コード LLM のマスク予測と検索エンジンを活用したネーミングバグ自動検出の試み

徳毛 悠人　佐伯 昌造　阿萬 裕久　川原 稔

変数名の適切さはソースコードの可読性に影響を与えるため，変数の名前は変数の役割や意味を表した命名であることが一般には望ましいとされている．変数名がその役割や意味を適切に表せていない状態をネーミングバグといい，その適切さの評価は人間によるコードレビューによって行われる．そこで本論文では，大規模言語モデルと検索エンジンを活用し，ネーミングバグに該当するかを自動的に判定する手法を提案している．具体的には，変数の名前を隠した関数単位のプログラムを作り出し，大規模言語モデルによって元の名前を推定できるかでもって，その名前がネーミングバグに該当するかを判定するというものである．加えて，検索エンジンの検索結果として得られた変数名の候補に元の名前が挙がるかでもって再評価をする．そして，評価実験を通じて提案手法の有効性について検討している．

1　はじめに

プログラムにおいて，変数はデータやオブジェクトを格納・参照するために欠かせない存在である．開発者は変数に名前を割り当てるが，その名前は変数の役割を反映していることが望ましい [1] [16] 適切な名前が与えられた場合，開発者だけでなく第三者にも読みやすく理解しやすいコードとなる．逆に，役割とは関係のない名前やわかりにくい名前が与えられた場合，それによってコードが理解しにくいものになったり誤解を招きやすいものになったりする恐れがある [2]．これをネーミングバグいう [6] [10]．

それゆえ，“変数名の適切さ” は重要であるが，その評価を客観的かつ機械的に行うことは容易でない．変数の命名についてはコーディング規約等でガイドラインが与えられることがあるが，その内容は “ローカル変数には短く意味のある名前をつけること [4]” や “グローバル変数には分かりやすい名前をつけること [7]” といった定性的で具体性に欠ける内容であることが多く，自動的に判定できるものではない．このため，変数名の適切さを人手で確認するしかないのが現状である．また，開発の進行・保守の過程で変数の名前と役割の間で乖離が生じる場合もある．したがって，コードの作成時のみならず，変更時にも変数名の適切さ確認が必要であり，この作業は重要であると同時に開発者やレビューアの負担にもなっている [14]．

そこで本論文では，変数名の適切さを自動評価するための手法を提案する．具体的には，Python プログラムにおける変数に着目し，大規模言語モデルのマスク予測を活用した評価を行う．ただし，この手法では一般によく使われるような名前が候補として出やすいため，必ずしも開発対象ソフトウェアのプロジェクトやドメインに合った名前の評価にそのまま使えるとは限らない．そのため，コード検索技術にも着目し，当該変数が使われている関数と構造が類似した関数を検索して，そこで使われている変数名も活用する．そして，オープンソースソフトウェアから収集された変数名データを用いて提案手法の有効性を確認する．

An Automated Dection of Naming Bugs using Code LLM Mask Prediction and Search Engine.

Yuto Tokumo, 愛媛大学大学院理工学研究科, Graduate School of Science and Engineering, Ehime University.

Shozo Saeki, Hirohisa Aman, and Minoru Kawahara, 愛媛大学総合情報メディアセンター, Center for Information Technology, Ehime University.

2 先行研究

本節では，メトリクスとランダムフォレスト，並びに大規模言語モデルを活用した変数名自動評価に関する先行研究を紹介する．

2.1 メトリクスとランダムフォレストを利用した名前変更予測

変数名の適切さ評価は人手に頼るところが多く，必ずしも容易ではない．そうした負担の軽減に向け，Zhang ら [17] は識別子に対して名前変更が必要かどうかを予測する手法を提案している．具体的には，静的コード解析を用いてさまざまなメトリクス測定を行い，そこで得られたメトリクスデータ（特徴データ）を説明変数，名前変更の有無を目的変数としたランダムフォレストモデルを構築し，変更予測を行うという手法を提案している．本論文で注目しているのは変数名の適切さに対する自動評価であるが，Zhang らの手法では名前変更が必要か否かの予測（判定）が行われ，その判定は適切さ評価とみなすことができることから重要な先行研究であるといえる．

