THE IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS (JAPANESE EDITION)

電子情報通信学会 論文誌 D 情報・システム

VOL. J103-D NO. 7
JULY 2020

情報・システムソサイエティ

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

論　文

軽量な類似度計算によるプロジェクト間のソースファイル集合の再利用検出

伊藤　薫[†a)]　石尾　隆[†,††]　神田　哲也[†]　井上　克郎[†]

Source File Set Reuse Detection between Projects with Lightweight Similarity Calculation

Kaoru ITO[†a)], Takashi ISHIO[†,††], Tetsuya KANDA[†], and Katsuro INOUE[†]

あらまし　ソフトウェア開発の現場において，オープンソースソフトウェアのソースコードをコピーして再利用することが一般的に行われている．ソフトウェアの再利用は，独自に開発した場合と比べて品質を向上させるが，プロジェクトの開発期間が長くなるにつれ，どこから，どのバージョンをコピーしたのかという情報が失われてしまうことがある．そこで本研究では，分析対象ソフトウェアのソースファイルと再利用したライブラリの版管理システムのリポジトリの内容を比較し，再利用したバージョンを自動的に検出する手法を提案する．具体的には，局所性鋭敏ハッシュ(LSH) を用いた高速なファイル単位での類似度計算を導入し，ファイル単位の類似度の合計をライブラリのバージョン単位での類似度とし，最も類似度の高いバージョンを再利用元として検出する．再利用情報が記録されているオープンソースソフトウェアをデータセットとして提案手法を適用した結果，99.3%の割合で利用しているライブラリのバージョンを正しく検出することを確認した．

キーワード　b-bit MinHash，オープンソースソフトウェア，再利用分析，ソフトウェアリポジトリマイニング

1. まえがき

ソフトウェア開発の現場において，オープンソースソフトウェア（Open Source Software，以降 OSS）を再利用することが一般的に行われている [1]．OSS の再利用は，独自に開発した場合と比べて品質や信頼性の向上，機能の開発期間の短縮などの効果をもたらす [2]．

ソフトウェア開発者は，しばしば，OSS として作られたライブラリのソースファイルを再利用して自身のソフトウェアに取り込む．ソースファイルは，常にそのまま使われるわけではなく，例えば Mozilla が開発している Gecko Engine では，ファイル圧縮に関するライブラリである zlib をコピーしたのち，独自のマクロを使用するよう zconf.h ファイルに記述を追加している．また，Google が開発するスマートフォン用 OS である Android のバージョン 8.0.0 は画像ファイルを扱うライブラリである libpng を再利用しており，ファイルの一つ pngpread.c にバグ修正のパッチを適用した状態で使用している．

再利用したライブラリのバージョン番号は，ソフトウェアの保守において重要な情報である．再利用したライブラリに含まれる脆弱性や欠陥の影響を回避するために，開発者はライブラリを新しいバージョンに更新するか，修正パッチを適用しなければならない [3]．脆弱性や欠陥の報告において，その影響範囲はバージョン番号によって記述されることから，影響の有無を把握し適切な更新作業を実施するために，正しいバージョン番号を把握することが必要となる．

しかしながら，プロジェクトの開発期間が長くなるにつれ，ソフトウェアをどこから再利用したのか，ソフトウェアのどのバージョンを再利用したのかという情報が失われてしまうことがある [4]．全てのソースファ

† 大阪大学大学院情報科学研究科，吹田市
Graduate School of Information Science and Technology, Osaka University, 1–5 Yamadaoka, Suita-shi, 565–0871 Japan
†† 奈良先端科学技術大学院大学先端科学技術研究科，生駒市
Graduate School of Science and Technology, Nara Institute of Science and Technology, 8916–5 Takayama-cho, Ikoma-shi, 630–0192 Japan
a) E-mail: ito-k@ist.osaka-u.ac.jp
DOI:10.14923/transinfj.2019JDP7077

イルについて内容を比較すると高コストだが，ソースファイルに編集が加えられていなければ，SHA-1 などのハッシュ値を各ソースファイルに対して計算することで，ソフトウェア中のソースファイルと一致するソースファイルをもつライブラリのバージョンを簡単に検出できる．しかし，先に紹介した例のように編集が加わっている場合には，単純な同一性の判定だけではバージョン情報を復元することができない．

本研究では，分析対象ソフトウェアのソースファイル集合とライブラリのバージョンごとのファイル集合との間の類似度を比較することで，分析対象のソフトウェアが利用しているライブラリのバージョンを検出する手法を提案する．本手法は，既存手法 [5] で用いたファイル間の類似度の考え方をファイル集合に拡張し，ファイル集合間で最も似ているファイルの組における類似度の合計を集合間の類似度とすることで，ファイル集合全体として最も類似するライブラリのバージョン一つを自動的に検出する．また，局所性鋭敏ハッシュ（Locality Sensitive Hashing，以降 LSH）の一つである b-bit MinHash 法を用いたファイル比較により，ファイル間の類似度の高速な計算を実現する．この手法を用いれば，ソフトウェア開発者は再利用したライブラリの更新が必要かどうか，ソースコードからバージョン番号を高速に復元し，判断できるようになる．

本論文は著者らの発表内容 [6] を拡張したものである．発表 [6] では b-bit MinHash による類似度の推定値を真の類似度の計算の枝刈りのためだけに使用していたが，本論文では類似度の推定値をそのまま合算することでバージョンの検出に用いる．また，ライブラリの名称はディレクトリ名等から判断ができる状況を想定し，既存手法 [5] との比較実験を追加した．本論文の貢献は以下のとおりである．

- b-bit MinHash による類似度の推定値を直接合算することでソースファイル集合の再利用元を高速かつ正確に検出する手法を提案した．
- 提案手法が実用的であることをオープンソースソフトウェアにおける実際の再利用の事例を用いた評価実験で確認した．ソフトウェアの全てのバージョンについてライブラリのどのバージョンを再利用しているかを調べる場合，平均で 104 秒程度の処理時間で，99.3%の割合で再利用しているライブラリのバージョンを正しく判定することができた．従来手法 [5] では最大で数時間の計算が必要であったが，それを大幅に短縮し，同時に正確さも向上している．

以降，**2.** では研究の背景について述べ，**3.** では提案手法を詳しく説明する．**4.** では提案手法の評価を行い，**5.** で妥当性への脅威について述べる．最後に，**6.** でまとめを述べる．

2. 背　　景

2.1 起源分析

起源分析とは，ソフトウェア内に存在するソースコードから，その作成に使われた（コピー元となった）他のソフトウェアやライブラリなどの起源を分析する手法である．コピーされたソースコードは，編集されていなければその起源を辿ることは容易いが，その後それぞれのソフトウェアに合わせて手を加える場合があり，ソースファイルの類似性に着目した分析が行われている．

プロジェクト間でのソースファイルの再利用を検出するために，コードクローン検出と呼ばれる互いに同一または類似するコード片の組を検出する手法が適用されている．German らは，複数のオープンソースプロジェクト間のコードクローンについて調査し，互いにどの程度コードクローンをもつのか分析することで，そのコードクローンの起源となるプロジェクトを特定した [7]．Inoue らは，インターネット上の OSS 中から，対象とするコード片に対するコードクローンを検出し，それらを時系列に並べることでその利用の変遷を表示する Ichi Tracker を提案した [8]．また，Sasaki らは，動作に影響のないコメントや空白を削除した場合のコードクローンに着目し，BSD 系 OS の一種である FreeBSD で利用されている OSS 同士でどのようにソースコードが再利用されているか調査した [9]．Lopes らは，版管理システムの一種である Git のオンラインホスティングサービスである GitHub 上で管理されているオープンソースプロジェクトについて，プロジェクト間でどのくらい同一のソースファイルが使われているか，どのようにコードクローンが分布しているか分析した [10]．

