Vol.2021-SE-207 No.8
2021/3/1

マイクロベンチマークサービスにおけるプログラム断片の分析

才木 一也[1,a)]　伊原 彰紀[1,b)]

概要：ソフトウェアベンチマークは，異なる実装方法における実行時間，メモリ消費量，CPU 使用率などの性能効率性を比較するために使用される．特に，マイクロベンチマークは，プログラム中の数行程度の性能を比較する方法であるため，汎用的に使用できる実装が多く，Web 上でマイクロベンチマークの結果を公開するサービスも存在する．開発者は自身のプログラム断片でも同じ性能結果を得ることができるか確認することが多く，サービスには類似するベンチマーク（トピック）の結果が多数公開されている．類似するトピックを収集することで，多様な実装を比較することが可能であると考える．本論文では，多様な実装を収集・比較するために，ベンチマークを計測したプログラム断片に基づいて，類似するトピックを分類する手法を提案する．ケーススタディとして，JavaScript プログラムのマイクロベンチマークサービス jsPerf に登録される 12 万件のトピックで計測されたプログラム断片の類似度を測定し，類似するトピックの紐付けを行った．特に，検証されることが多い 2 件のトピック “for vs forEach” と “innerHTML vs removeChild” に類似するトピックをそれぞれ収集した結果，多様な実装が収集可能であることを明らかにした．

A Classification of Micro Benchmark using Program Snippets

1. はじめに

ソフトウェアの性能効率性は，JIS X 25010:2013 (ISO/IEC 25010:2011) においてシステム/ソフトウェア製品品質の一つに含まれており，ソフトウェア品質を測る重要な尺度である [7]．昨今では，ソフトウェアの大規模化，多機能化に伴い，ソフトウェアシステムのスループット，平均応答速度の低下が課題として挙げられる．特に，Web システム，Web アプリケーションの役割が大きくなり，ソフトウェアシステムの開発において性能効率性を向上する必要性が高まっている．しかし，ソフトウェアの性能効率性に関する問題は，セキュリティに関する問題に比べて優先的に改善されることは少ない．その理由の一つは，性能効率性に関する問題の改善には，開発者の経験，プログラミング知識に依存することが多く，ソフトウェアのリリース後に気づくことが多いためである [5]．

ソフトウェア性能効率性の負荷検証を行うためには数千・数万件規模のテスト実行など，膨大な検証作業を必要とする．ソフトウェア開発中に大規模な検証を実施する前に，検証対象を一部の処理に絞った方法としてマイクロベンチマークが使用される．マイクロベンチマークは，メソッドのようなプログラム中の一部の性能評価を行うベンチマークであり，マイクロベンチマークを低コストで実施するためにWeb サービスが提供されている．JavaScript で実装されたプログラムを対象とするマイクロベンチマークサービス jsPerf [*1]では，過去に多数の開発者が JavaScript プログラムとその実行速度を評価した結果を公開している．マイクロベンチマークは，プログラム中の数行程度の性能を比較する方法であるため，汎用的に使用できる実装が多いが，自身のプログラム断片でも同じ性能結果を得ることができるか確認することが多く，サービスに登録されるベンチマーク（トピック）には，類似するトピックが多数存在する．類似するトピックを収集することで，多様な実装を収集・比較することが可能であると考える．

本論文では，多様な実装を収集・比較するためにプログラム断片の類似度に基づいて，類似するトピックを分類

[1] 和歌山大学システム工学部　〒 640–8510 和歌山県和歌山市栄谷 930
[a)] s226101@wakayama-u.ac.jp
[b)] ihara@wakayama-u.ac.jp

[*1] jsPerf: https://jsperf.com/ (2020 年 2 月現在アクセス不可)

ⓒ 2021 Information Processing Society of Japan

する手法を提案する．具体的には，対象とするマイクロベンチマークにおいて比較するプログラム断片を収集し，Code2Vec を用いてプログラム断片の特徴ベクトルに基づきトピック間の類似度を算出する．そして分類の基準とする類似度の閾値を作成し，閾値を超える類似度を示すトピックを分類する．

続く 2 章では，マイクロベンチマークの概要と，関連研究，および課題を述べる．3 章で類似したベンチマークの分類方法，4 章で評価方法について述べ，5 章で対象とするデータを指定し，分類を行う．そして 6 章で分類結果に対して，妥当性や分類したことによる効果についての考察を述べる．7 章でまとめを行う．

