一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

OSSプロジェクトにおける既存ライブラリ機能の再実装の実態調査

小村　太一†　　クラ ラウラ ガイコビナ†　　肥後　芳樹†

† 大阪大学大学院情報科学研究科 〒565–0871 大阪府吹田市山田丘 1–5
E-mail: †{ta1kmr,raula-k,higo}@ist.osaka-u.ac.jp

あらまし　大規模化が進む OSS では，外部ライブラリで提供される機能を利用すべき場面でも，依存関係の追加を回避する等を目的として自前の実装を行う「再実装」が発生する．本研究では，先行研究が対象とした「ライブラリの関数の呼び出しから自前の実装に置き換えた場合」の再実装ではなく，「ライブラリの関数の実装をコピーアンドペーストした場合」の再実装に着目した．コードクローン検出に基づく調査を行うことで，先行研究では捉えきれなかった再実装の実態を解明した．先行研究と本研究における再実装の性質の差異を明らかにするとともに，GitHub 上のスター数上位 10 件のライブラリとアプリケーションに発生している再実装の数と性質を明らかにした．
キーワード　既存ライブラリ機能の利用，コードクローン，再実装

1. ま え が き

大規模化が進む OSS では，必要な機能が外部ライブラリで提供されている事実の認識不足等の理由から，既存機能の利用が適切な場面で，自前の実装を行う「再実装」が発生しうる [1]．再実装することで，新たな依存関係の導入を回避し，プロジェクトの独立性を保てるが，実装や保守のコストが増大したり，レビューの不足によって信頼性が低下したりする等の課題が生じる [2]．

このような課題がありながらも再実装は発生しているため，その発生件数と発生理由が調査された [2]．この先行研究においては，再実装は「利用していた外部ライブラリを自前の実装に置き換えること」と定義されたため，ライブラリ機能が自前の実装に置き換えられたコミットが調査された．その結果，45 個の Python プロジェクトにおいて計 48 個の再実装された関数を発見した．しかし，置き換えではなく，ライブラリの関数をコピーアンドペーストした場合については先行研究の手法では検出できない．

したがって，本研究では，先行研究では再実装と見なさなかった「外部ライブラリの関数をコピーアンドペーストすること」を再実装と定義し，その発生数や性質を調査することで，先行研究では捉えきれなかった再実装の実態を明らかにすることを目的とした．ただし，コピーアンドペーストした後に変更を加えた場合も調査の対象とする．また，本研究では関数とメソッドを区別せず，いずれも関数と呼ぶ．

本研究における再実装の定義から，コードクローン検出ツールを利用して再実装の検出を行った．コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことである [3]．コードクローンの関係にあるコード片の組をクローンペアと呼ぶ．コピーアンドペーストによる再実装を行ったアプリケーションは，再実装元のライブラリと類似したコード片を持つと考え，クローンペアとして再実装の検出を試みた．

まず，先行研究 [2] における再実装と，本研究が対象とする再実装の性質に差異があることを明らかにした．このことから，既存のライブラリ機能を自前の実装に置き換える際，置き換えたライブラリの実装をコピーアンドペーストせず，自前で実装することが多いと結論付けた．

次に，GitHub 上のスター数上位 10 件のライブラリとアプリケーションにおいて，再実装がどの程度発生し，どのような場合に発生しやすいかを明らかにした．具体的には，目視確認したライブラリとアプリケーション間の 100 個のクローンペアのうち，76 個が再実装であり，変数名が異なる，説明変数が追加されている等の違いを持つ「書き方の違い」や，「機能の追加または削減」に分類される再実装が多かった．加えて，言語モデル・深層学習関連のライブラリが再実装されやすい傾向があることを明らかにした．

2. 準　　備

2.1 コードクローン

コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことである [3]．コードクローンの関係にあるコード片の組をクローンペアと呼ぶ．