ソースファイルに含まれるバグや脆弱性の問題を調査するためには，ソフトウェアだけでなく，どのバージョンを使っているかまで特定することが必要である．そこで Kawamitsu らは，分析対象のソフトウェアとライブラリのリポジトリを入力として，ソフトウェアのリビジョンごとにソースファイルがどのライブラリのバージョンから再利用されているか最長一致部分列を元に分析する手法を提案した [5]．Ito らは，

GNU/Linux OS の一種である Debian が公開しているパッケージ集合をデータベース化し，入力されたソースファイルと最も類似するソースファイルを検索するウェブサービスを提案した [11]．これらの手法はファイルごとに再利用元の候補となるバージョンを報告するが，複数のバージョンに同一のファイルが含まれることもあるため，ソフトウェアが再利用したライブラリ全体としてのバージョンは使用者が判断しなければならない．

著者らは，入力をファイル単位からソースファイルの集合へと拡張し，データベース中から最も多く類似するソースファイルを含む OSS を検索する手法を提案した [6]．Jewmaidang らは，ソフトウェアのリポジトリとライブラリのリポジトリを比較することで，ソフトウェアが利用しているライブラリのバージョンの変遷を可視化した [12]．

2.2 類似度計算の高速化手法

起源分析においては，ファイル同士の類似度を高速に計算することが重要となる．Kawamitsu ら [5] は類似度としてファイル同士の最長一致部分列の長さを使用しているが，その計算には両方のファイルの長さの積に比例した時間が必要であり，様々な最適化を行った状態でも，合計数千万行のリポジトリの組の分析に最大で 4 時間程度かかることを報告している．

類似度の計算を高速化する方法として，ソフトウェアから類似したコード片の組を検出するコードクローン検出技術では，比較すべき候補を事前に効率良く絞り込む手法が適用されている．Jiang らは，ソースコードから作成した木構造の断片について，LSH を用いてクラスタリングすることで高速にコードクローンを検出する手法を提案した [13] また，横井らは，ソースコードから作成した TF-IDF ベクトルについて，cross-polytope LSH を用いてクラスタリングし，コードブロック単位でのコードクローンを検出する手法を提案した [14]．Sajnani らは，転置インデクスを使って比較すべき候補を絞ることで，効率的にコードクローンを検出する SourcererCC を提案した [15]．しかし，これらの手法は，互いに類似したソースコード片が少ないことを仮定したものであり，起源分析のように，多数の類似ファイルから最も類似したファイルを選択したいという目的には適さない．

2.3 *b*-bit MinHash 法を用いた類似度計算

類似度の計算そのものを高速化する手法として，類似度の一つである Jaccard 係数 [16] には，その推定値を高速に求める MinHash 法 [17], [18] が提案されている．Jaccard 係数は集合間の類似性を計測する指標であり，以下の式で表される．

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

ここで S_1, S_2 は，それぞれファイルの内容を表現した集合である．MinHash 法は，任意のハッシュ関数 h_i が与えられたとき，以下のように集合 S からハッシュ値が最小である要素を取り出す関数 $m_i(S)$ を使用する．

$$m_i(S) = \min_{s \in S} \ h_i(s)$$

二つの集合 S_1, S_2 に対して $m_i(S_1), m_i(S_2)$ を計算すると，$m_i(S_1) = m_i(S_2)$ となる確率が $J(S_1, S_2)$ に一致する．そのため，複数のハッシュ関数を用意すれば，これらのハッシュ値の一致割合から，Jaccard 係数の推定値を求めることができる．

MinHash 法を省メモリで実現する方法として b-bit MinHash 法 [19] が提案されている．この手法では，MinHash 法で計算したハッシュ値 $m_i(S)$ の下位 b ビットのみをハッシュ値として用いる．例えば $b = 1$ のとき，b-bit MinHash 法のハッシュ関数 $b_i(f)$ は以下の式で表される．

$$b_i(S) = \mathrm{LSB}(m_i(S))$$

ここでの LSB は Least Significant Bit を表す．

二つの集合 S_1, S_2 に対して $b_i(S_1)$ と $b_i(S_2)$ が一致する確率 $P(S_1, S_2)$ は，集合間の Jaccard 係数と，偶然ハッシュ値の下位 b ビットが一致する確率の和となる．ハッシュ値の下位 b ビットが偶然一致する確率は $\frac{1}{2^b}$ であるので，$P(S_1, S_2)$ は以下の式で表される．

$$P(S_1, S_2) = \frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right) \times J(S_1, S_2)$$

最も時間計算量，空間計算量ともに少なくなるのが $b = 1$ のときであり，k 個のハッシュ関数を用いても k ビットの記憶域で済む．$P(S_1, S_2)$ の近似値として，k 個のハッシュ値の一致割合 $P_o(S_1, S_2)$ を用いて，前述の式を変形した以下の式により集合 S_1, S_2 間の Jaccard 係数の推定値 $J_e(S_1, S_2)$ を計算することが可能である．

$$J_e(S_1, S_2) \ = \ \left(P_o(S_1, S_2) - \frac{1}{2}\right) \times 2$$

$$P_o(S_1, S_2) = 1 - \frac{1}{k}\sum_{i=1}^{k} \mathrm{XOR}(b_i(S_1), b_i(S_2))$$

比較したいソースファイルをそれぞれ k ビットの列に変換すれば，ファイル間の相互の比較はそれらのビット列の XOR 演算によって高速に実行することができる．

3. 提案手法

提案手法は，分析対象のソフトウェアにおいてライブラリから再利用したファイルを格納しているディレクトリ D_Q とそのライブラリの版管理システムのリポジトリ L を入力とし，D_Q に再利用されているライブラリのバージョン r_Q と，ディレクトリ内のファイル $q \in D_Q$ それぞれに最も類似したライブラリのバージョン $r[q]$ を出力する．入力をディレクトリとしたのは，再利用したライブラリのファイル群が一つのディレクトリにまとめて配置されることが多いためである．再利用されたバージョンの情報を D_Q 全体に対して一つ出力すると同時にファイルごとにも個別情報を出力するのは，全体としては古いバージョンのライブラリを利用している場合でも，例えば特定のファイルだけがセキュリティパッチの適用で新しいバージョンのものに差し替えられているといった状況を提示できるようにするためである．なお，ここでのバージョンとは，開発者がライブラリを対外的に公開する，版管理システムの中でタグと呼ばれるラベルが紐づけられているリビジョンのことである．