2. マイクロベンチマーク

2.1 性能効率改善の関連研究

性能効率性の問題を検出する手法として性能回帰テストがある．性能回帰テストは，システムに負荷をかけて，システムが実際に使用される環境における性能や耐久性を検証する [2, 3]．性能回帰テストは数時間から数日に渡って実行され，テスト中にシステム運用中の性能効率性の情報を記録する．テスト終了後，収集した性能効率性の情報が過去のテスト結果と乖離がないかを調査する．性能回帰テストは収集するメトリクス数が多いためテストの実施に時間がかかり，また性能の目標値は，分析担当者の経験や主観に頼らざるを得ない．

性能回帰テストにかかるコストを少なくするために，従来研究では性能回帰を自動で検出する手法を提案している．Luo らが提案するシステム PerfImpact は，2 つのリリース間で性能低下の原因となる可能性のあるコード変更を自動的に識別している [4]．Luo らのアプローチは，性能回帰を示す入力値を検索し，問題のあるコード変更を特定するために，ソフトウェアの 2 つの実行トレースを比較している．しかし，性能効率性の問題を検出するタイミングはシステムをリリースした後に検出されることが多く，ソフトウェア開発のリリース前に性能効率性の問題を検出，修正するためには膨大なリソースが必要となる．また，修正方法はソフトウェアに依存し，過去と同じ修正方法で改善できるとは限らない．

2.2 マイクロベンチマークツール: jsPerf

マイクロベンチマークは，数千・数万件規模のデータの準備，テスト実行など大規模な性能テストとは異なり，より小さな規模のクラスやメソッド単位のプログラムを性能テストするベンチマークである．マイクロベンチマークにおける性能評価は，同じ入出力でありながら，異なる方法で実装された 2 つ以上のプログラムの性能を比較するものである．

マイクロベンチマークツールとして，Java 言語で実装

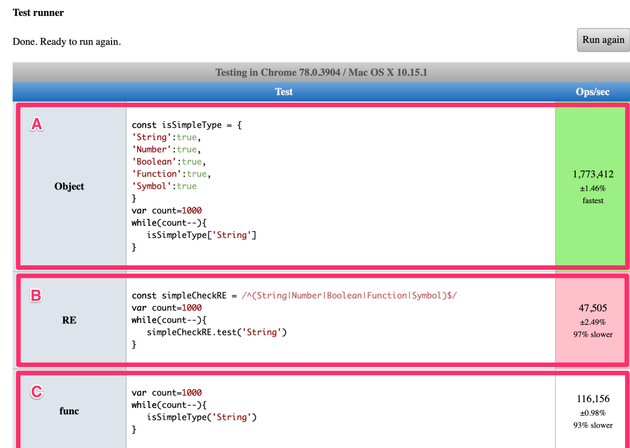

図 1 “REandObject” を構成するテストケース

されたプログラムを対象とするマイクロベンチマークのフレームワークには，JMH (Java Microbenchmark Harness) や Caliper などがある．JMH，Caliper は，計算機にインストールすることで性能テストを実行するため，性能テスト結果を多数の開発者と共有することはない．一方で，JavaScript 言語で実装されたプログラムを対象とするマイクロベンチマーク jsPerf は，開発者が Web サービスにプログラム断片をアップロードし，Web サービス上で性能効率テストを実行し，異なる方法で実装されたプログラム断片の性能を比較することができる．さらに，性能評価した結果は，Web 上で公開・共有することができる．過去に jsPerf において比較されたプログラム，および，その性能効率性を比較した結果は，誰でも参照することができ，性能効率性の高い実装方法を知ることができる．2020 年まではサービスが運用していたが，2021 年 2 月現在 jsPerf のサーバは停止している．jsPerf のシステムは GitHub[*2]に公開されている．本論文は 2020 年 7 月に収集した jsPerf のデータを使用する．

jsPerf の具体的な出力結果を，jsPerf において公開していたトピック “REandObject” [*3]の事例を用いて説明する．図 1 は，比較されたプログラム断片のそれぞれの性能評価結果を A，B，C の領域に示している．各領域内の左から，プログラム断片のタイトル，性能を比較するプログラム断片，性能評価の結果を示す．性能評価はプログラムの実行速度を計測し，複数のプログラム断片の性能を比較した結果を示す．実行速度が速い結果を緑色，遅い結果を場合は赤色で表示される．図 1 では A のプログラム断片の実行速度が最も速く，B が最も遅いことを示す．

jsPerf では，過去に実行された性能評価のプログラム断片の一部を，別の開発者が編集（追加・削除・変更）し，性能評価を再実行することも可能である．編集されたプロ