2.2 大規模言語モデルを利用した変数名の自動評価

上述したメトリクスとランダムフォレストを用いた名前評価とは別のアプローチとして，コード補完技術がある．この技術は，コーディングの際にそれまでに入力したソースコード片から次に記述が必要な変数やメソッド呼出し等を推定して候補を自動生成するというものである．コード補完技術の研究は，Mandelin らによるツール Prospector の開発に始まり，それ以降さまざまなアプローチが提案されてきた [3] [5] [8] [15]．これには型補完やコードの局所性を活用する手法等が含まれ，これらを利用することでコードの自動生成や予測の精度向上を実現させている．

近年では，大規模言語モデルを活用したアプローチも見られるようになってきた．例えば，Mastropaolo ら [9] は大規模言語モデルを活用した変数名推薦の効果を調査しており，その有効性が報告されている．さらに，言語モデルでは一部のトークンをマスクしておいて，そこに入る適切なトークンを予測する “マスク予測” というタスクが可能であることから，Mori ら [11] [12] [13] はマスクされた元の変数名がマスク予測の結果でも候補として挙がってくるかどうかでもって不適切な変数名（ネーミングバグ）を検出するという手法を提案している．

この手法にはいくつか制限がある．Mori らはこの手法によって Java 言語で記述されたソースコードにおけるローカル変数名の適切さ評価を行っていたが，マスク予測の結果の扱いについては単に “候補に挙がっていたか否か” だけに注目していた．大規模言語モデルを用いたマスク予測では複数の名前が候補として提示されるが，Mori らは何番目の候補として目的の名前が提示されたかという情報やそこで一緒に出力されるスコアには注目していなかった．加えて，Mori らは変数の型情報も活用するという工夫 [12] でもって大規模言語モデルによる判定に調整を施していたが，それは Java 言語で書かれたソースコードであったがゆえに利用可能な情報であったため，Python を対象とした場合には同様の調整が不可能という課題もあった．

これらの課題点に対処するため，本研究では大規模言語モデルのマスク予測機能を用いた変数名評価法の改善を試みる．さらには，Python では型情報が使えないことから，これに代わってコード検索技術を活用することもあわせて提案する．

3 提案手法

本節では大規模言語モデルとコード検索技術を活用した，変数名の評価法を提案する．なお，大規模言語モデルとしては CodeBERT を，コード検索技術としては Apache Lucene をそれぞれ利用する．

3.1 CodeBERT を活用した変数名評価

2.2 節で述べたように，本論文では大規模言語モデルのマスク予測によって得られた出力のスコアを活用することで課題点を改善することを提案する．以下に手順を示す．

（1） 変数のマスク処理

CodeBERT を用いたマスク予測を行うため，マ

スク予測の対象となるローカル変数を “<mask>” という特殊トークンに置き換える．具体的には，Python の標準ライブラリである tokenize モジュールを用いることで，対象とするローカル変数の開始位置と終了位置を取得し，置き換える．

（2） CodeBERT を用いたマスク予測
手順（1）でマスク処理を施した関数を入力テキストとして，CodeBERT のマスク予測を行う．この結果として出力されたトークンとスコアを収集し，各トークンに対するスコアの合計を求める．なお，推定されたトークンに空白文字が混在している場合があるため，その場合は空白を除去する．そして，スコアの合計値を降順に並べ，上位 5 つのトークンをマスク予測の結果として使用する．

（3） 適切さの評価
手順（2）で得られた上位 5 件の結果を使用し，それぞれの予測結果と特殊トークンに置き換える前の変数名が完全一致する場合は適切と判定し，そうでない場合はネーミングバグと判定する．

3.2 類似コード検索を活用した変数名評価

2.2 節で述べたように，本論文ではコード検索技術を活用することで課題点を改善することを提案する．以下に手順を示す．

4 評価実験

本章では提案手法の有効性を確認するために行った評価実験について報告する．

4.1 目的

本実験の目的は，提案手法による変数名評価の有効性を実際のデータを用いて示すことである．そのため，次の 2 つの研究課題（Research Question: RQ）を立て，評価実験を通じてこれらに回答していくことにする：

RQ1: 大規模言語モデルのマスク予測を用いた変数名評価（提案手法の第 1 段階）によって，Python ソフトウェアにおける変数名の適切さ・不適切さをどの程度正しく評価できるか？