提案手法は，D_Q に含まれるソースファイルと，ライブラリの各バージョン $v \in L$ に含まれるファイル群を比較し，最も類似したファイルを含むようなバージョンを特定する．具体的には，ファイル間の類似度 $\mathrm{sim}(q, f)$ が与えられたとき，ライブラリのあるバージョン v に対する入力ファイル群の類似度の合計値 $S_Q(v)$ が最大となるような v を求める．入力ファイルの一つ q に対して，バージョン v の中で最も類似しているファイルの類似度 $S(q, v)$ は，以下の式で定義される．ただし，ファイル間の類似度は，あるしきい値 θ 以上の値の場合のみ記録し，θ 未満の場合は 0 とする．

$$S(q, v) = \begin{cases} S_{\max}(q, v), & \text{if } S_{\max}(q, v) \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

$$S_{\max}(q, v) = \max_{f \in \mathrm{Files}(v)} \mathrm{sim}(q, f)$$

ここで $\mathrm{Files}(v)$ は，バージョン v に含まれるファイルの集合である．このとき，$S_Q(v)$ の定義は以下のとおりである．

$$S_Q(v) = \sum_{q \in D_q} S(q, v)$$

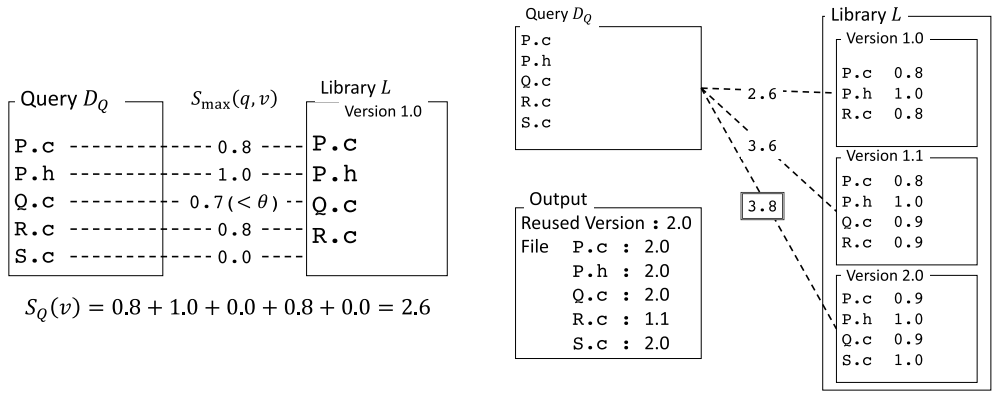
つまり，$S_Q(v)$ は $S(q, v)$ を合計することで，D_Q に使われているファイルのどれだけの内容が v に由来するものかを表す．

図 1 に提案手法の動作例を示す．この図は，Query D_Q として与えられた 5 個のファイルに対して，Library L の三つのバージョンのうちどれを再利用したのか特定しようとした場合を示している．図 1(a) は D_Q と L のある一つのバージョンとを比較する際の動作例を示している．D_Q に含まれる全てのソースファイルについて，$S_{\max}(q, v)$ を計算する．この例では類似度のしきい値を $\theta = 0.8$ としているため，Q.c では $S_{\max}(q, v) = 0.0$ となる．そのため，全体での類似度は $S_Q(v) = 2.6$ となる．この手順を L の全てのバージョンにおいて行った結果が図 1(b) である．$S_Q(v)$ が最大となるのは，Version 2.0 のときであり，これを D_Q が再利用しているライブラリのバージョンであると出力する．また，それぞれのソースファイルごとに，最も類似度が高いソースファイルが含まれるライブラリのバージョンも個別に出力する．この出力結果の場合，D_Q はライブラリ L の Version 2.0 を再利用している可能性が高いが，R.c という 1 ファイルのみ異なるバージョンが混ざった状態であることが判断できる．この情報から，開発者はライブラリの更新やパッチの適用などの活動を行うことができるようになる．

$S_Q(v)$ を計算するためには，入力されたディレクトリ D_Q と，ライブラリ L に所属するファイルの総当たり比較が必要となる．そこで，本研究では高速化のために b-bit MinHash 法を用いた類似度計算を採用する．以降，本手法でのファイル間での類似度の定義と，具体的な計算手順を示す．

3.1 ソースファイル間の類似度計算

本研究では，ソースファイルをそれぞれ字句の trigram 多重集合とみなし，それらの類似度として Jaccard 係数の推定値を b-bit MinHash 法で計算する．あるソースファイルの字句集合を f とし，その trigram 多重集合を $\tau(f)$ とすると，trigram 多重集合を用いたファイル f_1 とファイル f_2 との間の Jaccard 係数 $J_\tau(f_1, f_2)$ は以下の式で定義される．



(a) ある 1 つのバージョンとの比較（類似度の閾値 $\theta = 0.8$）　(b) ライブラリの全バージョンとの比較とその結果

図 1　提案手法の動作例

$$J_\tau(f_1, f_2) = \frac{|\tau(f_1) \cap \tau(f_2)|}{|\tau(f_1) \cup \tau(f_2)|}$$
$$\tau(f) = \{\langle t_i, t_{i+1}, t_{i+2}\rangle | 1 \le i \le |f| + 2\}$$

ただし，$\langle t_i, t_{i+1}, t_{i+2}\rangle$ は f 中の連続する三つの字句の組である．$\tau(f)$ を構築する際，f は先頭と末尾に空要素を二つずつ加えることで，全ての $\tau(f)$ の要素は少なくとも一つの空でない字句をもつ組になるようにする．字句解析はプログラミング言語ごとに異なるが，本論文の実験対象となる C 言語用の解析では，コメントと空白は削除するが，プリプロセッサのための指令（例えば #include など）はそれぞれ個別のトークンとして保持するようにした．なお，識別子などは特に正規化等を行わず，そのまま保持している．これらの情報はバージョンの特定に有用であることが報告されているためである [5].

しかしながら，正確に $J_\tau(f_1, f_2)$ を計算する場合，あらかじめ集合ごとに要素を特定の順序で整列しておいたとして，$\tau(f_1), \tau(f_2)$ の大きさの和に比例した時間が必要となる．そこで，これを軽量に計算するため，本研究では b-bit MinHash 法を用いて類似度を計算する．$b = 1$ とし，ハッシュ関数を k 個用いることで，k ビットのビットベクトルに対する XOR 演算を行うだけで，ファイル f_1, f_2 の組に対する類似度 $\mathrm{sim}(f_1, f_2) = J_e(\tau(f_1), \tau(f_2))$ を求める．

本研究では具体的なハッシュ関数の実装として，$\tau(f)$ における trigram τ の n 回目の出現に対して，以下の関数を適用する．(注1)

$$h_i(\tau, n) = \mathrm{SHA1}(\tau) \times n \times r_i$$

ここで SHA1(τ) は trigram τ を構成する三つのトークンを連結したバイト列に対する SHA-1 ハッシュ値であり，r_i は xorshift [20] による擬似乱数（ただし最小ビットは 1 に固定したもの）である．n と r_i で SHA-1 ハッシュの順序を並べ替え，$\tau(f)$ の全要素に対して最小となる値を保存し，得られたハッシュ値の下位 b ビットを得る．

b-bit MinHash 法によって得られる類似度 $\mathrm{sim}(f_1, f_2)$ は推定値であり，$J_\tau(f_1, f_2)$ に対して二項分布する [19]．その分散は $J_\tau(f_1, f_2)$ に応じて変化し，$J_\tau(f_1, f_2)$ がより高いほど小さくなり，1 のとき 0 になる．言い換えると，ある二つのソースコード間の $J_\tau(f_1, f_2)$ と $\mathrm{sim}(f_1, f_2)$ の差分は 0 を中心とした二項分布となり，$J_\tau(f_1, f_2)$ が大きいほどその差分も小さい．このことから，十分な要素数をもつソースファイル集合同士であれば，$J_\tau(f_1, f_2)$ を定義どおりに計算した場合と比べても，十分正確に類似度の合計値を求めることが期待できる．