[*2] jsPerf: https://github.com/jsperf/jsperf.com
[*3] jsPerf“REandObject”: https://jsperf.com/reandobject/3（2020 年 2 月現在アクセス不可）

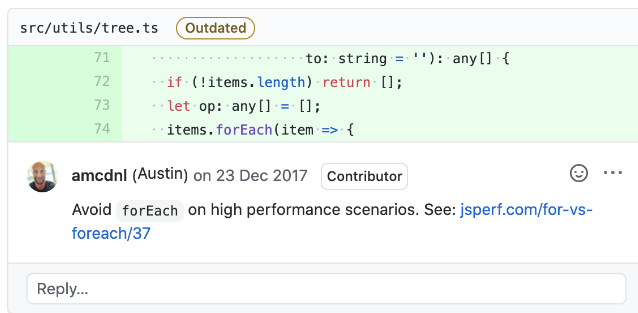

図 2 変更要求の根拠として jsPerf を参照する実例

グラム断片は，同じトピックの異なるリビジョンとして保存される．jsPerf に投稿されているマイクロベンチマークは，Web で公開されていたため，開発者が自由に参照することが可能であった．

2.3 性能評価を要する実装方法の多様性

jsPerf において公開された性能評価は，ソフトウェア開発者が実行性能の高いプログラム実装方法を説明するための裏付けとして活用することも多い [6]．GitHub の Pull Request において，ソフトウェアの性能効率を改善する変更要求の根拠として jsPerf に投稿されているマイクロベンチマークを活用している実例を図 2 に示す *4．本事例で参照されているマイクロベンチマークのトピック “for vs forEach” では JavaScript で実装するループ処理のうち “for” と “forEach” の性能効率を比較しており，“for” が “forEach” の実装よりも性能効率が良いと評価結果を出している．この結果に基づき，本 Pull Request で議論する開発者は “forEach” を用いた実装から “for” を用いた実装に変更するよう要求している．このように jsPerf に投稿されるマイクロベンチマークはパフォーマンス改善に活用される．

JavaScript で実装するループ処理は “for” と “forEach” 以外にも “while”，“for ... in”，“for ... of” などをはじめとして多岐にわたり存在しているが，図 2 に示す実例では “for” と “forEach” を用いたループ処理のみの性能評価を計測しており，他のループ処理に関するパフォーマンス改善の検討が行われていない．jsPerf は多数のトピックで性能効率を計測しているが，トピック間で紐付けられていないため，開発者が多様な実装のトピックを収集することは容易ではない．本論文では，類似するトピックを分類するための手法を提案し，多様な実装方法の収集を実現する．

3. ベンチマークの分類方法

3.1 概要

図 3 は，本論文で提案するトピックの分類方法の概略図を示す．提案する分類方法は，jsPerf の各トピックの最新リビジョンで比較されたプログラム断片を Code2Vec を用いて特徴ベクトルに変換し，トピック間の特徴ベクトルの類似度に基づきトピックを分類する．分類基準に用いる閾値は，同一トピックに含むリビジョンのプログラム断片から生成したベクトル間の類似度の平均値を使用する．本章では，分類方法の詳細を順に述べる．

3.2 手順 1: プログラム断片の収集

jsPerf の各トピックの最新リビジョンで比較されたプログラム断片を収集する．ただし，構文に誤りを含むプログラム断片や，プログラムの長さが極端に短いプログラム断片は，トピックの類似度を判別できないため分析対象外とする．本論文ではプログラムの最小単位であるトークンに分け，トークン数が 5 以上のプログラム断片のみを分析対象とする．さらに，JavaScript で記述されたプログラムを抽象構文木 (AST) に変換するツール esprima *5を用いて，分析対象の中から実行可能であるプログラム断片のみを対象とする．

3.3 手順 2: プログラム断片のベクトルの生成

本論文では，jsPerf に投稿されるプログラム断片から類似する実装方法を収集するために，Code2Vec [1] を用いて各トピックに投稿されたプログラム断片の特徴をベクトル化する．Code2Vec はプログラムを連続分散ベクトルとして表現するためのニューラルネットワークモデルである．プログラムのメソッド部分を抽象構文木のパスの集合に分解し，各パスの原子表現を学習すると同時に，それらの集合を集約する方法を学習することによって，メソッド本体のベクトル表現からメソッド名の予測を行う．Code2Vec の使用にはプログラム中のメソッド部分が必要であるが，マイクロベンチマークに存在するプログラム断片はメソッドとして記述されておらず，処理内容のみ記述されている．本論文ではプログラム断片をメソッドの本体，マイクロベンチマークのタイトルをメソッド名としてプログラム断片をメソッドの形式に変換することで Code2Vec を実行する．各トピックに含まれる複数のプログラム断片の特徴ベクトルの平均をトピックの特徴ベクトルとする．