コードクローンを検出するツールには，行単位で最長共通部分列を計算する Nicad [4] や，コードをトークン列に変換し，連続する N トークン列間の最長共通部分列を計算する NIL [5] が存在する．NIL は多数の変更が加えられたコードクローンの検出において Nicad を上回る精度を示している．本研究では，2.2 節で述べる「変更が加えられた再実装」を可能な限り多く検出するために，NIL を採用する．

2.2 再　実　装

大規模化が進む OSS では，必要な機能が外部ライブラリで提供されている事実の認識不足等の理由から，既存機能の利用が適切な場面で，自前の実装を行う「再実装」が発生しうる

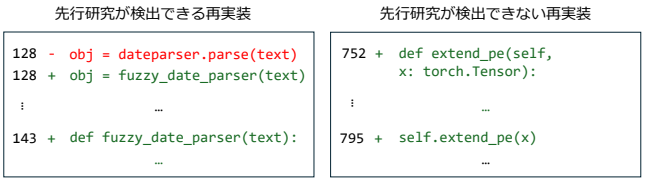

図 1　先行研究で検出できる再実装と検出できない再実装の例

[1]．再実装には，新たな依存関係の導入を回避し，プロジェクトの独立性を保てるという利点がある反面，以下の課題が生じることが先行研究により指摘されている [2]．

- バグ修正やテスト等の保守作業を，再実装した側が自前で行う必要がある
- コピーアンドペーストせずに自前で実装する場合，余分に時間が掛かる
- ライブラリと比較してレビューが十分である保証がなく，信頼性が低下する

先行研究 [2] では，再実装を「利用していた外部ライブラリを自前の実装に置き換えること」と定義し，ライブラリ機能が自前の実装に置き換えられたコミットを調査した．その結果，45 個の Python プロジェクトにおいて計 48 個の再実装された関数を発見した．しかし，置き換えではなく，ライブラリの関数をコピーアンドペーストした場合については先行研究の手法では検出できない．図 1 に先行研究の手法で検出できる場合と，できない場合の例を示す．例の左側のように，ライブラリの関数の呼び出しが自前の実装に置き換えられている場合再実装として検出できるが，例の右側のように置き換えが発生しない場合は検出できない．

先行研究では検出できていないケースが存在することから，本研究では調査対象とする再実装を「外部ライブラリの関数をコピーアンドペーストすること」と定義し，コードクローン検出ツールを用いた調査を行う．ただし，コピーアンドペーストした後に変更を加えた場合も調査の対象とする．再実装の発生数や，その性質を調査することで，先行研究では捉えきれなかった再実装の実態を明らかにする．

3. 調 査 設 計

再実装の実態を調査するために 2 つの RQ を設定する．

RQ1：先行研究 [2] で再実装であると判定された関数は，本研究における再実装の定義でどの程度検知可能か？

RQ2：どのような再実装が最も多く発生しているか？

RQ1 では，先行研究 [2] における再実装と，本研究が対象とする再実装の性質に差異があるかを検証する．RQ1 の目的は，置き換えによる再実装において，どの程度コピーアンドペーストが伴っているかを明らかにすることにある．

RQ2 では，再実装がどの程度発生し，どのような場合に発生しやすいかを明らかにする．

4. リポジトリの収集

4.1 RQ1

先行研究 [2] で再実装であると判定された 48 件の事例から

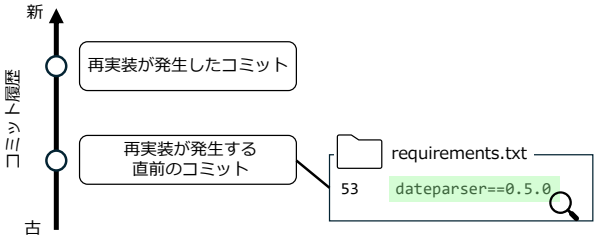

図 2　置き換えられたライブラリのバージョン特定

成るデータセット(注1)に基づき，各再実装に対応するアプリケーションのコミットとライブラリのソースコードをダウンロードした．

表 1 に，先行研究のデータセットの規模を示す．データセットは再実装であると判定されたアプリケーションの関数が含まれるコミットの URL と，置き換え前後の関数の行番号，そのコミットで関数が置き換えられたライブラリの名前から成る．このデータセットに基づき，コミットが削除されている，またはライブラリの情報が欠落しているアプリケーションを除く 45 個のコミットを GitHub からダウンロードした．これらには同じリポジトリの異なるコミットも含まれている．