RQ2: 大規模言語モデルのマスク予測とともに類似コード検索も活用した変数名評価（提案手法の第 2 段階）によって，評価精度はどの程度改善されるか？

RQ1 は，提案手法の基盤となる，大規模言語モデルの有効性に関する基本的な問いである．提案手法では，ソースコードの文脈を捉えることで適切な変数名を大規模言語モデルが提示してくれることを期待している．それゆえまずは，大規模言語モデルのみでどの程度正しく変数名評価が可能なのかを明らかにしようとしている．あわせて，大規模言語モデルだけではどういった変数名はうまく評価できないのかという点についても考察していく．

RQ2 は，本研究で新たに着目したコード検索技術の有効性に関する問いである．大規模言語モデルで予測（推薦）される変数名は，広く一般に使われるような名前が主体であると想定される．それゆえ，開発対象ソフトウェアのプロジェクトやドメインに特化した名前が十分にサポートされるかどうかは疑問であり，そのために構造が類似したコードを検索して使用することで評価精度向上が期待できるのではないかと筆者は考えている．この点について明らかにするのが RQ2 の狙いである．

4.2 対象

本実験では，Python で書かれたオープンソースソフトウェアで使われている変数を実験対象として使用する．対象のソフトウェアとしては，多くの開発者によってソースコードが開発・保守され続けているという理由から，表 1 に示す 5 つのライブラリソフトウェアを選定した．いずれも Python でのソフトウェア開発で広く使用されている著名なライブラリとなっている．

これらのソースファイルについて，それぞれの変更履歴を調べ，最後の更新から最も時間が経過している 10 個，即ち，最も長くそのままのかたちを維持し続けている 10 個を各ソフトウェアから抽出した．ソースファイルの一覧については，付録 A を参照さ

【注目変数】value

【コード断片】

```
1  def format_header_param_html5(name, value):
2      if isinstance(value, six.binary_type):
3          value = value.decode('utf-8')
4      value = _replace_multiple(value, _HTML5_REPLACEMENTS)
5      return u'%s="%s"' % (name, value)
```

図 1　“適切” な変数名のデータ例

【注目変数】defects

【コード断片】

```
1  def format_header_param_html5(name, defects):
2      if isinstance(defects, six.binary_type):
3          defects = defects.decode('utf-8')
4      defects = _replace_multiple(defects, _HTML5_REPLACEMENTS)
5      return u'%s="%s"' % (name, defects)
```

図 2　“不適切” な変数名のデータ例

表 1　実験対象のソフトウェア

ライブラリ名
numpy
pandas
pip
scikit-learn
scipy

れたい．

これらのソースファイルは，いずれも著名なライブラリソフトウェアにおいて長期間に渡って変更されることなくそのままのかたちで運用されていることから，その中に不適切な名前の変数が含まれている可能性は低いと考えられる．そこで，これらの中に登場するローカル変数に着目し，それらを “適切な変数名” のデータとして使用することとした．結果として，630 個の変数名とそれが使われているコード断片（関数）のペア（例えば図 1）をデータセットとして使用することにした．

一方，変数名評価を行う上では “不適切な変数名”（ネーミングバグ） のデータも必要であるため，上述の “適切な変数名” データをあえてランダムにシャッフルしたものを考える．つまり，あるローカル変数が使われているコード断片（関数）について，あえてその名前を別のローカル変数にランダムに置き換えたもの（例えば図 2）を用意し，そちらを “不適切な変数名” のデータと見なすことにする．それゆえ，こちらについても 630 個の変数名とコード断片のペアをデータセットに加えることとした．

以上のかたちで，適切な変数名と不適切な変数名をあわせて，合計 1260 個の変数データセットを実験対象として用意した．

4.3　手順

本実験の手順を以下に示す．

（1） 変数のマスク処理
　データセットに含まれるすべての関数について，評価対象の変数名を “<mask>” という特殊トークンに置き換える．

（2） 大規模言語モデルによるマスク予測
　3.1 節に示した手順により，データセットに含まれるすべてのソースコード（マスク処理済み

のもの）について変数名のマスク予測を行い，元の（マスク処理前の）名前が適切であったか否かを判定する．ただし，その際には，スコア 1 位のみ，2 位以内，3 位以内，4 位以内，及び 5 位以内のそれぞれについて判定精度を求める．

(3) 類似コード検索を活用した評価

手順（2） で ネーミングバグ と判定された変数に関しては，3.2 節に示した手順により，“適切” な名前として再調整可能かどうかを調べる．そして，手順（2） と同様に判定精度を求める．

