修士学位論文

題目

パッケージに対する b-bit MinHash を用いた バイトコード中のライブラリ検出手法

指導教員
井上 克郎 教授

報告者
伊藤 薫

平成 30 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 29 年度 修士学位論文

パッケージに対する b-bit MinHash を用いたバイトコード中のライブラリ検出手法

伊藤 薫

内容梗概

ソフトウェア開発において，ある機能の一部を実現するために，すでに存在するソフトウェアからソースをコピーして再利用するということは一般的に行われている．再利用する際にソフトウェアの名称やバージョンといった情報を正しく管理することができれば，既存のソフトウェアを再利用することで開発費用を削減することができる．しかし，実際の開発現場においては，必要な情報を記録せずに再利用していたり，ディレクトリ構成が変更されたり，再利用した部分の担当者が変わってしまうことがあるため，再利用した元のソフトウェアに関する情報が失われてしまうことがある．

そこで，本研究ではソフトウェアの再利用情報を復元することを目的として，Java で開発された Java アーカイブ形式のソフトウェアを対象に，内部に含まれるライブラリを検出する手法を提案する．Java において，ソフトウェアの再利用の際にはソースファイルではなく，コンパイラによって生成された Java バイトコードを含むクラスファイルを用いる．提案手法では，Java パッケージそれぞれにおいて，含まれるクラスファイルごとにシグネチャを計算し，b-bit MinHash 法によりパッケージのハッシュ値を計算する．それらの情報から再利用しているライブラリをデータベース中から検出する．クラスファイルのシグネチャをパッケージに関する情報を除外して計算することでパッケージリネームに対応し，かつ b-bit MinHash 法を用いることで，クラスファイルの多少の入れ替えに対応している．

提案手法の有効性を確認するために，再利用したライブラリを内部に含んでいるソフトウェアに対して提案手法を適用した．その結果，再利用されたと記録されているライブラリを特定できていることを確認した．また，既存手法よりも精度は若干低下するが，計算コストを削減することに成功した．

主な用語

ライブラリ検出
b-bit MinHash
Java バイトコード解析

目 次

1 はじめに **4**

2 ソフトウェア開発における外部ソフトウェアの再利用 **6**

2.1 ソフトウェアの再利用 6

2.1.1 Java におけるソフトウェアの再利用 6

2.1.2 パッケージリネーム 7

2.2 再利用分析 8

2.2.1 ソースコードを利用した再利用分析 9

2.2.2 Java における再利用分析 10

3 パッケージに注目したライブラリ検出手法 **12**

3.1 パッケージ間の類似度 12

3.2 LSH による類似集合の高速検索 13

3.2.1 MinHash 法 13

3.2.2 b-bit MinHash 法 14

3.3 クラスファイルのシグネチャ 14

3.4 パッケージに注目したハッシュ値 15

3.4.1 パッケージのシグネチャ 16

3.4.2 パッケージに対する b-bit MinHash の適用 16

3.5 再利用元の検出 16

3.5.1 共通するパッケージを持つライブラリの選択 18

3.5.2 組み合わせ探索 19

4 実験 **22**

4.1 実装 22

4.2 事前準備 22

4.2.1 データベースの構築 22

4.2.2 実験用 Java アーカイブファイルの作成 23

4.3 検出精度の評価 23

4.3.1 評価方法 23

4.3.2 結果 24

5 まとめ **28**

謝辞 **29**

参考文献 **30**

1　はじめに

ソフトウェア開発において，ある一部の機能を実装する際にすでに存在する他のソフトウェアからソースファイルを再利用することが一般的に行われている．開発者はソフトウェアを再利用することで開発コストを削減でき，かつ広く用いられているソフトウェアから再利用することで，独自に開発するよりも信頼性の高い機能を実装することができる．特に，ほかのソフトウェアで再利用することを目的として開発されたソフトウェアに，ライブラリと呼ばれるものがある．

ライブラリはある特定の機能に特化したソフトウェアであり，可搬性が高くなるように設計されていて，再利用が容易である．現在数多くのライブラリがインターネット上でオープンソースソフトウェアとして公開，配布されている．オープンソースソフトウェアは，インターネットを介して様々な開発者が実装を行っておりかつ利用者も多いため，独自に開発した機能と比べてバグや脆弱性の発見が迅速に行われる．結果として信頼性が高い状態が保たれやすく，企業での製品開発でも活用されることがある [6][7].