あわせて，ダウンロードしたアプリケーションに対応するライブラリをダウンロードした．ライブラリについては先行研究のデータセットにバージョン情報の記載がないため，以下の手順でバージョンの特定とダウンロードを行った．

図 2 に示すように，アプリケーションにおいて再実装が行われる直前のコミットに含まれる requirements.txt 等の設定ファイルに記載されているバージョンのライブラリを，Python のパッケージ管理システムである PyPI(注2)からダウンロードした．ただし，以下の場合，アプリケーションで利用していたライブラリのバージョンが不明である．

- アプリケーションの再実装が発生する直前のコミットに含まれる設定ファイルにバージョンの記載がない
- 置き換えられた関数が含まれるライブラリが，標準ライブラリの CPython である

そのため，アプリケーションの再実装が発生する直前のコミット時点における，最新バージョンのライブラリをダウンロードした．なお，CPython についてはパッケージ管理システムで

表 1　先行研究データセットの規模

項目	数
先行研究で再実装と判定されたライブラリとアプリケーションの関数ペア	48
ダウンロード対象外	
アプリケーションのコミットが削除	1
アプリケーションの置き換え前のライブラリが不明	2
ライブラリのソースコードがない	2
ダウンロードしたライブラリとアプリケーションのペア	43

（注 1）：https://github.com/swatlab/reuse_reimpl/blob/master/real_world_examples/reimpl/Python_Reimplementation_Cases.csv

（注 2）：https://pypi.org/

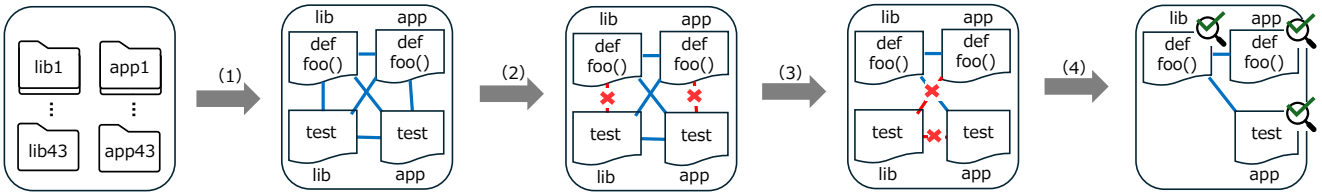

図 3　RQ1 の調査手順

ある PyPI ではなく GitHub からダウンロードした．

以上の手順により，最終的に 43 個のアプリケーションに対応するライブラリをダウンロードした．2 個のアプリケーションに対応するライブラリは，ソースコードが PyPI 上に残っておらず，ダウンロードできなかった．

4.2　RQ2

表 2，3 に示す，主要言語が Python である GitHub 上のスター数上位 10 件のライブラリとアプリケーションのリポジトリをダウンロードした．ただし，学習用リポジトリなどのコンテンツ集や，フォークのプロジェクトは収集の対象外とした．

加えて，本研究において，ライブラリは「他のプロジェクトから利用されているプロジェクト」，アプリケーションは「独立したプログラムとして動作するプロジェクト」と定義した．そこで，収集対象とするライブラリは README.md や公式ドキュメントにコードからの呼び出し例が記載されている，もしくは README.md に “Library” や “Framework” という単語が含まれているプロジェクトとした．またアプリケーションは，収集対象のライブラリ以外で，README.md に実行までの流れや Web サービスの場合はリンクが記載されている，もしくは README.md に “Application” という単語が含まれているプロジェクトとした．

