徴であることが分かる。しかし、For をルートに持つ構文木からは似た構造を持つ構文木であるにも関わらずコメントの有無が分かれるなど、本研究で設計した構文木の構造ではあらわせないコメントの可否の決定に関わる要素があることが示唆された。ルートノード名にコメントの有無を判別する傾向が見られたが、より細かく構造の特徴を分析するために子ノードの情報も含めた集計を行う必要がある。

6.2 今後の課題

本研究では Tensorflow のソースコードファイルのみで実験を行ったが、プロジェクトによってコメントの記述ルールが異なる場合があるため、今後は異なるプロジェクトのソースコードファイルも加え、本研究で見られた傾向と比較し分析を行う必要がある。また、ノード名の分析はルートノード名のみでは不十分であり、構造の分析のためにさらにスコープを広げ親ノードと子ノードの関係を分析する必要がある。本研究で設計した特徴ベクトルのから親ノードの Index を用いることで集計を行うことが可能である。しかし、特徴ルートノード名 For の構文木とソースコードから、構造のみからは得られない特徴の存在が示唆された。コメントを記述するのは人間であるため、コメント記述の可否の決定に関する要素を構文木の構造に盛り込むことが可能か検討を行う必要もある。

参考文献

[1] Dustin Boswell、Trevor Foucher(2012), ”リーダブルコード——より良いコードを書くためのシンプルで実践的なテクニック”，オライリー・ジャパン
[2] スティーブ マコネル, Steve McConnell (2005), クイープ, ”CODE COMPLETE 第 2 版 下 完全なプログラミングを目指して”, 日経 BP
[3] 阿萬 裕久 (2011) ”オープンソースソフトウェアにおけるコメント記述およびコメントアウトとフォールト潜在との関係に関する定量分析” 情報処理学会論文誌 Vol.53 pp.612–621
[4] 小田 悠介 (2014) ”ソースコード構文木からの統計的自動コメント生成” 情報処理学会研究報告
[5] 前橋 和弥 (2009), ”プログラミング言語を作る”, 技術評論社
[6] Meet the Nods (最終閲覧日：2020 年 1 月 22 日), https://greentreesnakes.readthedocs.io/en/latest/nodes.html
[7] Python 言語リファレンス (最終閲覧日：2020 年 1 月 22 日), https://docs.python.org/ja/3/reference/index.html
[8] 本橋智光　著, 株式会社ホクソエム　監修, ”前処理大全［データ分析のための SQL/R/Python 実践テクニック］”, 技術評論社