オープンソースライブラリは信頼性と可搬性が高い反面，脆弱性が見つかった際に利用中のライブラリをバージョンアップするかパッチファイルをあてる更新作業が必要となる．再利用した際に正確にライブラリ名とそのバージョンについて記録を行い，その後も管理されていれば更新作業も容易に行えるが，全てのソフトウェアで管理が行き届いているわけではないことが判明している [19].

更新作業を行う際に再利用した情報が無ければ，その情報をソースコードなどから復元する必要がある．再利用情報を復元する研究は多数行われており，再利用分析と呼ばれる．ソースコードが利用可能な場合，コードの一部分を入力として類似するソースコードの一部をインターネット上から検索する Inoue ら [10] の Ichi Tracker や，ファイルを入力として再利用したと思われるライブラリの各バージョンから再利用元と推定されるファイルを検出する川満ら [13] の手法などが研究されている．また，坂口ら [23] によって，同じライブラリから再利用していると思われるファイル群を検索クエリとして，ファイルごとのソースコードの類似度をもとにデータベース中からライブラリの候補を検出する手法なども開発されている．

本研究で対象としている Java では，クラスファイルと呼ばれるコンパイラにより生成されたバイナリファイルの一種であるバイトコードファイルを再利用することが一般的である [8]．Java のソフトウェアは Java アーカイブファイル（Jar ファイル）で配布され，Java ファイルをコンパイルする際に利用するライブラリの Jar ファイルをオプションで指定することで，ライブラリに含まれるクラスを利用したソフトウェアを作成できる．このような特徴から，Java における再利用分析では Jar ファイルを入力とする手法が提案されている．Davies

ら [5] の Software Bertillonage では，入力された Jar ファイル中に含まれるクラスファイルと，データベースに登録されているライブラリの持つクラスファイルの一致数をもとにソフトウェア間の類似度を求め，最も類似度の高いライブラリを再利用元として検出する．しかし，Java では複数のライブラリを再利用することも多く，その場合は Davies らの手法では対応できない．

Jar ファイル中から再利用した複数のライブラリを検出する手法としては，Ishio ら [11] によって Software Ingredients が開発されている．Software Ingredients では，クラスファイルの情報をもとにデータベースから候補となるライブラリを検出し，その後貪欲法で可能な限りライブラリ中のクラスファイルと対応付けを行い，複数のライブラリを検出している．ただし，この手法では再利用の際に Java の名前空間であるパッケージ名が変更されていた場合に対応できない．

Ishio らの手法をもとに，矢野ら [22] はパッケージ名の変更があったとしても再利用したライブラリを検出する手法を開発した．この手法では，クラスファイルの情報のうちパッケージに関する情報を削除して，かつ複数のライブラリを検出する際に貪欲法ではなく一部総当たりで組み合わせを検索し，最もクラスファイルをカバーできるようなライブラリの組み合わせを検出する．ただし，組み合わせを探索する際に再利用されたライブラリがある程度多くなった場合組み合わせ爆発が起きて検出が不可能である問題があった．

本研究では，Jar ファイルを入力として，Java パッケージに注目して再利用されたライブラリをあらかじめ作成したデータベース中から検出する手法を提案する．提案手法では，Java パッケージを既存手法で用いていたクラスファイルの情報の集合とみなし，集合全体を表すハッシュ値を用いることで再利用しているライブラリを検出する．検出に用いる単位をクラスファイルからパッケージにすることで，計算量が削減される．また，パッケージのハッシュ値に b-bit MinHash 法 [15] を用いることで，ハッシュ値の比較によって類似するパッケージを検出できるようにする．これにより，クラスの一部が変更または削除されていたとしても検出が可能となる．

以降，2 章ではソフトウェアの再利用分析に関する研究について述べ，3 章では提案手法とその要素技術について説明する．また，4 章では提案手法の検出性能を評価するための実験について詳述し，5 章では本研究を総括する．

2　ソフトウェア開発における外部ソフトウェアの再利用

現在，パーソナルコンピュータの普及と性能上昇に伴いソフトウェア開発への参入障壁は低くなっており，開発されるソフトウェアの数も増加している．また，企業の管理システムや事業の IT 化も推進されており，個人法人問わずソフトウェア開発はますます一般的になりつつある．しかし，開発するソフトウェアのある一部の機能はすでにインターネット上に再利用可能な状態で存在することがある．その場合ソフトウェア開発者は定められたライセンスに基づいてソフトウェアを再利用することができる．