5.　調 査 手 順

5.1　RQ1

図 3 に示す手順で調査を行った．詳細を以下に示す．

手順（1）では，クローンとして検出される関数の最小行数を 10 行に設定し，ライブラリとアプリケーションを NIL に入力してクローン検出を行った．これは，小さい関数が誤って再実装として検出されることを避けるためである．

手順（2）では，各クローンペアが，ライブラリに含まれる関数とアプリケーションに含まれる関数のペアであるかを判定し，該当しないペアは除外した．これは，手順（1）で検出されたクローンペアに単一プロジェクト内のペアや，アプリケーションとアプリケーションの間のペア，ライブラリとライブラリの間のペアが含まれ，これらを取り除くためである．

手順（3）では，ライブラリ側の関数がテストファイルに含まれている場合，外部から利用するために提供されている関数ではないと判断し，そのペアは除外した．一方，アプリケーション側の関数がテストファイルに含まれており，ライブラリ側の関数がテストファイルに含まれていない場合は，ライブラリの関数をコピーアンドペーストしている可能性があるため，除外しなかった．

手順（4）では，目視でクローンペアを確認することで，先行研究で再実装であると判定された関数がアプリケーション側に含まれ，かつライブラリ側も置き換えられた関数であったかを判定した．

以上の手順により，先行研究で再実装であると判定された関数は，本研究における再実装の定義でどの程度検知可能かを確認した．これにより，先行研究における再実装と本研究が対象とする再実装との間で，性質に差異があるかを明らかにした．

表 2　スター数上位 10 件のライブラリ 2025 年 9 月 15 日時点）と Python ファイルの行数，除外したテストファイル

順位	ライブラリ名	行数	除外したテストファイル	
			ファイル数	行数
1	Transformers	1,657,819	1	647
2	PyTorch	2,022,493	33	141,554
3	FastAPI	88,450	1	95
4	Whisper	4,267	0	0
5	Django	499,627	1	1,739
6	Manim	23,606	0	0
7	MarkItDown	6,451	0	0
8	Flask	18,108	0	0
9	Browser Use	55,930	0	0
10	CPython	1,043,723	6	13,164

表 3　スター数上位 10 件のアプリケーション（2025 年 9 月 15 日時点）と Python ファイルの行数

順位	アプリケーション名	行数
1	AutoGPT	132,909
2	Stable Diffusion web UI	43,654
3	youtube-dl	167,906
4	Langflow	168,707
5	The F*ck	16,354
6	ComfyUI	117,955
7	Home Assistant	2,708,023
8	Deep-Live-Cam	3,001
9	Screenshot to Code	6,751
10	GPT Academic	62,319

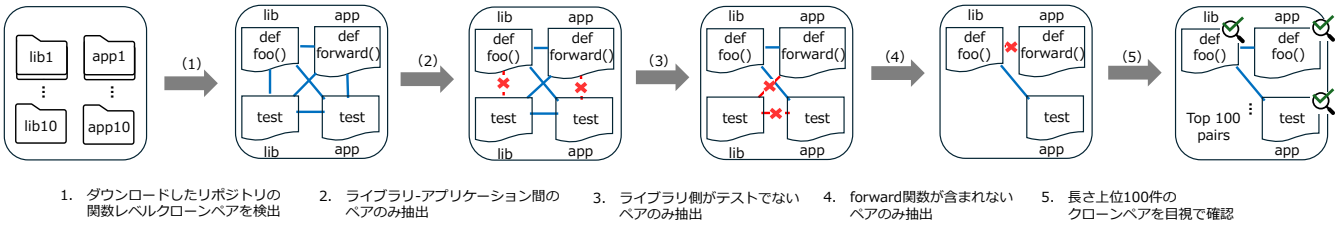

図 4　RQ2 の調査手順

5.2　RQ2

図 4 に示す手順で調査を行った．詳細を以下に示す．なお，手順（1）から（3）は 5.1 節と同様であるため，省略する．

手順（4）では，forward 関数を含むペアの除外を行った．これは，forward 関数の除外前に 30 ペア程度の目視確認を行ったところ，そのほとんどが深層学習の順伝搬を行う関数であり，上位 100 ペアの大部分が forward 関数のペアとなることで，調査データに偏りが生じると判断したためである．