図 2 に定義どおりに $J_\tau(f_1, f_2)$ を計算する具体例を，図 3 に b-bit MinHash 法を用いて類似度の推定値を計算する具体例を示す．図 2 中の ⟨ A, B, C ⟩ は連続する字句の組を表し，_ は空要素を表す．また，⟨ A, B, C ⟩^{u} となっている要素は，その集合に固有の要素である．ここから $J_\tau(a, b) = \frac{11}{19} = 0.579$ が得られる．図 3 は 10 個のハッシュ関数を用いて b-bit MinHash 法を適用した場合を示す．それぞれのソースコードの trigram 多重集合に対して 10 個のハッシュ関数 b_i を適用し，図に示すようなハッシュ値の列が得られた

(注1)：実装は https://github.com/NAIST-SE/CodeHash で公開している．

```
Example code:
  a: while ((*dst++ = *src++) != '\0');
  b: while (*dst++ = *src++);
```

$\tau(a)$	$\tau(b)$
⟨_, _, while⟩,⟨_, while, ⟨⟩,	⟨_, _, while⟩,⟨_, while, ⟨⟩,
⟨while, (, ⟨⟩^{u},⟨⟨, (, * ⟩^{u},	⟨while, (, *⟩^{u},
⟨⟨, *, dst⟩,⟨*, dst, ++⟩,	⟨⟨, *, dst⟩,⟨*, dst, ++⟩,
⟨dst, ++, =⟩,⟨++, =, *⟩,	⟨dst, ++, =⟩,⟨++, =, *⟩,
⟨=, *, src⟩,⟨*, src, ++⟩,	⟨=, *, src⟩,⟨*, src. ++⟩,
⟨src, ++,)⟩,	⟨src, ++,)⟩,
⟨++,), !=⟩^{u},⟨⟩, !=, '\0'⟩^{u},	⟨++,), ;⟩^{u},
⟨!=, '\0',)⟩^{u},⟨'\0',), ;⟩^{u},	
⟨⟩, ;, _⟩,⟨;, _, _⟩	⟨⟩, ;, _⟩,⟨;, _, _⟩

$$J_\tau(a,b) = \frac{|\tau(a) \cap \tau(b)|}{|\tau(a) \cup \tau(b)|} = \frac{11}{19} = 0.579$$

図 2　正確な $J_\tau(f_1, f_2)$ の具体例

入力	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}
$\tau(a)$	0	1	1	0	0	0	1	1	0	1
$\tau(b)$	0	1	1	0	0	1	0	1	0	1

$$\mathrm{sim}(a,b) = \left(\frac{8}{10} - \frac{1}{2}\right) \times 2 = 0.6$$

図 3　b-bit MinHash を用いた類似度計算例

とすると，一致しているハッシュ値は八つあるので，$\mathrm{sim}(a,b) = (\frac{8}{10} - \frac{1}{2}) \times 2 = 0.6$ となる．この例では 0.021 の誤差が生じているが，実際の適用ではハッシュ関数の個数を増やすことで，より正確に類似度を計算する．

3.2　再利用元バージョンの特定

Algorithm 1 に提案手法の疑似コードを示す．入力は再利用元ライブラリのバージョンを調査したいファイル集合 D_Q，そのソフトウェアが再利用しているライブラリのリポジトリ L である．似ていないファイルの類似度がノイズとして加わることを防ぐため，類似度がしきい値 θ 以上のファイルの組だけを扱う．また，b-bit MinHash 法の誤差を許容する範囲として m を導入している．加えて，計算コストを削減するために，ソフトウェアとライブラリそれぞれに含まれるファイルについて，事前に b-bit MinHash 法のためのハッシュ列やファイルの SHA-1 ハッシュ値を計算しておく．

アルゴリズムは 2 行目から 20 行目までが，ライブラリ L の全てのバージョン $v \in L$ に含まれるファイル f について，入力されたファイル $q \in D_Q$ と比較するループ構造となっている．計算を高速化するため，まず 5 行目でファイルが同一の拡張子をもつ（同一のプログラミング言語である）ことを確認し，次いで 6 行目でファイルの内容が完全一致するかどうかを SHA-1 ハッシュによって確認する．ファイルが完全一致していれば，字句解析などを一切行うことなく，類似度を 1.0 とする．完全一致でないファイルに対しては，9 行目で trigram 多重集合の大きさの比較を行い，大きさがあまりに異なる（類似度が決してしきい値 θ を超えない）ファイルについては比較を行わないようにする．この条件式は，二つの集合の大きさの比がしきい値 θ よりも小さい場合，Jaccard 係数が以下のように θ を上回ることがないことを利用している．

$$J_\tau(q,f) \le \frac{\min(|\tau(q)|, |\tau(f)|)}{\max(|\tau(q)|, |\tau(f)|)} < \theta$$

類似度がしきい値 θ を超える可能性があるファイルの組に対してのみ，10 行目で b-bit MinHash 法による類似度の計算を行う．$\mathrm{sim}(q,f)$ は Jaccard 係数を中心として二項分布するため，$\mathrm{sim}(q,f)$ が θ を超えないソースファイル間であっても，$J_\tau(q,f)$ は θ を超え

Algorithm 1 再利用しているライブラリのバージョンの検出

入力
　D_Q：解析対象ソフトウェアのあるバージョンでの再利用ライブラリが含まれるディレクトリ
　L：ソフトウェアが利用しているライブラリのリポジトリ
　θ, m：類似度のしきい値
出力
　r_Q：D_Q 中のソースファイルに最も類似する L のバージョン
　$r[q]$：D_Q 中のソースファイル q に最も類似するファイルをもつ L のバージョン

```
 1: Initialize S(q, v) ← 0 for all possible q ∈ D_Q and v ∈ L
 2: for v ∈ L do
 3:     for f ∈ Files(v) do
 4:         for q ∈ D_Q do
 5:             if f is same file extension to q then
 6:                 if SHA1(f) = SHA1(q) then
 7:                     S(q, v) ← 1.0
 8:                 else
 9:                     if min(|τ(q)|,|τ(f)|) / max(|τ(q)|,|τ(f)|) ≥ θ then
10:                         if sim(q, f) ≥ θ − m then
11:                             if sim(q, f) ≥ S(q, v) then
12:                                 S(q, v) ← sim(q, f)
13:                             end if
14:                         end if
15:                     end if
16:                 end if
17:             end if
18:         end for
19:     end for
20: end for
21: r_Q ← arg max_{v∈L} Σ_{q∈D_q} S(q, v)
22: r[q] ← arg max_{v∈L} S(q, v) for each q ∈ D_Q
```

ている場合がある．そこで，マージン m だけしきい値を引き下げることで，類似しているソースファイルの組合せが見落とされないようにする．

類似度の計算が完了すると，アルゴリズムの 21 行目でバージョンごとに類似度の合計を行い，最も類似度の合計値が高かったライブラリのバージョンを出力 r_Q として特定する．また，22 行目で，D_Q に含まれたファイル q それぞれに対して最も類似しているファイルを保有するバージョン $r[q]$ を求める．なお，同一のファイルを含むライブラリのバージョンが複数存在する場合があるが，r_Q の出力では，類似度が最大の候補が複数ある場合には最新のバージョンを報告する．$r[q]$ の候補が複数ある場合，その中に r_Q が含まれている場合は r_Q を使用し，そうでない場合は最新版を報告する．