4.4 結果

本節では，上述の手順で行った実験の結果を示す．

4.4.1 大規模言語モデルによるマスク予測

手順（2） に従い，マスク処理を施したソースコードを入力として，大規模言語モデルによる変数名のマスク予測を行った．判定結果を表 2 に示す．

表 2 に示した結果に基づいて，適切な名前をどの程度正確に予測できていたかを評価するため，適合率と再現率，及び F 値を求めた．その結果を表 3 に示す．

表 3 から分かるように，閾値を下げることで適合率はやや低下しているがそれでも 0.96 を上回る値を維持できており，大規模言語モデルのマスク予測でもって提示される名前と一致していれば，その名前は実際に適切な変数名となっていたことが分かる．一方，再現率は 0.208 ～ 0.349 となっており，適合率に比べて大幅に低い水準となっていた．即ち，実際には適切な名前であったと思われる多くの変数名が，大規模言語モデルによるマスク予測ではネーミングバグと見なされてしまっていた．

表 2　5 位以内の名前を “適切” とした場合の判定結果

		予測		
		適切	不適切	合計
正解ラベル	適切	220	410	630
	不適切	7	623	630
	合計	227	1033	1260

表 3　“適切” な名前に関する判定精度

スコアの閾値	適合率	再現率	F 値
1 位	0.985	0.208	0.343
2 位	0.983	0.271	0.425
3 位	0.984	0.295	0.454
4 位	0.967	0.324	0.485
5 位	0.969	0.349	0.513

表 4　“不適切” な名前に関する判定精度

スコアの閾値	適合率	再現率	F 値
1 位	0.557	0.997	0.715
2 位	0.577	0.995	0.731
3 位	0.585	0.995	0.737
4 位	0.594	0.989	0.742
5 位	0.603	0.989	0.749

次に，不適切な変数名が正しくネーミングバグと判定されるかを確認するため，同様に適合率，再現率，及び F 値を算出した．結果を表 4 に示す．

こちらについては “適切な” 変数名の検出とは真逆の傾向にあり，閾値に関わらず高い再現率（0.989 ～ 0.997）を示す一方，適合率は 0.557 ～ 0.603 に留まっていた．つまり，大規模言語モデルのマスク予測による判定では，ほとんどの “不適切な” 変数名を実際に ネーミングバグ と見なすことに成功していたが，代わりに ネーミングバグ と判定された変数の約半数は実際には “適切な” 変数名であったことが見て取れる．

4.4.2 類似コード検索を活用した評価

手順（3）に従い，大規模言語モデルのマスク予測で ネーミングバグと判定された変数に対して，それに対応するコード断片（関数）と類似したコード断片を検索し，上位 10 位以内に入ったコード断片と紐づけられた変数名を “適切な名前” の候補として加えるという “2 段階目の” 判定を行った．その際には，類似コード検索の範囲を同じ開発プロジェクト内に限定した場合と同一プロジェクトに限定しない場合の 2 種類を考え，それぞれのかたちで評価を行った．

検索の範囲を同一プロジェクトに限定した場合の結果を表 5 に示す．なお，表中の括弧内の数値は，同一プロジェクト内での類似コード検索を行う前からの変化量を表す．あわせて，検索の範囲を同一プロジェクトに限定しなかった場合の結果を表 6 に示す．

表 5，6 に示した結果に基づいて，適切な名前をどの程度正確に予測できていたかを評価するため，適合

表 5　5 位以内の名前を “適切” と見なしたときの判定結果：同一プロジェクト内での類似コード検索も併用

		予測		
		適切	不適切	合計
正解ラベル	適切	356(+136)	274(−136)	630
	不適切	10(+3)	620(−3)	630
	合計	366(+139)	894(−139)	1260

表 6　5 位以内の名前を “適切” と見なしたときの判定結果：同一プロジェクトに限定しない類似コード検索も併用

		予測		
		適切	不適切	合計
正解ラベル	適切	352(+132)	278(−132)	630
	不適切	9(+2)	621(−2)	630
	合計	361(+134)	899(−134)	1260