2.1　ソフトウェアの再利用

ソフトウェアを開発する際，開発者は機能の実装を独自に行うのではなく，既に存在するライブラリなどのソースコードを再利用することがあると知られている [6]．ソースコードを再利用することで，局所的な機能を実装する必要がなくなり，開発コストの削減が可能となる．また，ライブラリとして提供されているソフトウェアは，個人で開発されている場合もあるが，オープンソースソフトウェアとして不特定多数の開発者が共同でメンテナンスを行っている場合もあり，独自に実装するよりも検証が十分になされているため信頼性が高いという利点もある．Mohagheghin ら [16] による調査では，ライブラリ中のコンポーネントに含まれる不具合の数は，独自に実装したソフトウェアのコンポーネント中に含まれる不具合の数よりも少ないと報告されている．

以上のようにライブラリを再利用してある機能を実装することは利点が多いと言えるが，信頼性が高いとはいえ，後々に脆弱性やバグが発見される場合もある．その場合，開発したソフトウェア中に含まれる再利用しているライブラリに関して更新作業が必要となる．開発したソフトウェアのドキュメント中に，どのライブラリのどのバージョンをどこで利用して開発したか正しく記載され管理されている場合であれば，ライブラリの更新作業は比較的容易であると言えるが，往々にして時間経過により担当者の交代やソフトウェア中の構成の変更などが生じるなどして，再利用した際の情報は正しく記録されていないことがあると報告されている [19]．そこで，記録されていない情報を復元するため，これまでに様々な方法が開発されてきた．

2.1.1　Java におけるソフトウェアの再利用

C 言語などにおいては，再利用の際に組み込む自分のソフトウェアに合わせて，コピーしてきたソースコードに手を加えることがある．その場合，ファイルのコピーを見つけるだけでは再利用情報の再構成に必要な情報が不十分な場合がある．本論文でフォーカスを当てている Java においては，前述のソースファイルを再利用するか，コンパイル後のバイ

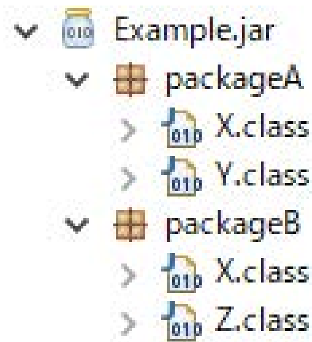

図 1: Java アーカイブの構造の例

ナリファイルを再利用するかのふた通りの再利用方法がある．Heinneman ら [8] は，彼らが行ったオープンソースソフトウェアの Java ソフトウェアに関しての再利用方法の調査について，ソースファイルを再利用する white-box reuse よりも，バイナリファイルを再利用する black-box reuse が一般的だとした．

2.1.2　パッケージリネーム

Java において，クラス名が衝突した際に異なるクラス名であると判別するためにパッケージと呼ばれる名前空間が存在する．Java アーカイブ中ではパッケージはそれぞれディレクトリとして構造が保持され，それぞれのクラスを利用する際にパッケージ名をクラス名の前にピリオドで連結して記述することで異なるクラスとして扱うことができる．図 1 に Java アーカイブの構造を示す．この例では，Example.jar 内に packageA，packageB が存在し，それぞれに X.class が存在するが，packageA.X，packageB.X と記述することでそれぞれのクラスを区別することができる．

ライブラリを再利用する際にパッケージ名を変更する場合がある．パッケージ名変更を支援するツールとして Maven Shade Plugin[1]や Jar Jar Links[2]というツールが存在する．パッケージリネームはいくつかの理由で行われる．

一つ目の理由として，同一ライブラリでバージョンの競合がある．ソフトウェアで利用しているライブラリにおいて，あるライブラリが依存しているライブラリをほかのライブラリでも利用していて，かつ依存するバージョンが異なる場合である．その例を図 2 に示す．図中の頂点はソフトウェア，矢印は依存関係を示している．図は開発中のソフトウェアでライブラリ A のバージョン 1 とライブラリ B を利用していることを表し，またライブラリ B は

[1]https://maven.apache.org/plugins/maven-shade-plugin/
[2]https://code.google.com/archive/p/jarjar/