4. 評　　価

本研究では提案手法の妥当性を以下の基準で評価する．

（1） ライブラリのバージョンを検出する正確さ

（2） ライブラリのバージョン検出に要する時間

ファイル単位で再利用元のバージョンを検出する Kawamitsu らの手法 [5] を基準としてみた場合，提案手法の特徴は，再利用されたファイル集合全体での類似度をもとにバージョンを検出すること，b-bit MinHash 法による高速な類似度の計算を導入したことの 2 点にある．そこで，正確な Jaccard 係数の計算を用いてファイル単位で再利用元バージョンを検出する場合を基準として，ファイル集合全体での類似度をもとにバージョンを検出する効果と，Jaccard 係数を b-bit MinHash 法で計算した効果を計測する．

4.1 実 験 対 象

実験対象として五つのソフトウェアと四つのライブラリのプロジェクトを使用する．表 1 にこれらのソフトウェアと利用されているライブラリの組合せを示す．実験対象ソフトウェアは，再利用ライブラリのバージョンを記録していること，長期間保守されていること，規模が異なること，開発コミュニティが異なることを基準として選択した．五つのソフトウェアは全て libpng と zlib を利用しており，合計で 13 組の再利用関係が含まれている．

表 1　実験対象のソフトウェア・ライブラリの組合せ

ソフトウェア	libpng	curl	ogg	zlib
android	✓	✓	✓	✓
apitrace	✓			✓
fs2open	✓			✓
gecko-dev	✓		✓	✓
v8monkey	✓			✓

表 2 にこれらのソフトウェア及びプロジェクトの開発状況の情報を示す．ID が 1 から 8 のプロジェクトはライブラリを利用しているソフトウェアで，ID が 9 から 12 のプロジェクトはライブラリである．全てのプロジェクトは分散版管理システムの Git で管理されており，表中のリポジトリ URL からクローンした．android は機能部位単位で Git リポジトリを分けて管理しているため，元になったライブラリに対応するリポジトリを実験対象としている．コミット数はリポジトリをクローンした時点での最新のリビジョンまでを数えたもので，LOC は最新のリビジョンでの総ソースコード行数 (Lines of Code) である．バージョン数はタグとして付けられたバージョンの個数である．同一リビジョンに複数のバージョンが紐づけられている（同一内容のソースコードが複数のバージョンとしてリリースされた）場合は，それらをまとめて一つのバージョンとして扱う．

4.2 実 験 方 法

提案手法を Java で実装し，表 1 のソフトウェア・ライブラリの組に対して実行する．実験対象ソフトウェアの各バージョンに対して，ライブラリから再利用したファイルを格納しているディレクトリ (D_Q) をディレクトリ名に基づいて特定し，ライブラリのリポジトリ (L) と合わせて実装プログラムに与えて，バージョン番号を取得する．提案手法における類似度のしきい値は，$\theta = 0.9$，$m = 0.1$ とし，b-bit MinHash 法のビット数とハッシュ関数の個数は $b = 1$，$k = 2048$ とした．

検出結果の正確さは，実験対象ソフトウェアの各バージョンに対して，検出結果がソフトウェア側の記録と一致した割合によって求める．ソフトウェア側の記録は，リポジトリ中のコミットメッセージやドキュメントファイルの中に記録されていた再利用ライブラリのバージョン番号を手作業で検出したものである．この手作業での分析では，ライブラリからコピーされたファイルは利用せず，実験対象ソフトウェアで作成された記録のみを用いてバージョンの検出を行った．バージョン番号の識別には Git リポジトリにおけるタグを用いたが，gecko-dev と ogg の組においては，再利用しているバージョンの記録に ogg 側の Subversion

表 2　実験対象のソフトウェア・ライブラリを格納するリポジトリ

ID	リポジトリ名	リポジトリ URL	期間	コミット数	LOC	バージョン数
1	android(libpng)	https://android.googlesource.com/platform/external/libpng	2008/10-2018/4	378	88401	46
2	android(curl)	https://android.googlesource.com/platform/external/curl	2010/3-2018/6	205	208530	22
3	android(ogg)	https://android.googlesource.com/platform/external/libogg	2012/6-2017/10	21	3283	9
4	android(zlib)	https://android.googlesource.com/platform/external/zlib	2008/10-2018/1	259	38057	35
5	apitrace	https://github.com/apitrace/apitrace.git	2008/7-2018/2	4264	413558	9
6	fs2open	https://github.com/scp-fs2open/fs2open.github.com.git	2002/1-2018/3	16526	776585	319
7	gecko-dev	https://github.com/mozilla/gecko-dev.git	1998/3-2018/4	634881	18053155	98
8	v8monkey	https://github.com/zpao/v8monkey.git	2007/3-2012/2	87432	6604218	72
9	libpng	git://git.code.sf.net/p/libpng/code	2009/4-2018/3	5956	87744	1557
10	curl	https://github.com/curl/curl.git	1999/12-2018/7	23312	196098	179
11	ogg	https://github.com/xiph/ogg.git	2000/9-2018/4	497	3830	11
12	zlib	https://github.com/madler/zlib.git	2011/9-2017/1	419	35609	72

表 3　提案手法の実行結果

ソフトウェア	ライブラリ	ソフトウェアのバージョン数	記録と一致する数		記録と一致しない数		記録なし	
android(libpng)	libpng	46	46	100.0%	0	0.0%	0	0.0%
android(curl)	curl	22	19	86.4%	3	13.6%	0	0.0%
android(ogg)	ogg	9	9	100.0%	0	0.0%	0	0.0%
android(zlib)	zlib	35	35	100.0%	0	0.0%	0	0.0%
apitrace	libpng	9	8	88.9%	0	0.0%	1	11.1%
	zlib	9	9	100.0%	0	0.0%	0	0.0%
fs2open	libpng	293	293	100.0%	0	0.0%	0	0.0%
	zlib	293	293	100.0%	0	0.0%	0	0.0%
gecko-dev	libpng	98	98	100.0%	0	0.0%	0	0.0%
	ogg	98	94	95.9%	0	0.0%	4	4.1%
	zlib	98	98	100.0%	0	0.0%	0	0.0%
v8monkey	libpng	72	72	100.0%	0	0.0%	0	0.0%
	zlib	72	72	100.0%	0	0.0%	0	0.0%
全体		1,154	1,146	99.3%	3	0.3%	5	0.4%

リポジトリのリビジョン番号が使用されていたため，それらを手作業で Git リポジトリのバージョン番号に読み替えた．

ファイル単位で再利用元バージョンを検出する場合と比較するため，提案手法のファイル単位での出力 ($r[q]$) についても，ファイル単位でソフトウェア側の記録との一致率から正確さを求める．Kawamitsu らの手法 [5] に従ってファイル単位で最も類似したライブラリのバージョンが複数検出された場合は，ソフトウェア側の記録に該当するバージョンを含むとき記録と一致するとみなす．提案手法は，既存手法と比べると，ファイル集合全体での集計結果を用いて，特定の 1 バージョンだけに候補を絞った出力を行っているとみなすことができるため，出力されるバージョン数も比較を行う．