表 7　“適切” な名前に関する判定精度：同一プロジェクト内での類似コード検索も併用

スコアの閾値	適合率	再現率	F 値
1 位	0.984	0.484	0.649
2 位	0.982	0.522	0.682
3 位	0.983	0.537	0.694
4 位	0.972	0.552	0.704
5 位	0.973	0.565	0.715

表 8　“適切” な名前に関する判定精度：同一プロジェクトに限定しない類似コード検索も併用

スコアの閾値	適合率	再現率	F 値
1 位	0.987	0.476	0.642
2 位	0.985	0.516	0.677
3 位	0.985	0.530	0.689
4 位	0.975	0.546	0.700
5 位	0.975	0.559	0.710

率と再現率，及び F 値を求めた．その結果を表 7 及び 8 に示す．

これら 2 つの表から分かるように，類似コード検索の範囲を同一のプロジェクト内に限定した場合と限定しない場合で特に違いは見られなかった．そして，適合率に関しては，大規模言語モデルのみを用いていた場合は 0.969 ∼ 0.985 であった（表 3）のに対し，類似コード検索を活用することで 0.975 ∼ 0.987 と少しだけ向上することが分かった．再現率についても，大規模言語モデルのみを用いていた場合は 0.208 ∼ 0.349 に留まっていたが，類似コード検索を活用することで 0.476 ∼ 0.565 まで向上することとなった．つまり，類似コード検索を活用するという第 2 段階の判定を導入することで，大規模言語モデルでの弱点であった再現率の低さを緩和することに成功していた．

次に，不適切な変数名が正しくネーミングバグと判定されるかを確認するため，同様に適合率，再現率，及び F 値を算出した．結果を表 9 及び 10 に示す．

こちらについても，類似コード検索の範囲を同一プロジェクト限定するか否かの違いは特に見られなかった．そして，適合率に関しては，大規模言語モデルのみの場合は　0.557 ∼ 0.603 であったが，類似コード検索を活用することで 0.65 ∼ 0.69 にまで改善した．一方，再現率については，元々非常に高い水準（0.98 ∼ 0.99）にあったため，特に違いは見られなかった．

4.5　考察

本節では，各 RQ に対応させるかたちで実験結果に対する考察を述べる．

4.5.1　RQ1

4.4.1 節で示したように，大規模言語モデルによるマスク予測を用いたところ，適切さ評価の最も良い（F 値で判定）結果としては，適合率 0.969，再現率 0.349，F 値 0.513 となった．即ち，大規模言語モデ

表 9　“不適切” な名前に関する判定精度：同一プロジェクト内での類似コード検索も併用

スコアの閾値	適合率	再現率	F 値
1 位	0.658	0.992	0.791
2 位	0.674	0.990	0.803
3 位	0.681	0.990	0.807
4 位	0.687	0.984	0.809
5 位	0.694	0.984	0.813

表 10　“不適切” な名前に関する判定精度：同一プロジェクトに限定しない類似コード検索も併用

スコアの閾値	適合率	再現率	F 値
1 位	0.655	0.994	0.789
2 位	0.672	0.992	0.801
3 位	0.679	0.992	0.806
4 位	0.685	0.986	0.808
5 位	0.691	0.986	0.812

ルでもって名前が “適切” と判定された場合には，約 97% の確率で実際に適切な名前であったことがうかがえる．一般に，大規模言語モデルは事前に大量のソースコードで学習が行われており，そのうえでマスク予測タスクでは，入力に対して確率的に最も尤もらしい変数名を出力する仕組みになっている．つまり，多数のソースコードでよく使われるような変数名（“明らかに適切である” と思われる変数名）が提示されやすく，それゆえ実際のソースコードでもその名前が使われていた場合には，“適切な” 変数名と見なして問題なかったものと思われる．一方，再現率は 0.349 と適合率に比べて大幅に低い水準になっていた．つまり，実際に実験で使用した “適切” な変数名の約 35% しか大規模言語モデルでは “適切” と判定されなかった．このことから，大規模言語モデルによるマスク予測では，頻出する変数名については正しく “適切” と判定できる可能性が高いものの，出現頻度が必ずしも高くないような名前の場合は判定が難しいのではないかと推察される．