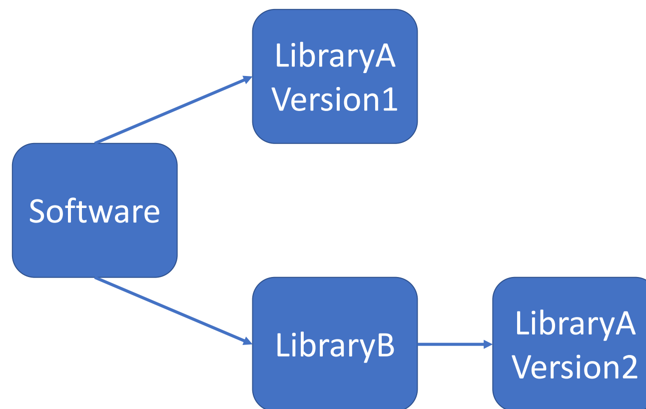

図 2: バージョン競合の例

バージョン 2 のライブラリ A を利用している．このとき，ライブラリ A の二つのバージョンは同じパッケージ名とクラス名がともに重複しているファイルを持つため，同時に利用することはできない．そこで，競合を回避するためにパッケージ名を変更することでバージョンの異なるライブラリを同時に複数利用することができる．

二つ目の理由として，依存関係の単純化がある．ライブラリの開発に際して異なるライブラリを再利用する場合に，ライブラリ利用者に別途依存するライブラリを導入させるのではなく，開発したライブラリ中にパッケージリネームを行い取り込むことで，ライブラリの利用者が依存関係を意識することなくライブラリを利用できるようにする場合である．図 2 の場合であれば，ライブラリ B がライブラリ A のバージョン 2 をパッケージリネームして取り込んでおくことで，利用者がライブラリ A の異なるバージョンを利用する場合もバージョンが競合することを考慮する必要が無い．

ここで挙げた二つの場合は，パッケージリネームによって得られるメリットにみに着目している．しかし，依存するライブラリを直接取り込んでしまうことは，正しく依存関係を把握することが難しくなり，ライブラリの管理に悪影響を与える可能性を孕んでいる．

2.2 再利用分析

ソフトウェア開発におけるライブラリなどの再利用について，失われてしまった出典の情報などを復元するための手法に再利用分析がある．再利用分析とは，ソフトウェア内に存在するソースファイルのうち，他のソフトウェアやライブラリなどから流用されたものの再利

用元を分析することである．再利用する際にソースファイルを編集していなければ，単純なファイル比較やファイル中の文字列の比較で再利用元を容易に特定できる．しかし，実際の開発環境では細かな設定の変更や関数の入出力を変更したい場合などコピーしたソースファイルに手を加えることがあり，この場合は単純に比較するだけでは再利用元の特定が困難である．プログラミング言語によって，再利用にソースコードを用いるかコンパイル後のバイナリファイルを用いるかの違いがあるため，再利用分析に関しても言語ごとに異なるアプローチを取る必要がある．そこで，編集されたソースファイルの出自を分析するためにこれまで様々な手法が開発されてきた．

2.2.1　ソースコードを利用した再利用分析

ソースコードを再利用することができる言語では，コードの記述内容を比較することで再利用元の分析が可能である．例えば，ソースコード間の類似性を表す要素としてコードクローン [2] がある．コードクローンはソースコード中に含まれる一致または類似するコード片のことで，主にソースコードのコピーアンドペーストでの再利用によって生じる．コードクローン検出について様々な研究が行われており，特にソフトウェアの再利用に注目して行われた研究に Inoue ら [10] の Ichi Tracker がある．Ichi Tracker は対象のある一つのソースファイルについてインターネット上のオープンソースソフトウェア中から類似するソースファイルを取得し，対象のコード片との類似度を用いて結果を分類し，取得したファイルを時系列で並べることでソースファイルの再利用の変遷を可視化する．ほかに，Sasaki ら [17] 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，FCFinder というツールで UNIX OS の一種である FreeBSD 中のオープンソースソフトウェア間で C 言語により記述されたソースファイルの再利用状況を調査した．

また，コードクローンを直接検出するのではなく，異なる二つのソースファイルに着目してその類似度をもとに再利用分析する手法もある．Kawamitsu ら [13] は，分析対象のソフトウェアの開発リポジトリとあるライブラリの開発リポジトリを入力として，分析対象のそれぞれのバージョンで再利用されているライブラリのバージョンを特定する技術を開発した．それぞれのリポジトリ中に存在するソースファイルで組を作り，その組での字句単位での和集合に対する最長一致部分列の大きさの割合を類似度が最大となるような組を検出し，その情報をもとに再利用元のバージョンを特定する．ただし，この手法では利用されているライブラリのバージョンを特定するにとどまるので，どのライブラリが再利用されているかは既知であり，どのバージョンが再利用されているか未知の場合でしか利用できない．

この問題を克服するため，Ishio ら [12] は膨大な数のオープンソースソフトウェアの，異なるバージョンでのソースファイルを収集しそれをデータベースとすることで，ソースファイルの集合を入力として再利用元と考えられるソフトウェアとそのバージョンをデータベー