手順（5）では，ライブラリとアプリケーションの関数のうち，短い方の行数に基づき，クローンペアを降順にソートした．その後，目視で確認しながらクローンペア間の差異を明らかにするために表 4 に示す分類を，同じく表に示す根拠に基づき行った．この分類は，クローンペアを著者が目視で確認した上で決定した．この分類を用いることで，再実装の意図がコードから明白であり，ライブラリ機能での置き換えが不要な関数がどの程度存在するのか把握できる．ただし，「再実装であると判定できない」ケースとは，2.2 節の定義に則り，以下のいずれかに該当する場合を指すこととした．

- コメントが一致せず，かつ呼び出す関数がまったく一致しない
- 制御構造を含まず，フィールドの初期化や代入のみを行うといった，再実装の判断をするには不十分な処理しか含まれない

6.　調 査 結 果

6.1　RQ1

ダウンロードしたライブラリとアプリケーション間のクローンペアを NIL で検出し，テストコードを除外した結果，79 個のペアが抽出された．

抽出された 79 ペアの実装を目視で確認したところ，先行研究 [2] で再実装であると判定された 48 個の関数のペアと一致したのは 1 ペアだった．すなわち，既存ライブラリ機能から自前の実装に置き換えたと先行研究で判定された関数のうち，本研究における再実装として検出された関数は，43 個のアプリケーションについてクローンペアを検出したことを考慮すると，約 2.33% に留まったことになる．

以上の結果から，RQ1 に対しては以下のように回答できる．

> RQ1 に対する回答
> 先行研究で再実装と判定された関数は，本研究における再実装の定義で 2.33% というごく一部しか検知できなかった．

6.2　RQ2

表 2，3 のリポジトリに含まれる全ソースコードを NIL へ一括で入力し，ライブラリとアプリケーション間のクローンペアのみを抽出した．その結果，1,000 個のクローンペアが検出された．その後，ライブラリ側の関数がテストファイルに含まれていた場合にそのペアを除外したところ，894 ペアが残った．加えて，forward 関数を含むペアも除外した結果，最終的に 399 ペアが残った．これら除外後の 399 ペアについて，5.2 節で述べた手順に従い，関数が長い順に上位 100 ペアを目視で確認したところ，76 ペアが再実装であると判定できた．

また 100 個のクローンペアについて，表 4 に示す分類を行った．その結果，「書き方の違い」に分類されるクローンペアが最も多く，「機能の追加または削減」がそれに次ぐことがわかった．「コピーアンドペースト」に分類される，ほとんど変更が加えられていないクローンペアも 20 ペア存在した．

次に，どのライブラリとアプリケーションの間で再実装が最も発生しているかを確認するため，各ライブラリとアプリケーション間の再実装発生数を調べた．結果を表 5 に示す．表 5 から，Transformers と ComfyUI の間で生じた再実装が最も多く，次に Transformers と Stable Diffusion web UI の間で生じた再実装が多いことがわかった．このことから，言語モデル関連の

表 4　クローンペアの分類と発生数

大分類	小分類	分類根拠の説明	発生数
再実装	書き方の違い	説明変数を挟んでいる，変数名が異なる，利用ライブラリが異なる 等	40
	機能の追加または削減	引数や分岐，機能の数に差がある	31
	コピーアンドペースト	private - public の違いは考慮せず一致	20
	独自要素への置換または独自要素の挿入	独自の例外やクラスに置き換わる，プロジェクト名が出力に追記される	11
	互換性対応	互換性対応を目的とする	6
	バグ修正	コメントにバグ修正が目的である旨を含む	1
再実装ではない		再実装であると判定できない	24