提案手法の実行時性能の評価は，実装プログラムをそれぞれ 10 回ずつ実行し，その実行時間を計測することで行う．入力されたソースファイル群の読み込みと比較に大きな時間を要するため，ソフトウェアの読み込み，ライブラリの読み込み，ソースファイルの比較のステップと全体での時間を分けて記録する．計測に用いる計算機環境の OS は Oracle Linux 4.1.12-112.16.7.el7uek.x86_64，CPU は Intel Xeon E5-2690 v4，RAM は DDR4-2400 ECC Memory 512 GB，ストレージは SAS 接続の 1TB のもの，Java の実行環境は OpenJDK 8 である．時間の計測は，Java のシステムメソッドの一つである nanoTime メソッドの呼び出しを実装プログラム中に記述することで行う．また，マルチスレッド処理は使用せず，単一のスレッドで処理を行う．

b-bit MinHash 法がもたらす効果を評価するため，評価実験では Jaccard 係数を b-bit MinHash 法を用いて近似計算した場合と，ファイルの内容から正確に計算した場合の 2 通りを実行する．

4.3　提案手法の正確さ

表 3 にソフトウェアのバージョンごとの正確さを計測した結果を示す．fs2open のバージョン数が表 2 のバージョン数と異なる値だが，これは一部のバージョンでライブラリが利用されていなかったためである．検出結果と記録はソフトウェアごとでは 86.4%から 100.0%が，全体では 99.3%が一致し，ソフトウェアとライブラリの組のうち，13 組中 10 組については検出結果と記録が全て一致した．記録と一致しなかった android における curl 再利用の三つのバージョンにつ

いては，再利用したライブラリのドキュメントファイルと検出結果のバージョンが一致したことから，ソフトウェア側の記録が間違っている可能性がある．また，再利用したバージョンの記録がない場合についても，apitrace の 1 例ではライブラリのドキュメントファイルに記述されているバージョン情報と提案手法の検出結果が一致している．これらのことから，提案手法は再利用しているライブラリのバージョンを高い精度で検出することが可能である．

表 4 に提案手法の全てのファイルに対して求めた全体での正確さを示す．既存手法，すなわち最も類似したファイルを選択する方式の正確さは 94.7%であり，提案手法は 95.6%でそれを上回った．また，既存手法ではファイルごとの情報だけでは単独のバージョンに絞り込むことはできず，最頻値で 4 バージョン，中央値で 23 バージョン，最大で 1612 バージョンを列挙した．提案手法は，全体での結果と各ファイルについて個別のバージョンが一致する場合はただ一つ，一致しない場合でも二つのバージョンだけに絞り込んでおり，それでも正確さで既存手法を上回っていることから，既存手法よりも簡潔な出力で，正確さを達成している．

表 4　ファイル単位での正確さ

ソフトウェア	ライブラリ	提案手法	既存手法
android	libpng	100.0%	98.4%
	curl	86.4%	86.4%
	ogg	100.0%	100.0%
	zlib	100.0%	99.6%
apitrace	libpng	90.7%	90.7%
	zlib	100.0%	100.0%
fs2open	libpng	100.0%	100.0%
	zlib	100.0%	100.0%
gecko-dev	libpng	100.0%	95.4%
	ogg	96.3%	95.5%
	zlib	100.0%	100.0%
v8monkey	libpng	100.0%	86.4%
	zlib	100.0%	100.0%
全体		95.6%	94.7%

表 5 に，提案手法におけるファイル単位での検出結果がソフトウェア全体の検出結果と異なったファイルの数を示す．表中のファイル数の列は，ソフトウェア中のライブラリのディレクトリに含まれるソースファイルの数を全てのバージョンで足し合わせた値である．提案手法は 1.61%のファイルに対して，ファイル集合全体でのバージョンとは異なるバージョンを出力した．それらのファイルを目視で分析したところ，以下のような事例が含まれていた．

- 利用しているライブラリの新しいバージョンでは削除されたファイル．新しいバージョンのライブラリを上書きコピーした結果，ライブラリ側で削除されたファイルだけが上書きされずに残ったと考えられる．
- ライブラリ側でバグや脆弱性の修正版応がリリースされる前に，修正パッチをソフトウェア側で独自に適用したファイル．そのファイルのみ，ライブラリの新しいバージョンとして検出された．
- ソフトウェア側で独自に変更を加えたファイル．例えば条件式の中で関数を呼ぶのではなく変数に代入してから条件式で参照するなどのコーディングスタイルの統一を実施したものがあった．変更内容が，他のバージョンの内容と偶然類似していた場合に，異なるバージョンとして報告された．

提案手法は全体として最も類似したバージョンを報告するため，そのバージョンを基準に差分を分析することで，これらの影響を迅速に把握することができた．

表 5　ソフトウェア全体とファイル単体での検出結果の違い

ソフトウェア	ライブラリ	バージョン数	ファイル数	不一致数		バージョンあたりの不一致数	
android(libpng)	libpng	46	3,211	50	1.56%	1.09	1.56%
android(curl)	curl	22	13,137	273	2.08%	12.41	2.08%
android(ogg)	ogg	9	63	0	0.00%	0	0.00%
android(zlib)	zlib	35	2,960	11	0.37%	0.31	0.37%
apitrace	libpng	9	194	0	0.00%	0	0.00%
	zlib	9	240	0	0.00%	0	0.00%
fs2open	libpng	319	6,446	0	0.00%	0	0.00%
	zlib	319	6,445	0	0.00%	0	0.00%
gecko-dev	libpng	98	2,520	116	4.60%	1.18	4.60%
	ogg	98	536	4	0.75%	0.04	0.75%
	zlib	98	2,646	1	0.04%	0.01	0.04%
v8monkey	libpng	72	1,586	215	13.56%	2.99	13.56%
	zlib	72	1,756	0	0.00%	0	0.00%
全体		1,206	41,740	670	1.61%	0.56	1.61%

これはファイル単位で検出を行っていた既存手法では実施できなかった分析である．

図 4 に，提案手法（b-bit MinHash 法）で計算した類似度と，正確な Jaccard 係数の誤差の分布を示す．左端の箱ひげ図が提案手法の実行中に計算された全ての類似度の値の誤差の分布である．b-bit MinHash 法はファイルの内容が同一（Jaccard 係数 1）であるとき差分は原理的に 0 となるので，ばらつきは非常に小さい．中央の箱ひげ図がこれらの類似度をファイル集合ごとに合計したときの誤差であり，最大でも 0.52 である．右端の箱ひげ図のようにファイルの個数で正規化すると，最大で 0.006 と，ソフトウェア全体の類似度に対する比率としては極めて小さな誤差であり，実際に本実験ではバージョンの検出には影響せず，提案手法は Jaccard 係数を正確に計算した場合と同一の結果を出力した．

4.4 提案手法の実行時性能

表 6 に提案手法の実行時間を計測した結果を示す．左から全体での実行時間と，実験対象ソフトウェアのソースファイルの読み込みに要した時間，ライブラリのソースファイルの読み込みに要した時間，比較に要した時間を示す．三つの段階以外にも出力の集計等の処理が含まれるため，三つの段階の合計時間は全体の合計時間とは一致しない．ソフトウェアの読み込み時間は実装の都合上全てのバージョンを読み込んだ時間である．読み込み時間はバージョン数と全バージョンを通してのユニークなファイル数が多いほど増大する．表 2 を見るとバージョン数はライブラリの方が多い傾向にあるため，ソフトウェアよりもライブラリの方が読み込み時間が長くなる傾向にある．