ス中から検出する手法を開発した．

2.2.2 Java における再利用分析

Java でのソフトウェア開発では，ライブラリを再利用する際にソースコードではなくコンパイル後のバイナリファイルの一種であるクラスファイルを用いることが一般的である [8]．さらに Bavota ら [1] の調査では，Java でのソフトウェア開発ではその進行に応じて利用するライブラリの数が増加していくことが報告されており，再利用分析の重要性は高い．バイナリファイルの再利用が一般的な Java では，前節で述べたような手法をそのまま適用して再利用分析ができないため，ソースファイル以外の情報から再利用分析する手法や，クラスファイルから情報を抽出することで再利用分析を可能にする手法などが開発されている．

ソースファイル以外から情報を得る場合，Java においては主に Apache Maven Repository[3]に登録されている Java ライブラリを対象にして POM ファイルと呼ばれる登録ライブラリの設定ファイルを参照することが多い．Kula ら [14] は，ソフトウェアが再利用しているライブラリの依存関係がどのように変遷したのかを可視化する手法を提案した．注目するソフトウェアを中心に置き，径方向をソフトウェアのバージョンの変化として依存しているライブラリが円状に配置され，更新されているかやバージョンによっては使われていないというようなことも容易に確認できる．Yano ら [20] は，POM ファイルからライブラリの利用状況などの統計的な情報を取得し，その情報から複数のライブラリについてどのような組み合わせで利用されているか検索する VerXCombo というツールを開発した．ソフトウェア中のライブラリを更新する際，更新後のバージョンに後方互換性がない場合，そのライブラリに依存する他のライブラリが動作しなくなる可能性がある．VerXCombo を用いてライブラリの同時利用される組み合わせを確認することで，過去に利用された実績のあるライブラリの組み合わせを把握することができ，その情報に従ってライブラリの更新が可能かどうか考慮することで，後方互換性に端を発する問題を回避することが可能となる．

しかしながら，外部ファイルから得られる情報には限りがあり，可能であればソースファイルから情報を抽出したい．Java においては，コンパイルされたバイナリファイルであるクラスファイルにはある程度の文字列情報が含まれている．Teyton ら [18] は，Java アーカイブファイルから取得したパッケージ名を利用して，その Java ソフトウェア中に含まれるライブラリを特定する手法を開発した．しかし，同一ライブラリであればパッケージ名はたいていの場合同一であるため，バージョンの特定まではできない．また，前述したパッケージリネームが行われた場合に対応できない．

Davies ら [5] は，Java アーカイブ中からクラスファイルを取り出し，クラスファイルから

[3]https://mvnrepository.com/

得られる情報をもとに異なる Java アーカイブ間の類似度を計算することで，再利用したライブラリとそのバージョンを検出する Software Bertillonage というツールを提案している．このツールでは，Java アーカイブ形式でライブラリを再利用している場合には有効な手法であると言える．しかし，fat jar や jar with dependencies と呼ばれる Java アーカイブファイルを展開して開発中のソフトウェア固有のソースファイルとともに一つの Java アーカイブファイルに圧縮している場合にはあまり有効ではない．このような Java アーカイブファイルは Maven Assembly plug-in[4]や Java ソフトウェアの統合開発環境である eclipse[5]の実行可能ファイル作成機能などで簡便に作成することが可能で，複数のライブラリに由来するソースファイルを持つ Java アーカイブファイルで存在するソフトウェアは少なくない．

この問題点に対応するため，Ishio ら [11] は Software Ingredients というツールを開発した．このツールでは，Software Bertillonage と同様に Java アーカイブからクラスファイルを取り出し，コンパイラに依存しない箇所，すなわちクラス名やメソッド名などの文字列情報をもとに，注目するソフトウェアと一致するクラスファイルの割合から，貪欲法によって再利用しているライブラリを検出する．ただし，この手法においてもパッケージリネームには対応することができない．

[4]http://maven.apache.org/plugins/maven-assembly-plugin/
[5]https://eclipse.org/