ライブラリ

```
def rot_pos_emb(self, grid_thw):
    pos_ids = []
    for t, h, w in grid_thw:
                …
        wpos_ids =
wpos_ids.permute(0, 2, 1, 3)
        wpos_ids =
wpos_ids.flatten()
                …
    rotary_pos_emb_full =
self.rotary_pos_emb(max_grid_size)
    rotary_pos_emb =
rotary_pos_emb_full[pos_ids]
.flatten(1)
    return rotary_pos_emb
```

アプリケーション

```
def get_position_embeddings(self,
grid_thw, device):
    pos_ids = []
    for t, h, w in grid_thw:
                …
        wpos_ids =
wpos_ids.permute(0, 2, 1,
3).flatten()
                …
    rotary_pos_emb_full =
self.rotary_pos_emb(max_grid_size,
device)
    return
rotary_pos_emb_full[pos_ids]
.flatten(1)
```

図 5　ライブラリとアプリケーション間の再実装例

ライブラリである Transformers の機能は，Stable Diffusion に関係するアプリケーションにおいて頻繁に再実装されていることがわかる．図 5 に Transformers と ComfyUI の間で生じた再実装の例を示す．このペアでは，アプリケーション側が device という引数を追加することで，デバイスを指定して PyTorch の関数を呼び出せるようにしている．また，flatten 関数を呼び出している箇所で，ライブラリ側が 2 行で行っている処理をアプリケーション側は 1 行にまとめて書いている．このことから，「書き方の違い」，「機能の追加または削減」に分類した．

調査したライブラリは最大で 2 つのアプリケーションにのみ再実装されており，多くのアプリケーションに再実装されているライブラリは確認されなかった．

以上の結果から，RQ2 に対しては以下のように回答できる．

RQ2 に対する回答

100 個のクローンペアを目視で確認した結果，「書き方の違い」に分類される再実装が 40 件と最も多く，次いで「機能の追加または削減」に該当する再実装が 31 件存在した．このように，ライブラリの関数の機能が不足または過剰である場合や，アプリケーション側の要件とライブラリ側が依存する別のライブラリが異なる場合などに，アプリケーション側で再実装が行われる可能性が示唆される．
また，調査の範囲内では，言語モデル関連のライブラリと Stable Diffusion に関連するアプリケーションの間で再実装が多く確認された．

7. 考　　察

7.1 先行研究と本研究における再実装の差異

6.1 節の結果は，先行研究と本研究が対象とする再実装の性質が異なる可能性を示唆している．すなわち，既存のライブラリ機能を自前の実装に置き換える際，置き換えたライブラリの実装をコピーアンドペーストするのではなく，自前で実装することが多いと考えられる．このことから，2.2 節で述べた「実装に余分な時間がかかってしまう」という再実装の課題が強調される．一方で，置き換えによる再実装を行う場合は依存関係を最小限に抑えられることが示唆される．

表 5　各ライブラリとアプリケーション間の再実装発生数

ライブラリ	アプリケーション	再実装発生数
Transformers	ComfyUI	46
Transformers	Stable Diffusion web UI	17
CPython	youtube-dl	6
PyTorch	ComfyUI	4
CPython	GPT Academic	1
Django	Home Assistant	1
PyTorch	Home Assistant	1

7.2 再実装の発生頻度と開発者への提言

6.2 節の結果より，目視確認した 100 個のクローンペアのうち 76% が再実装であった．これにより，GitHub でのスター数が多い著名なライブラリとアプリケーションの間であっても，再実装は稀な現象ではないことが明らかとなった．