3.4 手順 3: トピックの類似判定基準の算出

本論文では，各トピックに類似するトピックを紐付けることで分類を行う．ただし，実装方法の種類が多く存在するトピック間の類似度は低くなることが考えられる一方で，実装方法の種類が少なく，規模が小さいトピック間の類似度は高くなることが考えられる．本論文では各トピックに含まれる最新のリビジョン（最大 10 件分）のプログラム断片から生成したベクトル間の類似度をコサイン類似

*4 https://github.com/swimlane/ngx-datatable/pull/1189#discussion_r158553249

*5 esprima: https://esprima.org/

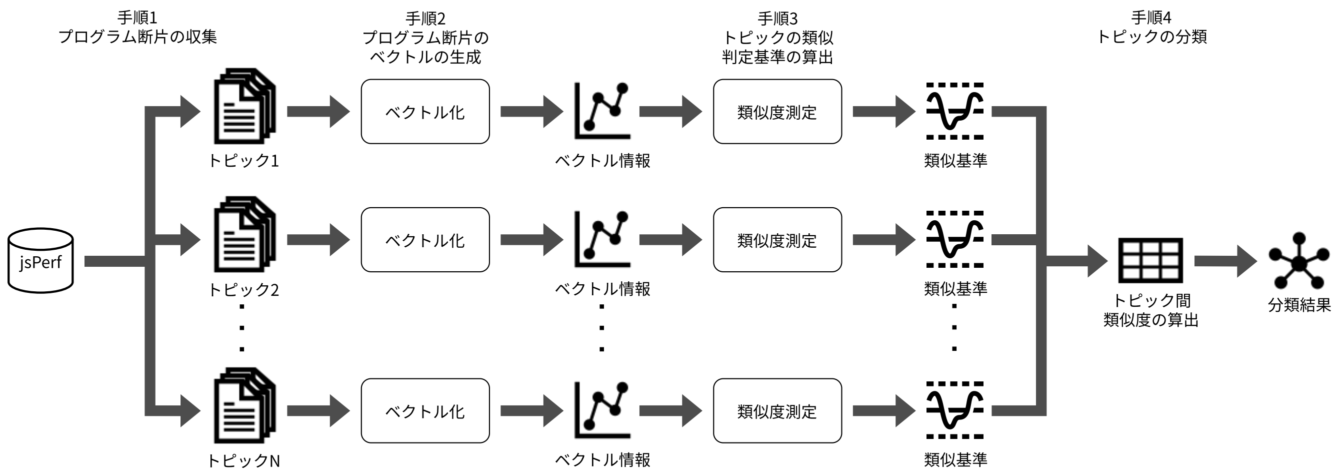

図 3 類似するトピックの分類手法の概略図

度を用いて算出し，類似度の平均値を閾値とする．トピック間の類似度が閾値以上である場合，類似するトピックとして分類する．

3.5 手順 4: トピックの分類

各トピックの最新リビジョンの特徴ベクトルに基づき，トピック間の類似度をコサイン類似度を用いて算出する．分析対象とするトピックの全ての組み合わせにおいて類似度を計測する．手順 3 においてトピックのリビジョンから決定した閾値以上のトピックのみ紐付けし，トピックを分類する．トピック間の類似度を用いて分類する際，1 つの組み合わせにつき，2 つのトピックのそれぞれの閾値と 1 つの類似度が比較される．比較の結果，双方の閾値を超える場合や，双方の閾値を超えない場合，片方のみ閾値を超え，一方からは閾値を超えない場合がある．本手法では閾値を超えたトピックのみを起点とした紐付けを行う．

4. 評価方法

4.1 評価観点

本章では 3 章で提示した手法によって分類した結果を 2 つの観点で評価する．

- **閾値の妥当性**: 各トピックに閾値を設定し，閾値以上の類似度を示すトピックを分類したが，提案する閾値によって分類されたトピックに不適切なトピックが含まれているか否かを確認する．
- **トピックの多様性**：類似するトピックの中に，多様な実装方法のトピックが含まれているか否かを確認する．