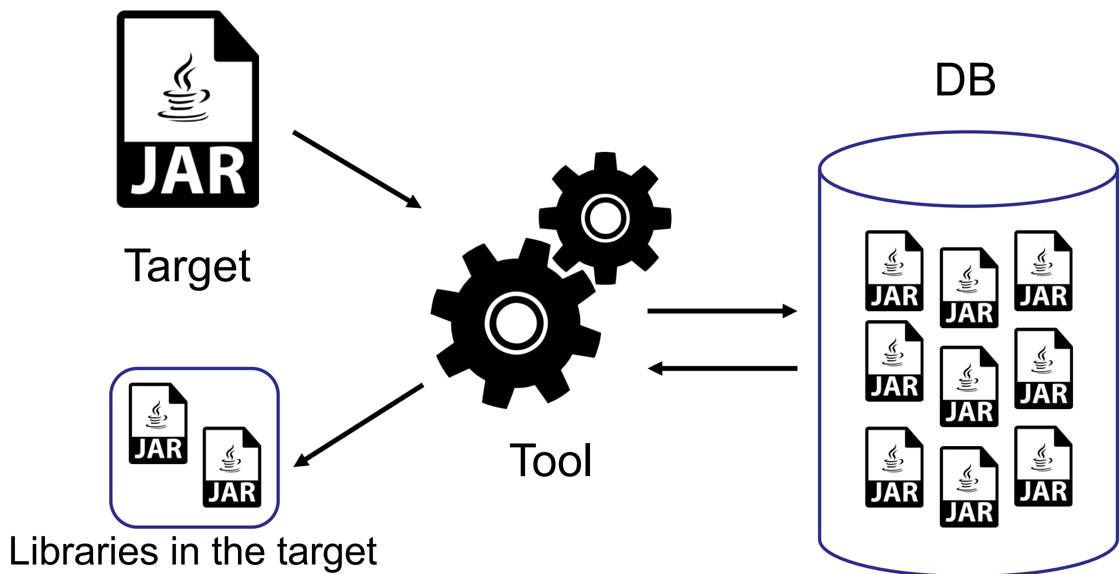

図 3: 提案手法の概要図

3 パッケージに注目したライブラリ検出手法

本研究では，Java のバイトコードである Java アーカイブに含まれる外部ライブラリをパッケージ単位での再利用情報に基づいて検出する手法を提案する．本手法は入力として Java アーカイブファイルを受け取り，事前に構築したデータベースに登録されているライブラリと比較することで，入力されたソフトウェア中に含まれるライブラリの一覧を得る．手法の概要図は図 3 のようになる．また，どのパッケージがどのライブラリから再利用されたかについての情報も得ることができる．

Java のバイトコードである jar ファイルには，コンパイルされた複数の Java バイトコードやそれが使用する画像などのリソースが含まれるが，本手法では内部に含まれるバイトコード，つまりクラスファイルのみを比較に利用する．また，直接クラスファイルを比較するのではなく，クラスファイルの属するパッケージごとにハッシュ値を求め比較することで，データベース中から検索する際の計算量を削減している．本章では，手法の具体的な手順について説明する．

3.1 パッケージ間の類似度

本手法ではパッケージをクラスファイルの集合と見做す．クラスファイルの集合を S_1, S_2 とすると，パッケージ間の類似度とは，比較する二つのパッケージに含まれるクラスファイ

ルの和集合と共通部分を用いて，以下の式 1 で表される．これは Jaccard 係数で表される類似度で，二つの集合 S_1, S_2 の和集合に対する共通部分の割合である．

$$sim(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{1}$$

しかしながら，パッケージに含まれるクラス同士を直接比較するのでは，クラスファイル単位で検出をする場合と変わらない．何らかの方法で類似するパッケージを高速に検出することができれば，一致するクラスファイルを検出するよりも軽量に利用しているライブラリを検出可能となる．そこで，集合間の類似度を統計的に計算する手法を用いる．

3.2 LSH による類似集合の高速検索

Jaccard 類似度を計算するには，それぞれの要素を総当たりで比較して和集合と共通部分の大きさを計算する必要がある．そのため，比較する集合の持つ要素数が大きくなればなるほど計算コストは高まっていく．その問題を解決するため，計算量を削減して類似度を統計的に推定する手法として Locality Sensitive Hashing (LSH) を用いる．LSH とは邦訳では局所性鋭敏ハッシュと呼ばれ，主に近似最近傍探索に利用される手法である [9]．微かな違いに鋭敏な何らかのハッシュ値を用意し，そのハッシュ値同士を比較することで類似度を高速に計算することができる．LSH の定義は，あるハッシュ族 H を用いて式 2 であらわされる [4]．

$$\mathrm{Pr}_{h \in H}[h(x) = h(y)] = sim(x, y) \tag{2}$$

ここで，$sim(x, y)$ は x, y 間の類似度で，値域は $[0, 1]$ である．Jaccard 類似度を推定する LSH の一つとして，MinHash 法 [3] が挙げられる．