単一のリポジトリ内において，コードをコピーアンドペーストした箇所がコピー元の変更に追随する割合は約半分に留まることが先行研究 [6] で指摘されている．異なるプロジェクト間においてはメンテナが異なる場合が多いために，一方の変更を検知することが同一リポジトリ内よりも困難であり，ライブラリ側の変更に追随しないリスクは先行研究の指摘よりも高くなると考えられる．ライブラリ側の変更に追随しない場合，バグ修正などがアプリケーション側に適切に反映されず，不具合を内在したままリリースされる危険性がある．

このことから，アプリケーション開発者は，ライブラリの機能を再実装する際，関数のコメントや README 等のドキュメントにその事実やコピー元の情報を記録しておくべきである．これにより，リポジトリの参照者への周知が可能となり，ライブラリ側で再実装の元となった関数が更新された際に，その変更に追随できる可能性を高められると考えられる．本研究で再実装と判定した関数の多くは，コピーアンドペーストの痕跡があるにも関わらず，コメントにコピー元の情報が記載されていなかった．加えて，IoT 分野の OSS プロジェクト間におけるコードクローンを調査した研究 [7] では，クローンペアの管理に自動化ツールを用いる重要性が述べられている．本研究において，コードクローン検出ツールを用いることでライブラリとアプリケーション間の再実装を検出できた．この結果は，アプリケーション開発者が再実装の管理に自動化ツールを活用できるという実効性の裏付けになると考えられる．

また，ライブラリ開発者がバグ修正などの重要な変更を行う際には，たとえ外部からの利用を想定していない関数が変更の対象であっても，リリースの実施とリリースノートへの変更内容の明記を通じて，ライブラリの利用側へ変更を周知することが望ましい．これにより，Dependabot(注3)等の依存関係更新ツールを利用しているプロジェクトに対し，更新の通知やプルリクエストの作成を通じた伝達が可能となる．

さらに，アプリケーション側だけでなく，ライブラリ開発者側へのフィードバックを支援する仕組みも不可欠である．ライブラリ開発者は自分のライブラリが 6.2 節で示した規模で再実装されている実態を把握する手段を持たないため，外部からの利用を想定していない関数の変更を周知することが困

(注3)：https://github.com/dependabot

ライブラリ

```
def
get_2d_sincos_pos_embed(embed_dim,
grid_size, add_cls_token=False):



    grid_h = tf.range(grid_size,
dtype=tf.float32)
    grid_w = tf.range(grid_size,
dtype=tf.float32)
                …
```

アプリケーション

```
def
get_2d_sincos_pos_embed(embed_dim,
grid_size, cls_token=False,
extra_tokens=0,
scaling_factor=None,
offset=None,):
    grid_h = np.arange(grid_size,
dtype=np.float32)
    grid_w = np.arange(grid_size,
dtype=np.float32)
                …
```

図 6　ライブラリ側とアプリケーション側で利用しているライブラリが異なる例（tf は Tensorflow，np は numpy を表す）

難であると考えられる．したがって，自分のライブラリが他のプロジェクトで再実装されているかを自動的に検知し，ライブラリ開発者に提示するツールが必要である．

7.3　再実装のカテゴリに基づいた知見と提言

6.2 節の結果より，書き方の違いに関する再実装の割合が最も高かった．書き方の違いには，単なる変数名の不一致や説明変数の追加といった軽微な違いから，利用しているライブラリの違いまで含まれる．加えて，機能の追加または削減に関する再実装も多く発生しており，ライブラリの機能が不足または過剰である場合があると考えられる．

この結果から，アプリケーション開発者は，図 6 に示す例のように，アプリケーション側とライブラリ側で利用しているライブラリが異なり，ライブラリを変更できない等の制約がない限り，ライブラリの既存機能の利用を検討するべきである．また，アプリケーション開発者がライブラリの既存機能に単に処理を加える際は，デコレータ等の拡張手段を利用して機能を追加するべきである．ライブラリの関数をコピーアンドペーストせず，拡張手段を利用して機能を追加することによって，ライブラリが提供している機能の保守を自前で行う必要がなくなる．そのため，「バグ修正やテスト等の保守作業を，再実装した側が自前で行う必要がある」という再実装の課題による影響を最小限に抑えることができると考えられる．