4.2 評価方法

本論文では，提案手法によって特定のトピックに紐付けられたトピックのプログラム断片を著者らが目視で確認し，特定のトピックに関連する内容であるか否かを判断する．特に，特定のトピックで比較された実装内容とは関係のない実装が含まれているか否か，また，特定のトピックにおける実装内容以外で同じ目的の実装を含むトピックを分類できているか否かを分析する．

5. ケーススタディ

5.1 データセット

本論文は，マイクロベンチマークサービス jsPerf を対象にケーススタディを行う．本分析では jsPerf に登録される約 40 万件のトピックから，頻繁に比較されるプログラムを対象とする．図 4 は，jsPerf に登録される各トピックのタイトルにおいて使用頻度の多い単語上位 100 件のパレート図を示す．トピックのタイトルに頻繁に使用される単語は `array`，`jquery`，`for`，`object`，`string` などの配列処理，繰り返し処理に関連する性能効率性を検証している．使用頻度の多い 100 件の単語は，jsPerf に登録される約 34%のトピック（126,137 件）で使用されており，本論文ではこれらトピックを対象に分析する．

5.2 分類結果

図 5 は，3 章で述べた手法により類似するトピックを紐付けた結果をネットワーク図で示す．図中の赤色のノードは，紐づけられたトピック数が最も多い 20 件のトピック，青色のノードが赤色のノードに対し，閾値以上の類似度を示したトピックを示す．紐づけられた青いノードは，赤色のトピックとの類似度が高いほど近くに配置されている．図中左と右に大きなノード群が見られ，左側の群はループ処理に関する性能効率を検証したトピックであり，右側の群は関数オブジェクトの生成に関する性能効率を検証したトピックである．

全てのトピックを目視で確認することは難しいため，本論文では jsPerf において性能効率を検証する頻度の多いトピック “for vs forEach” と “innerHTML vs removeChild” を対象に，本論文が提案する分類基準として使用した閾値

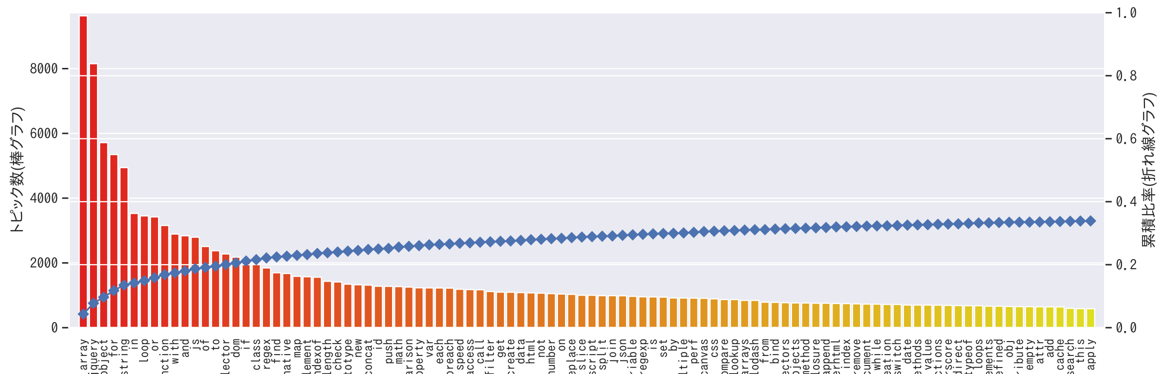

図 4　jsPerf におけるトピックのタイトルに使用される頻出単語（上位 100 件）

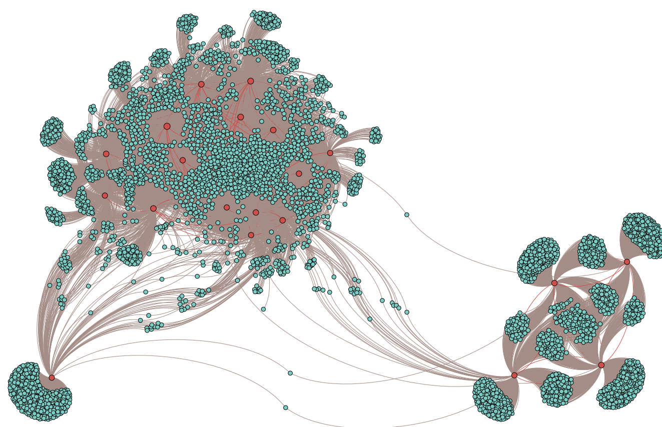
図 5　紐づけられたトピック数が多い上位 20 件分のトピック（赤ノード）の分類結果を表すネットワーク図