3.2.1 MinHash 法

MinHash 法では，あるハッシュ族 H を集合の要素に対してそれぞれ適用し，同じハッシュ族で計算されたハッシュ値のうちの最小値を保存しておく．この操作を複数の異なるハッシュ族 H を k 個用意して行い，異なる集合間で同じハッシュ族を用いて得られた値が一致する確率が Jaccard 類似度に一致することが知られている．これは全体集合 U の部分集合である集合 S_1, S_2 について考えるとき，つまり $S_1, S_2 \subset U$ であるとき，U の要素をランダムに並べ替えた際に，S_1, S_2 の和集合の要素のうち先頭になった要素が S_1, S_2 の共通部分に含まれる確率と同じであり，式 1 と一致する．

MinHash 法で用いられるハッシュ値には，偶然衝突する確率をなるべく低くするため 64 ビットの整数を用いることが多い．そのため，MinHash を適用する際に一つの文書に対して必要な情報量は 64 ビット整数× k 個で，文書あたり $8k$ バイト必要である．保存に必要な

容量は元の文書のサイズにかかわらず一定であり，場合によっては保存に必要な容量が元より大幅に増えてしまう．また，ハッシュ族 H の個数 k は誤差を低減させるために数百個用意する必要があり，二つの集合のハッシュ値の比較にも大きな計算量が生じる．

3.2.2　b-bit MinHash 法

MinHash 法での問題を解決するため，Li ら [15] は b-bit MinHash 法を開発した．この手法では，MinHash 法でのハッシュ値のサイズを削減するために 64 ビットハッシュ値の下位 b ビットのみを用いる．論文中ではもっとも計算量が少なくなるのが $b=1$ のときであると述べられており，このときの情報量は $1 \times k$ ビットとなるので 1/64 の保存容量で済む．また，Li らは b の値によらず誤差の規模は 10^{-4} オーダーであることから，計算コストだけでなく誤差の規模を含めて考慮したとしても $b=1$ とすることが最も適していると述べている．ハッシュ値同士の比較に関する計算量についても，k ビットの整数を一度比較するのみなので，大幅に削減が可能である．以上を踏まえ，本論文中では $b=1, k=2048$ とする．$b=1$ の時，二つの b-bit MinHash の値 b_1, b_2 間の類似度は以下の式 3 で表される．

$$sim(b_1, b_2) = \left(P_o(b_1, b_2) - \frac{1}{2}\right) \times 2 \tag{3}$$

ただし，$P_o(b_1, b_2)$ は $b_1(i), b_2(i)$ が一致する確率である．

3.3　クラスファイルのシグネチャ

クラスファイルの集合であるパッケージ間の類似度を計算するためには，どのような形であれクラスファイルの情報を比較する必要がある．クラスファイルそのものを比較に用いる場合ファイルサイズに応じて計算量が大きくなる上，コンパイルを行った環境によってクラスファイルが変化してしまうことがある．そこで本手法では，クラスファイルからコンパイル環境に依存しないような情報のみを抽出し，その情報から SHA-1 ハッシュ値を計算しクラスファイルのシグネチャとして利用する．

シグネチャの計算に利用する情報は，Davies ら [5] の Software Bertillonage や Ishio ら [11] の Software Ingredients などと同様に，クラス名やメソッド名，フィールドなどである．ただし，パッケージ名の情報は取り除くことで，パッケージリネームが行われて再利用されていた場合にも検出が可能なようにする．以下に実際に利用した情報を列挙する．

- クラス名
- 親クラス名
- 外部クラス名（内部クラス名だった場合）

図 4: クラスファイルのシグネチャの計算

- 実装されているインターフェイス
- フィールド宣言（順不同）
  - フィールド名
  - フィールドの型
- メソッド宣言
  - メソッド名
  - アクセス修飾子
  - 引数と戻り値の型
  - メソッド呼び出し（ただしメソッドの名前と引数と戻り値の型のみを利用，順不同）
  - フィールドへのアクセス宣言（ただしフィールドの名前と型，所属クラスのみを利用，順不同）

本手法では，これらの情報が完全に一致するクラスファイル同士を同一のクラスファイルとみなす．

また，Java ではクラス内に内部クラスを記述できる上，一部のクラスやメソッドは利用時に匿名クラスの実装が必要となる．それらのクラスは Java アーカイブ中では別のクラスファイルとして保持され，既存手法では内部クラスのクラスファイルも外部クラスと区別することなく検出に利用していた．本手法では，シグネチャ計算時に内部クラスの情報を外部クラスに一定の順序で連結することで一つのクラスシグネチャとして計算する．