同様に，不適切さ評価の最も良い結果としては，適合率 0.603，再現率 0.989， F 値 0.749 となった．適切さ評価とは逆の傾向にあり，適合率はあまり高くないものの，再現率はほぼ 99% となっていた．つまり，実際に不適切な変数名については，そのほとんどすべてを大規模言語モデルではネーミングバグとして判定できていた．しかしながら，適合率がさほど高くないことから，多くの変数を一律的にネーミングバグと判定してしまっていたのではないかと考えられる．換言すれば，“適切” と判定する基準が厳しすぎであり，結果として多くの変数名をまとめてネーミングバグと見なしてしまっていたことが懸念される．それでも，適切率は 50% を超えており，やみくもにネーミングバグ と判定してしまっているわけではないことから，大規模言語モデルのマスク予測機能による判定を否定するものでもない．

以上より，RQ 1 に対しては以下の通り回答する：**RQ1：大規模言語モデルのマスク予測を用いた変数名評価（提案手法の第 1 段階）によって，Python ソフトウェアにおける変数名の適切さ・不適切さをどの程度正しく評価できるか？**

大規模言語モデルが適切な名前であると判定した場合の信頼性は高く，その適合率は約 0.97 である．一方，大規模言語モデルだけでは多くの変数をまだネーミングバグと判定してしまっており，いわばかなり厳しい基準でもって判定している可能性が高い．実際，不適切な名前に関しては，再現率が約 99% なのに対して，適合率は 60% 程度に留まっていた．

4.5.2 RQ2

大規模言語モデルのマスク予測でネーミングバグと判定されてしまった変数に対し，2 段階目の判定として，類似コード検索を活用した変数名評価を行った．その結果，大規模言語モデルによるマスク予測だけでは 410 個の変数がネーミングバグと判定されていた（表 2）が，類似したコードを検索して，そこでの変数名を適切な変数名の候補に追加することで 136 個の変数を “適切” な名前であると再評価することができた．

これにより，適切な名前の評価に関しては，適合率は元々高い水準にあったことからほぼ変わらないものの，再現率は約 0.22 ポイント（約 62%）の向上が見られた．そのため，総合評価値である F 値も約 0.20 ポイント（約 39%）の改善となった．不適切な名前の評価に関しては，適切な名前の評価とは逆で，再現率が元々高い水準にあったことから特に変化は見られなかったが，適合率は約 0.09 ポイント（約 15%）の向上となり，F 値も約 0.06 ポイント（約 8%）の改善となった．

これらの結果から，大規模言語モデルのマスク予測を用いた変数名評価を行いつつ，そこで ネーミングバグと判定されてしまった場合には，類似コード検索技術も活用して同じような構造のプログラムで使われていた変数名も適切な名前の候補として追加することで大規模言語モデルでの弱点，即ち，判定基準が厳しすぎるという欠点を緩和できるものと思われる．この理由として，大規模言語モデルと類似コード検索での注目するポイントの違いが挙げられる．大規模言語モデルでは，大量のソースコードでもって学習されているがゆえに，いわば最も “普通な” 名前を提示する傾向にあるといえる．一方，類似コード検索の場合は，当該変数が使われているコード断片と構

造がよく似たものだけに注目しているため，広く一般に使われやすい名前というよりも，実際に似た処理で使われていた名前が候補に入ることになるため，結果としてうまく適切な名前を再評価できていたのではないかと考えられる．

2 つの技術を組み合わせることで多くの変数名の評価をより高い精度で行うことができたが，それでもなお，適切な変数名の評価における再現率は決して高くなく，一定数の（本来は）適切な変数名をネーミングバグと誤って判定してしまっている．そのような変数名を調べたところ，複合語やアンダーバーが使われた名前（例えば，`header_parts`，`__all__`）が比較的多く見られた．本実験では，実際の変数名と候補となる名前との完全一致でもって名前の適切さを評価しているため，この点についてさらなる工夫を検討する必要があると考えられる．

以上より，RQ 2 に対しては以下の通り回答する：

RQ2：大規模言語モデルのマスク予測とともに類似コード検索も活用した変数名評価（提案手法の第 2 段階）によって，評価精度はどの程度改善されるか？

大規模言語モデルでネーミングバグと判定された変数に対して，構造の似ているプログラムを類似コード検索で調べて活用することにより，その判定精度を改善できることを確認した．具体的には，適切な名前を正しく “適切” と判定できるか？については， F 値を約 39%改善できた．同様に，不適切な名前を正しくネーミングバグ と判定できるか？については，F 値を約 8% 改善できた．ただし，複合語やアンダースコアを使った名前に対しては，適切な評価が行えておらず，それらに対する工夫について今後検討していく必要があると考えられる．