表 1　目視で分析したトピックの閾値と分類されたトピック数

	for vs forEach	innerHTML vs removeChild
閾値	0.658	0.741
分類されたトピック数	320	25

の妥当性，トピックの多様性を調査する．表 1 は，各々の閾値と分類されたトピックの数を示す．

表 2 と表 4 は，それぞれ “for vs forEach”，“innerHTML vs removeChild” のトピックの最新リビジョンに含まれるプログラム断片の一覧である．“for vs forEach” はループ処理に使用する関数の比較，“innerHTML vs removeChild” は HTML 要素の削除による性能効率の比較を行うトピックである．

閾値の妥当性:　分析対象とするトピックのそれぞれに紐付けられたトピックのプログラム断片が，実際に類似するトピックか否かを判断するため，本論文では分析対象とするトピックと類似度が高いトピックから順に確認し，閾値よりも 0.1 低い類似度までのトピックを目視で確認した．具体的には，トピック “for vs forEach” の閾値は 0.658 のため 0.558 以上の類似度を持つ 660 件のトピック，トピック “innerHTML vs removeChild” の閾値は 0.741 のため 0.641 以上の類似度を持つ 83 件のトピックを目視で確認した．図 6 と図 7 は，それぞれトピック “for vs forEach” とトピック “innerHTML vs removeChild” における類似度別の類似するトピックの割合を示す．青い範囲は著者らが分析対象のトピックと類似するとみなしたトピックの割合，赤い範囲は分析対象のトピックと類似しないとみなしたトピックの割合を示す．図中の黒線は各トピックの閾値を示す．トピック “for vs forEach” の閾値以上に 98%，閾値から 0.1 低い類似度以上に 83%の類似していないトピックが含まれ，トピック “innerHTML vs removeChild” の閾値以上に 92%，閾値から 0.1 低い類似度以上に 55%の類似していないトピックが含まれていたため，本手法が提案する閾値は，類似しないトピック数を少なく抑えることができることを確認した．

トピックの多様性:　表 3 と表 5 は，それぞれ “for vs forEach”，“innerHTML vs removeChild” に紐付けられた閾値以上のトピックの中で，分析対象のトピックでは比較されていなかった方法のプログラム断片の一部を示す．“for vs forEach” では “for” と “forEach” を用いたループ処理のみの比較となっているが，分類することによって，“for ... of” や “map” などの他の実装が得られた．同様に “innerHTML vs removeChild” では “innerHTML” や “removeChild” を用いた実装の比較となっているが，分類によって “remove” を用いる実装や，“childNodes[i]” や “hasChildNodes” を用いた走査方法の実装などが得られた．

6.　考察

6.1　分類結果に基づく考察

本論文が提案するトピックのリビジョンに含まれるプログラム断片の類似度に基づく閾値を用いることで，類似するトピックに関係のないトピックの混入は少なく，高い精度で分類することができた．リビジョンは同一のトピック内での変更履歴であり，トピックの趣旨から大きく逸脱したプログラム断片が混入することが少ないため，高い精度

ⓒ 2021 Information Processing Society of Japan

表 2 トピック “for vs forEach” に含まれているプログラム断片

プログラム断片
for (i = values.length - 1; i >= 0; i--) { sum += values[i]; }
for (i = 0; (value = values[i]) !== undefined; i++) { add(value); }
length = values.length; for (i = 0; i <length; i++) { sum += values[i]; }
values.forEach(add);
for (i = values.length; i--;) { sum += values[i]; }
length = values.length; for (i = 0; i <length; i++) { add(values[i]); }
for (i = 0; i <values.length; i++) { sum += values[i]; }
for (i = values.length - 1; i >= 0; --i) { sum += values[i]; }
for (i = values.length - 1; i; i--) { sum += values[i]; }
for (i = 0; (value = values[i]) !== undefined; i++) { sum += value; }

表 3 トピック “for vs forEach” に紐付けられたトピックのプログラム断片

トピックタイトル	類似度	プログラム断片

for vs forEach vs map	0.986	$.each(values, function(key, value) {
		    sum += value;
		});

for vs forEach vs map	0.986	values.reduce(add, 0);

for vs forEach vs map	0.986	for (const value of values) {
		    sum += value;
		}

for vs forEach vs map	0.986	for (i in values) {
		    sum += values[i];
		}

for vs forEach vs map	0.986	for (i = values.length - 1; (value = values[i]) !== undefined; i--) {
		    sum += value;
		}