ライブラリ開発者は，図 6 に示す例のように，アプリケーション側が利用するライブラリが異なる場合でも機能を流用できるよう，特定のライブラリに依存しない柔軟な実装をするべきである．また，ライブラリ開発者は，デフォルト引数の活用により，一部の引数のみ指定しても動作するような柔軟な関数の設計をするべきである．機能が過剰な場合には，関数の分割等も検討するべきである．

7.4　再実装の傾向

6.2 節の結果より，言語モデル関連のライブラリと，Stable Diffusion という画像生成 AI に関連するアプリケーションの間で多くの再実装が確認された．この事実から，ライブラリは対象とする分野と関連するアプリケーションにおいて，再実装されやすい傾向があることが示唆される．

また，Transformers や PyTorch 等の言語モデル・深層学習関連のライブラリが再実装されやすい傾向が 6.2 節の結果から示唆される．これらのライブラリはモデルの提供を主目的としており，外部からはモデルの利用のみを想定している場合がある．しかし，実際にはその内部実装に含まれる汎用的な関数も，アプリケーション側は必要としていることが 6.2 節の結果から示唆される．したがって，言語モデルや深層学習関連のライブラリの開発においては，内部実装でも，汎用的な関数は外部から利用可能にする設計が求められると考えられる．

8.　まとめと今後の展望

本研究では，先行研究の再実装と本研究における再実装の差異に関する調査（RQ1）と，GitHub 上のスター数上位 10 件のライブラリとアプリケーションにおける再実装の実態に関する調査（RQ2）を行った．

RQ1 では，先行研究で再実装であると判定された 48 個の関数について，NIL を用いて検出を試みた．その結果，既存のライブラリ機能を自前の実装に置き換える際，置き換えたライブラリの実装をコピーアンドペーストせず，自前で実装することが多いという傾向が明らかになった．

RQ2 では，GitHub 上のスター数上位 10 件のライブラリとアプリケーションを対象に，NIL によるクローンペア検出を行った．その結果，ライブラリの機能が不足または過剰である場合や，アプリケーション側の要件とライブラリ側が依存する別のライブラリが異なる場合に，再実装が発生しやすいという傾向が明らかになり，それに基づくライブラリとアプリケーションの開発者双方への提言をまとめた．

本研究の課題として，RQ2 における分類を著者 1 人で行った点，および長さ上位 100 個のクローンペアしか確認できていない点が挙げられ，客観性や一般化に課題が残る．今後は，検出された計 399 個のクローンペアを複数人で確認し，議論を行った上で主観を排除した分類を行う予定である．

謝辞　本研究は JSPS 科研費 24H00692, 25K03102, 22H03567 の助成を受けた．

文　　献

[1] D. Kawrykow and M.P. Robillard, “Improving api usage through automatic detection of redundant code,” Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp.111–122, 2009.

[2] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo, “Why reinventing the wheels? an empirical study on library reuse and re-implementation,” Empirical Software Engineering, vol.25, no.1, pp.755–789, 2020.

[3] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proceedings of the International Conference on Software Maintenance, pp.368–377, 1998.

[4] J.R. Cordy and C.K. Roy, “The nicad clone detector,” Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, pp.219–220, 2011.

[5] T. Nakagawa, Y. Higo, and S. Kusumoto, “Nil: large-scale detection of large-variance clones,” Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.830–841, Association for Computing Machinery, 2021.

[6] R. Yokomori and K. Inoue, “An empirical analysis of git commit logs for potential inconsistency in code clones,” Proceedings of the 2024 IEEE International Conference on Source Code Analysis and Manipulation, pp.1–12, 2024.

[7] W. Zhu, N. Yoshida, Y. Matsubara, and H. Takada, “Multilingual investigation of cross-project code clones in open-source software for internet of things systems,” IEEE Access, vol.12, pp.179104–179118, 2024.