5 まとめと今後の課題

一般に変数名の適切さはソースコードの可読性に大きな影響を与えるが，その評価は人手によるコードレビューが主流であり，そのコストが問題となっている．近年では，この問題を解決するため，メトリクスとランダムフォレストを用いた評価手法や大規模言語モデルのマスク予測を活用した評価手法が提案されてきた．しかしながら，これらの手法にはそれぞれ課題点もあり，改善の必要があった．そこで，本論文ではそれらの課題を解決するため，変数名の自動評価を次の 2 段階で行う手法を提案した：

（1） ソースコード向けの大規模言語モデルである CodeBERT を活用し，それに対して評価対象の変数名を含むコード断片を与える．ただし，評価対象の変数名部分は前処理としてマスクしておく．そして，CodeBERT にマスク部分に入る名前を予測させる．

この手法自体は先行研究で既に提案されていたが，先行研究では予測結果として得られる複数の変数名候補に対して優先順位付けは行っておらず，その点で評価が粗いものとなっていた．そこで本論文では，CodeBERT が出力する確信度を活用し，この点の改善を試みた．

（2） CodeBERT のマスク予測機能だけでは十分な評価が行えないことが既に知られており，その課題点を克服するために本論文では類似コード検索技術を活用することを提案した．具体的には，CodeBERT によって ネーミングバグと判定されてしまった場合には，追加の措置として，評価対象の変数が使われているコード断片と構造が類似したコード断片を検索して，類似コードで使われていた変数名も適切な名前の候補に加えるという手法を提案した．

提案手法の有用性を確認するため，著名な Python ソフトウェアで使用されていた変数名を実験対象として評価実験を行った．その結果，CodeBERT 単体での変数名評価に比べ，類似コード検索技術も併用することで，適切な変数名及びネーミングバグの検出精度を向上させることに成功した：前者の F 値を約 39%，後者の F 値を約 8% だけ改善させることができた．以上より，本論文で提案した手法により，変数名の自動評価性能を向上させることができたと考える．

今後の課題として，現状ではうまく評価できない名前（複合語やアンダースコアを使った名前）に対する対応策の検討や他の AI 技術との比較が挙げられる．

謝辞 本研究は JSPS 科研費 25K15059，25K15062，26K15121 の助成を受けたものです．

参 考 文 献

[1] Boswell, D. and Foucher, T.: *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*, O'Reilly Media, Sebastopol, CA, 2011.

[2] Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., and Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques, *Empir. Softw. Eng.*, Vol. 19(2014), pp. 1040–1074.

[3] Chatterjee, S., Juvekar, S., and Sen, K.: SNIFF: A Search Engine for Java Using Free-Form Queries, *Fundamental App. Softw. Eng.*, Chechik, M. and Wirsing, M.(eds.), Lecture Notes in Computer Sc., Vol. 5503, Springer, March 2009, pp. 385–400.

[4] Gosling, J., Joy, B., Steele, G., and Bracha, G.: *Java* 言語仕様, 桐原書店, 東京, 3 edition, 2006.

[5] Gvero, T., Kuncak, V., Kuraj, I., and Piskac, R.: Complete completion using types and weights, *ACM SIGPLAN Notices*, Vol. 48, No. 6(2013), pp. 27–38.

[6] Høst, E. W. and Østvold, B. M.: Debugging Method Names, *Proceedings of the 23rd European Conference on ECOOP*, Genoa, Berlin, Heidelberg, Springer-Verlag, July 2009, pp. 294–317.

[7] Kernighan, B. W. and Pike, R.: プログラミング作法, アスキー, 東京, 2000.

[8] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D.: Jungloid Mining: Helping to Navigate the API Jungle, *ACM SIGPLAN Notices*, Vol. 40, No. 6(2005), pp. 88–61.

[9] Mastropaolo, A., Aghajani, E., Pascarella, L., and Bavota, G.: Automated variable renaming: are we there yet?, *Empir. Softw. Eng.*, Vol. 28(2023), pp. 45:1–45:26.

[10] 峯久朋也, 阿萬裕久, 川原稔: Transformer によるメソッド名推定を活用したネーミングバグの検出, コンピュータソフトウェア, Vol. 39, No. 4(2022), pp. 17–23.