Iteration methods	0.926	length = values.length
		while(length--){
		  sum += values[length];
		}

For vs Foreach vs Map	0.878	values.map(add)

for vs forEach	0.836	var i = 0;
		var value = 0;
		var len = values.length;
		values.every(function(item) {
		  return true;
		});

For loops jQuery & Dojo	0.764	var i;
		for (i in values) {
		  if (values.hasOwnProperty(i)) {
		    values[i] += 1;
		  }
		}

で分類できたと考える．しかし，閾値以下にも類似するトピックが多く含まれていた．類似するプログラム断片から閾値を作成したため，同じ目的のプログラム断片であっても大きく異なるプログラム内容であれば紐付けすることが難しい．“for vs forEach” で比較されている実装は基本的なループ処理の構文であるため，様々な実装に含まれる可能性が高い．例えば配列に要素を追加や削除などの操作をする実装は，ループ処理に組み込まれる場合がある．今後は，閾値以下のトピックからも類似するトピックを抽出する方法を検討する．

分類の基準としたトピックには存在しない実装のプログラム断片を，類似したトピックの分類によって収集できる

表 4 トピック “innerHTML vs removeChild” に含まれているプログラム断片

プログラム断片
`box.textContent = ’’;`

`while (box.lastChild) {`
`    box.removeChild(box.lastChild);`
`}`

`box.innerHTML = ’’;`

`while (box.firstChild) {`
`    box.removeChild(box.firstChild);`
`}`

表 5 トピック “innerHTML vs removeChild” に紐付けられたトピックのプログラム断片

トピックタイトル	類似度	プログラム断片

Fastest method of clearing all children from a DOM element	0.951	`while (box.firstChild) {`
		`    box.firstChild.remove();`
		`}`

Remove an element children	0.945	`box.querySelectorAll(’*’).forEach(element =>element.remove());`
Remove Node Children	0.900	`$(box).empty()`

Remove Node Children	0.899	`for (var i = box.childNodes.length - 1; i >= 0; i--) {`
		`  box.removeChild(box.childNodes[i]);`
		`}`

remove vs parentNode.removeChild	0.890	`var child;`
		`while (child = box.lastChild) {`
		`  child.remove();`
		`}`

remove vs parentNode.removeChild	0.890	`var child;`
		`while (child = box.lastChild) {`
		`  child.parentNode.removeChild(child);`
		`}`

Emptying a select box	0.827	`box.options.length = 0;`

Emptying a select box	0.827	`while (box.options.length) {`
		`  box.removeChild(box.options[0]);`
		`}`

removechild888888	0.825	`while (box.hasChildNodes()) {`
		`  box.removeChild(box.firstChild);`
		`}`

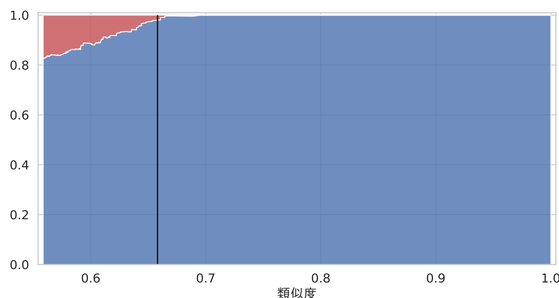

図 6 “for vs forEach” に分類されたプログラム断片に付与したラベルの割合

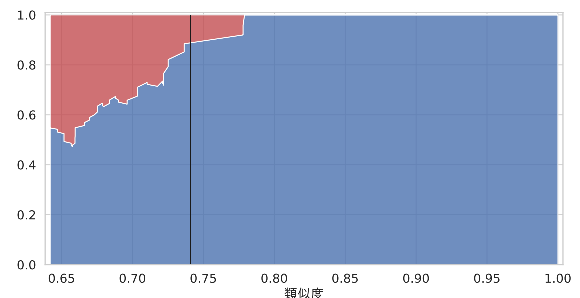

図 7 ”innerHTML vs removeChild”に分類されたプログラム断片に付与したラベルの割合

ことを確認した．しかし，分類されたプログラム断片の多くは細部が異なるのみで，実装自体はほぼ同一であった．特定の機能を実装する際に，存在する実装方法は限りがあるが，多くの開発者が同じ機能を実装するプログラム断片を比較するトピックを作成した結果，トピックのプログラム断片にほぼ一致するものが多くなったことが考えられる．今後は，一致するプログラム断片の除外や，類似した分類同士を統合することで，分類結果の再現率を高めるとともに，多様な実装の収集が容易になると考える．