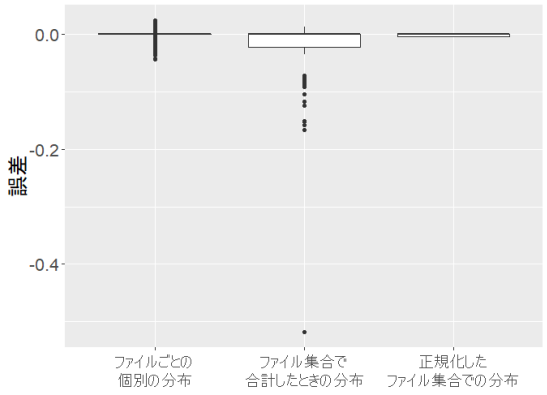


図 4　1 ビット MinHash 値と Jaccard 係数の誤差の分布

表 6 の「比較回数」の列は，提案手法における類似度の計算回数（Algorithm 1 における 10 行目の実行回数）を示している．類似度の計算回数は最大で 6.7×10^8 回行われているにもかかわらず比較時間は平均で 11 秒程度である．これは，提案手法における類似度の計算が一度のビット演算と算術演算だけであり，時間計算量が $O(1)$ となっているためである．

表 6 の右から 3 列目に，b-bit MinHash 法を利用しなかった場合（Jaccard 係数を直接計算した場合）の実行時間全体を 100%としたときの提案手法の実行時間を示す．ソフトウェア及びライブラリ読み込みに要する時間はどちらの方法でも変わらず，実行時間の短縮は比較に要する計算時間の差によるものである．また，右から 4 列目に，正確に Jaccard 係数を計算した場合の全体の実行時間を示す．これらから，比較回数が少ないものについては実行時間の短縮の効果が小さいが，平均でも 24.2%の実行時間で計算を完了できるようになったことが分かる．ライブラリが ogg の場合はほとんど改善がみられないが，これは ogg が Jaccard 係数を計算したとしてもほとんど時間がかからないほど小規模で比較回数が少なく，読み込みにかかる時間の方が支配的なためである．

表 6　提案手法の実行時間

ソフトウェア	ライブラリ	実行時間 全体 t_1 [ms]	ソフトウェア 読み込み [ms]	ライブラリ 読み込み [ms]	比較 [ms]	比較回数	類似度を正確に計算する場合との比較 正確に計算した場合の 実行時間全体 t_2 [ms]	t_1/t_2	使用メモリサイズ [MB] Jaccard 係数	b-bit MinHash
android	libpng	242,371	4,571	223,418	14,381	2.2×10^8	4,170,739	5.8%	944.46	4.32
	curl	95,864	9,064	45,515	41,287	6.7×10^8	3,220,414	3.0%	226.96	3.12
	ogg	981	601	357	22	3.2×10^3	1,044	94.0%	1.51	0.01
	zlib	7,250	2,275	4,342	632	1.2×10^7	105,303	6.9%	24.47	0.36
apitrace	libpng	214,544	1,480	212,289	773	1.5×10^7	545,570	39.3%	946.01	4.34
	zlib	6,156	928	5,147	80	1.0×10^6	16,602	37.1%	23.69	0.35
fs2open	libpng	279,272	2,012	210,973	66,286	5.0×10^8	11,871,778	2.4%	945.39	4.33
	zlib	9,045	1,335	5,064	2,645	2.8×10^7	300,693	3.0%	22.99	0.34
gecko	libpng	235,119	12,740	208,702	13,675	1.8×10^8	4,294,906	5.5%	958.16	4.40
	ogg	9,988	9,593	312	82	3.1×10^4	10,478	95.3%	23.08	0.34
	zlib	17,526	10,507	4,946	2,072	2.7×10^7	278,052	6.3%	1.71	0.01
v8monkey	libpng	231,638	7,603	214,594	9,440	1.7×10^7	3,000,782	7.7%	22.30	0.33
	zlib	12,141	6,007	5,048	1,085	1.2×10^8	178,727	6.8%	950.12	4.36
平均		104,761	5,286	87,747	11,727	1.4×10^8	2,153,468	24.1%	391.60	2.05

表 6 の右端の 2 列は，Jaccard 係数を計算する場合の集合表現を構築する際のメモリ使用量と b-bit MinHash 法を使用した場合のメモリ使用量の概算値である．Jaccard 係数を計算する場合は 1 ファイルの 1 要素ごとに 64 ビット整数を一つ使用する，b-bit MinHash の場合は 1 ファイルに対し 2048 ビット使用するとみなし，Java のデータ構造によるオーバーヘッドを無視して算出した値である．結果として，メモリの使用量は Jaccard 係数を計算する場合の 1%未満となり，大規模なライブラリのリポジトリに対する解析でも，通常の計算機で十分に取り扱うことができると考えられる．

上記の結果から，提案手法は既存手法よりも高速かつ省メモリである．また，ライブラリの読み込みには時間がかかるが，b-bit MinHash の計算結果をキャッシュとして保管しておけばその影響は小さくすることが可能である．提案手法を使用して，様々なプロジェクトに対して自動でライブラリの利用状況を監視するようなシステムの構築も十分可能であると考えている．

5. 妥当性への脅威

5.1 ファイル集合の再利用バージョン判定

本研究ではソフトウェア全体での再利用バージョンを決定する際，ファイル単位の類似度を合計する方法を用いた．本研究で用いた b-bit MinHash 法で計算した類似度は，文献 [19] によると誤差が 0 を中心に二項分布するため，類似度を合計することで，十分にファイル数が多い場合には正負の誤差が相殺され，平均で 0 に近づくことを期待した．評価実験においても，図 4 に示したように誤差を小さく抑えることができた．

ファイル単位の類似度を集計する方法はこれに限定されているわけではなく，例えばファイル単位でのバージョンの多数決を取るという方法も考えられる．この二つの類似度の集計方法は，類似度が以下の条件を満たす場合に，異なる結果を出力する．

$$|S(q,v_1) > S(q,v_2)| > |S(q,v_1) < S(q,v_2)|$$
$$S_Q(v_1) < S_Q(v_2)$$

このような状況は，ライブラリの再利用方法によって，v_1，v_2 のどちらが正解の場合にも起こり得る．類似度の合計を採用する場合では，再利用したファイルの一つを大幅に編集した結果，その類似度の差が支配的になり，実際に再利用したバージョンの過半数のファイルが完全一致であっても，間違ったバージョンを検出してしまう可能性がある．多数決を採用する場合では，再利用したファイルの過半数に対して編集した結果，実際に再利用したバージョンとの類似度が低下すると，他のファイルの類似度の値にかかわらず誤ったバージョンを検出してしまう．類似度の集計方法の違いが結果に影響する可能性はあるが，評価実験ではこのような状況は発生しなかった．ライブラリの再利用に際してファイルの修正は基本的には軽微であり，表 5 に示したとおり，全てのファイルのうち 1.61%以外はソフトウェア全体の検出結果のバージョンと最も類似していた．

5.2 実験対象の妥当性

実験対象のソフトウェアには，利用ライブラリのバージョンを適切に記録しているものだけを使用している．そのため，評価実験の結果には，それらのソフトウェアの特徴が影響している可能性がある．ただし，対象ソフトウェアは異なるコミュニティで開発されているものを選択したため，開発者の再利用方法に関する偏りの影響は少ないと考える．

実験対象の再利用状況を確認すると，ライブラリに含まれる全てのソースファイルを再利用する場合がほとんどだった．libpng については Android のみテストに使われるソースファイルなどは再利用していなかった．このようなライブラリの一部分のみを再利用する場合としては以下のものが考えられる．