[11] 森哉尋, 阿萬裕久, 川原稔: 変数名の自動評価に向けた名前のゆらぎに関する調査, ソフトウェア工学の基礎 *30*, (2023), pp. 171–172.

[12] 森哉尋, 阿萬裕久, 川原稔: CodeT5 と正規表現を活用した不適切な変数名の自動検出法とその評価, ソフトウェア工学の基礎 *31*, (2024), pp. 35–44.

[13] Mori, Y., Aman, H., Amasaki, S., Yokogawa, T., and Kawahara, M.: An Application of Program Slicing and CodeBERT to Distill Variables With Inappropriate Names, *Proc. IEEE/ACIS 22nd Int'l Conf. Softw. Eng. Research, Management & App.*, May 2024, pp. 356–361.

[14] Murphy-Hill, E., Parnin, C., and Black, A. P.: How We Refactor, and How We Know It, *IEEE Trans. Softw. Eng.*, Vol. 38, No. 1(2011), pp. 5–18.

[15] Tu, Z., Su, Z., and Devanbu, P.: On the Localness of Software, *Proc. 22nd ACM SIGSOFT Int'l Symp. Foundations of Softw. Eng.*, Vol. 2014, November 2014, pp. 269–280.

[16] Wake, W. C.: *Refactoring Workbook*, Addison-Wesley Professional, Boston, 2004.

[17] Zhang, J., Luo, J., Liang, J., Gong, L., and Huang, Z.: An Accurate Identifier Renaming Prediction and Suggestion Approach, *ACM Trans. Softw. Eng. & Methodol.*, Vol. 32, No. 6(2023), pp. Article 148.

A 付録: ソースファイルの一覧

本論文の実験で使用したソースファイルの一覧を表 11 に示す.

表 11 データ収集源別のファイル一覧

numpy
benchmarks/benchmarks/bench_import.py
doc/source/f2py/code/setup_skbuild.py
numpy/_utils/_convertions.py
numpy/_core/cversions.py
benchmarks/benchmarks/bench_overrides.py
numpy/f2py/_isocbind.py
numpy/dtypes.py
numpy/_build_utils/gcc_build_bitness.py
numpy/core/_utils.py
doc/postprocess.py
pandas
asv_bench/benchmarks/boolean.py
asv_bench/benchmarks/finalize.py
asv_bench/benchmarks/timedelta.py
pandas/_config/display.py
pandas/core/computation/api.py
scripts/run_vulture.py
asv_bench/benchmarks/plotting.py
asv_bench/benchmarks/tslibs/normalize.py
pandas/core/reshape/api.py
pandas/io/formats/_color_data.py
pip
src/pip/_vendor/msgpack/exceptions.py
src/pip/_vendor/urllib3/contrib/_appengine_environ.py
src/pip/_vendor/urllib3/fields.py
src/pip/_vendor/urllib3/filepost.py
src/pip/_vendor/urllib3/packages/backports/makefile.py
src/pip/_vendor/urllib3/util/queue.py
src/pip/_vendor/urllib3/util/response.py
src/pip/_internal/cli/status_codes.py
src/pip/_vendor/urllib3/exceptions.py
src/pip/_internal/utils/_log.py
scikit-learn
sklearn/externals/_packaging/_structures.py
asv_benchmarks/benchmarks/manifold.py
asv_benchmarks/benchmarks/metrics.py
asv_benchmarks/benchmarks/model_selection.py
asv_benchmarks/benchmarks/neighbors.py
asv_benchmarks/benchmarks/svm.py
asv_benchmarks/benchmarks/utils.py
sklearn/externals/_arff.py
benchmarks/bench_plot_hierarchical.py
asv_benchmarks/benchmarks/cluster.py
scipy
doc/source/tutorial/stats/plots/kde_plot4.py
doc/source/tutorial/stats/plots/kde_plot2.py
doc/source/tutorial/stats/plots/kde_plot5.py
scipy/fft/_debug_backends.py
scipy/spatial/transform/_rotation_groups.py
scipy/optimize/_trustregion_krylov.py
scipy/integrate/_ivp/dop853_coefficients.py
scipy/optimize/_trustregion_ncg.py
benchmarks/benchmarks/ndimage_interpolation.py
benchmarks/benchmarks/cluster_hierarchy_disjoint_set.py