6.2 制約

内的妥当性: 本論文では，プログラム断片のベクトル化に Code2Vec を使用している．Code2Vec はプログラムから特徴を抽出するツールが同封されており，Java と C#の 2 つのプログラミング言語が対応しているが，本論文の対象である JavaScript には対応していないため，JavaScript のプログラムから特徴を抽出するツールを自作した．そのため，Code2Vec で示されている精度であるとは限らない．しかし，目視による調査の結果，十分な精度が得られている．

外的妥当性: 閾値の決定に際し，マイクロベンチマークにリビジョンが存在していることを前提としている．したがって，リビジョンが存在していない場合，本手法では分類の基準とすることができない．また，本論文では評価対象を "for vs forEach" と "innerHTML vs removeChild" の 2 件に限定している．今後は，異なるトピックの分類についても調査する．

7. おわりに

本論文では，プログラム断片の類似度に基づいて，類似するマイクロベンチマークを分類する手法を提案した．また，分類することで特定の機能を実装する多様な方法を収集・比較できるか分析した．Web 上でマイクロベンチマークの結果を公開するサービスである jsPerf を対象に，マイクロベンチマークの収集, 分類を行い，マイクロベンチマーク "for vs forEach" と "innerHTML vs removeChild" を基準とする分類結果を分析によって得られた知見を述べる．

- 開発者は既にほぼ一致するプログラム断片がマイクロベンチマークに存在していても，同様なマイクロベンチマークを多数作成する．
- 機能的な類似度に基づいた分類を行うことで，実装が大きく異なるプログラム断片も分類できる．

開発者がマイクロベンチマークを用いてパフォーマンスの改善作業を行うためには，特定の機能を実装するプログラム断片の性能を比較する複数のマイクロベンチマークの中から最適なプログラム断片を選択する必要がある．本論文では，特定の機能を実装する多様な実装を収集・比較することで，開発者のパフォーマンス改善作業に合わせた提案をすることができると考える．また，分類方法の提案手法によって，機能的に類似したプログラム断片を分類し，開発者が実装しようとする特定の機能を実装する多様な実装を収集でき，開発者は収集した実装を比較することで適切な実装をすることができる．一方で，マイクロベンチマークの計測結果については着目していないため，今後はプログラム断片毎のパフォーマンスを明らかにすることで，開発者が特定の機能を実装するための，より性能効率の高い実装を開発者に提案手法を開発する．

謝辞 本研究は JSPS 科研費 18KT0013，18H03222 の助成を受けたものである．

参考文献

[1] Alon, U., Zilberstein, M., Levy, O. and Yahav, E.: code2vec: learning distributed representations of code, *Proceedings of the ACM on Programming Languages*, Vol. 3, No. POPL, pp. 1–29 (online), DOI: 10.1145/3290353 (2019).
[2] Avritzer, A. and Weyuker, E.: The automatic generation of load test suites and the assessment of the resulting software, *IEEE Transactions on Software Engineering*, Vol. 21, No. 9, pp. 705–716 (online), DOI: 10.1109/32.464549 (1995).
[3] Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y. and Flora, P.: Mining Performance Regression Testing Repositories for Automated Performance Analysis, *2010 10th International Conference on Quality Software*, IEEE, (online), DOI: 10.1109/qsic.2010.35 (2010).
[4] Luo, Q., Poshyvanyk, D. and Grechanik, M.: Mining performance regression inducing code changes in evolving software, *Proceedings of the 13th International Workshop on Mining Software Repositories (MSR'16)*, pp. 25–36 (2016).
[5] Zaman, S., Adams, B. and Hassan, A. E.: Security versus performance bugs, *Proceeding of the 8th working conference on Mining software repositories (MSR'11)*, pp. 93–102 (2011).
[6] 才木一也，安東亮汰，伊原彰紀：マイクロベンチマークサービスにおけるソフトウェアパフォーマンス改善方法の分析，電子情報通信学会技術研究報告，pp. 67–72 (2020).
[7] 日本産業規格：JIS X 25010:2013 システム及びソフトウェア製品の品質要求及び評価（SQuaRE）－システム及びソフトウェア品質モデル (2013).

ⓒ 2021 Information Processing Society of Japan