（1） ライブラリ内の一部のファイルのみが再利用された場合．例えばテストファイルの除去が行われるなど．

（2） ファイルごとに異なるバージョンのライブラリからの再利用が行われた場合．例えば修正パッチを最新版から取り込んだ場合など．

（3） あるファイルの一部分のみが再利用された場合．例えば特定の関数内のアルゴリズムだけが流用される場合など．

一つ目の場合については，本手法は再利用バージョンの判定にしきい値を超えるような類似度をもつソースファイルを用いるので，改変が大きくない限り再利用バージョンの検出は可能である．二つ目の場合については，提案手法の章で述べたように，ファイル単位での再利用バージョンも出力するため，開発者はそれぞれのファイルについて対応方法を検討することが可能である．加えて，この事例は **4.3** に挙げたとおり，実験対象としたソフトウェアについても一部のファイルについて確認されており，提案手法によって簡単に

確認できている．三つ目の場合については，入力されたディレクトリ中の全てのファイルについて $S(q,v)$ がしきい値を下回るのであれば提案手法で検出することができない．しかし，そもそも提案手法の適用対象は再利用したライブラリのみが含まれるディレクトリであるため考慮しない．このような場合はコードクローン検出手法など，異なる分析手法を用いる必要がある．

6. むすび

本研究ではソフトウェアのファイル集合とライブラリのリポジトリを入力として，分析対象のソフトウェアが再利用しているライブラリのバージョンを検出する手法を提案した．ファイル単位の類似度をソースファイル集合で合計することで，99.3%の正確さで全体としてどのバージョンを再利用しているかを検出し，個別にどのファイルが変更されているかを分析可能とした．また，b-bit MinHash 法を採用することで，既存手法よりも高速かつ省メモリな計算を達成した．提案手法を用いることで，ソフトウェア開発者は再利用しているライブラリのバージョンを非常に高精度に特定することができ，その情報を用いてライブラリの更新などについて判断を下すことができるようになる．

今後の課題として，再利用されたファイルに施された修正を，自動的に既知のパッチ等と照合し，再利用されたファイルを効果的に分析する手法の確立が挙げられる．また，あるソフトウェアに使用されている全てのソースファイルから，再利用ライブラリを自動的に検出する技術への拡張も，今後の課題である．

謝辞 本研究は科学技術研究費 JP18H04094, JP18H03221, JP19K20239 の助成を受けて行われた．

文　献

[1] C. Ebert, “Open source software in industry,” IEEE Software, vol.25, pp.52–53, May 2008.

[2] P. Mohagheghi, R. Conradi, O.M. Killi, and H. Schwarz, “An empirical study of software reuse vs. defect-density and stability,” Proc. the 26th Int. Conf. on Software Engineering, pp.282–292, May 2004.

[3] L. Constantin, “Developers often unwittingly use components that contain flaws,” iTWorld.com. https://www.itworld.com/article/2936575/software-applications-have-on-average-24-vulnerabilities-inherited-from-buggy-components.html.

[4] P. Xia, M. Matsushita, N. Yoshida, and K. Inoue, “Studying reuse of out-dated third-party code in open source projects,” Computer Software, vol.30, no.4, pp.98–104, 2013.

[5] N. Kawamitsu, T. Ishio, T. Kanda, R.G. Kula, C.D. Roover, and K. Inoue, “Identifying source code reuse across repositories using LCS-based source code similarity,” Proc. the 2014 IEEE 14th Int. Working Conf. on Source Code Analysis and Manipulation, pp.305–314, Sept. 2014.

[6] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue, “Source file set search for clone-and-own reuse analysis,” Proc. the 2017 IEEE/ACM 14th Int. Conf. on Mining Software Repositories, pp.257–268, May 2017.

[7] D.M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, “Code siblings: Technical and legal implications of copying code between applications,” Proc. the 2009 6th IEEE Int. Working Conf. on Mining Software Repositories, pp.81–90, May 2009.

[8] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, “Where does this code come from and where does it go?—Integrated code history tracker for open source systems,” Proc. the 34th Int. Conf. on Software Engineering, pp.331–341, June 2012.

[9] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, “Finding file clones in FreeBSD ports collection,” Proc. the 7th IEEE Working Conf. on Mining Software Repositories, pp.102–105, May 2010.

[10] C.V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, “Déjàvu: A map of code duplicates on GitHub,” Proc. ACM Program. Lang., vol.1, no.OOPSLA, pp.84:1–84:28, Oct. 2017.

[11] K. Ito, T. Ishio, and K. Inoue, “Web-service for finding cloned files using b-bit minwise hashing,” Proc. the 2017 IEEE 11th Int. Workshop on Software Clones, pp.1–2, Feb. 2017.

[12] K. Jewmaidang, T. Ishio, A. Ihara, K. Matsumoto, and P. Leelaprute, “Extraction of library update history using source code reuse detection,” IEICE Trans. Inf. & Syst., vol.101-D, no.3, pp.799–802, March 2018.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” Proc. the 29th Int. Conf. on Software Engineering, pp.96–105, May 2007.

[14] 横井一輝，崔　恩瀞，吉田則裕，井上克郎，“情報検索技術に基づく細粒度ブロッククローン検出，” コンピュータソフトウェア，vol.35, no.4, pp.16–36, 2018.

[15] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, “SourcererCC: Scaling code clone detection to big-code,” Proc. the 38th Int. Conf. on Software Engineering, pp.1157–1168, May 2016.

[16] P. Jaccard, “The distribution of the flora in the alpine zone.1,” New Phytologist, vol.11, no.2, pp.37–50, 1912.

[17] A. Broder, “On the resemblance and containment of documents,” Proc. the Compression and Complexity of Sequences 1997, pp.21–29, June 1997.

[18] M.S. Charikar, “Similarity estimation techniques

from rounding algorithms," Proc. the 34th Annual ACM Symposium on Theory of Computing, pp.380–388, May 2002.
[19] P. Li and C. König, "b-bit minwise hashing," Proc. the 19th Int. Conf. on World Wide Web, pp.671–680, April 2010.
[20] G. Marsaglia, "Xorshift RNGs," Journal of Statistical Software, vol.8, no.14, pp.1–6, 2003.

（2019 年 10 月 2 日受付，2020 年 2 月 6 日再受付，
3 月 27 日早期公開）

伊藤　　薫

2016 兵庫県立大学工学部卒，2018 大阪大学大学院情報科学研究科博士前期課程了．現在大阪大学大学院情報科学研究科博士後期課程に在学中．ソフトウェアの再利用分析に関する研究に従事．

石尾　　隆　（正員）

2003 大阪大学大学院基礎工学研究科博士前期課程了．2006 同大学大学院情報科学研究科博士後期課程了．同年日本学術振興会特別研究員 (PD)．2007 大阪大学大学院情報科学研究科助教．2017 奈良先端科学技術大学院大学情報科学研究科准教授．2018 同大学先端科学研究科准教授．博士（情報科学）．プログラム解析，プログラム理解に関する研究に従事．

神田　哲也

2016 大阪大学大学院情報科学研究科博士後期課程了．同年奈良先端科学技術大学院大学博士研究員．2017 大阪大学大学院情報科学研究科特任助教．2018 より同研究科助教．博士（情報科学）．ソフトウェア進化，ソースコード解析に関する研究に従事．

井上　克郎　（正員：フェロー）

1984 大阪大学大学院基礎工学研究科博士後期課程了（工学博士）．同年大阪大学基礎工学部情報工学科助手．1984〜1986，ハワイ大学マノア校コンピュータサイエンス学科助教授．1991 大阪大学基礎工学部助教授．1995 同学部教授．2002 より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索等のプログラム分析や再利用技術の研究に従事．本会フェロー．