例として，ある jar ファイル中の一つのパッケージに含まれるクラスファイルについてシグネチャを計算した結果を図 4 に示す．同一パッケージ内では同一名のクラスファイルは存在しないので，それぞれのパッケージはクラスファイルのシグネチャの集合として扱うことができる．

3.4 パッケージに注目したハッシュ値

前節ではクラスファイルのシグネチャについて計算方法を述べた．それを受けて，本節ではパッケージに関する二つのハッシュ値の計算方法について述べる．一つは，パッケージの

完全一致性を示すためのパッケージでのシグネチャである．二つ目は，若干の構成するクラスの変化に対応するための，クラスの集合として見た場合のパッケージの b-bit MinHash の値である．

3.4.1　パッケージのシグネチャ

パッケージ同士の同一性を示すシグネチャの計算方法について述べる．パッケージはクラスファイルで構成されているので，パッケージのシグネチャを計算する際もクラスファイルに含まれる情報を用いて行う．ここでは，3.3 節で述べたクラスファイルのシグネチャを用いて，パッケージのシグネチャは次のように計算される．

まず，パッケージに含まれるクラスファイルのシグネチャを計算する．次に，それらを";"を末尾に足して文字列として連結を行う，最後に，クラスファイルのシグネチャと同様に文字列から SHA-1 ハッシュ値を計算する．この値を比較することで，完全に一致するようなパッケージを検出することができる．

3.4.2　パッケージに対する b-bit MinHash の適用

3.2 節で述べた，b-bit MinHash をパッケージに適用する手法について説明する．b-bit MinHash 法では，精度を維持するために，MinHash 法に比べて多くのハッシュ族を用意する必要がある．そこで，本研究ではハッシュ族の個数は $k = 2048$ として扱う．

前述している通り，パッケージはクラスファイルの集合と見做すことができる．そこで，クラスファイルのシグネチャに対して b-bit MinHash 法を適用することで，k 個の 1 ビットのハッシュ値を得る．これは k ビットの一つのハッシュ値として扱うことができ，これをパッケージの b-bit MinHash の値とする．

パッケージを比較する際，このハッシュ値のハミング距離を計算することで，あるパッケージに含まれるクラスと他のパッケージに含まれるクラスがどの程度一致しているのか判断することができる．結果として，パッケージの中の一部分のクラスのみを再利用していた場合に対応することができるようになる．

3.5　再利用元の検出

前節までで述べた情報を元に，入力された Java アーカイブファイルに含まれるライブラリをデータベース中から検出する．一般的に Java のライブラリは完全に再利用されることが多く，パッケージのシグネチャ，つまり完全一致による検出で多くのライブラリ名とバージョンの特定が可能である．そのため，候補とするライブラリは入力ソフトウェアと共通するパッケージの割合が多いものから選択していく．

Algorithm 1 再利用元の推定アルゴリズム

INPUT: t : target jar file, $R = \{r_i | i = 1...n\}$: Database(Set of Library)

OUTPUT: $Result$: Subset of Repository(and overlapped packages)

1: $P_t \leftarrow Packages(t)$ ▷ t に含まれている再利用元が未確定のパッケージ
2: $Result \leftarrow \{\}$ ▷ 検出結果を格納するライブラリの集合
3: $i \in [1, n],\ P'_{r_i} \leftarrow \{\}$ ▷ $Packages(r_i)$ における P_t との共通部分
4: **loop**
5: **for** $i = 1$ *to* n **do**
6: **if** $r'_i \in Result$ **then**
7: **continue**
8: **end if**
9: $P'_{r_i} \leftarrow Common(P_t\ , Packages(r_i))$
10: **end for**
11: $m \leftarrow \max_{i \in [i, |R|]} \left(\frac{|P'_{r_i}|}{|Packages(r_i)|} \right)$
12: **if** $m < th$ **then**
13: **break** ▷ 共通割合の最大値が閾値より小さいとき終了
14: **end if**
15: $R'_m \leftarrow \left\{ r'_n | \ \frac{|P'_{r_i}|}{|Packages(r_i)|} = m \right\}$
16: $J \subseteq R'_m\ \wedge\ Packages(J) \subseteq P_t$ かつ $|Packages(J)|$ が最大となる J を選択
17: $Result \leftarrow\ Result \cup J$
18: $P_t \leftarrow\ P_t \setminus Packages(J)$
19: **end loop**
20: **return** $Result$

検出アルゴリズムを Algorithm1 に示す．解析対象のソフトウェア t と事前に作成したライブラリの情報を登録したデータベース R を入力として，R から t が再利用しているライブラリ群を検出し出力とする．ただし，R では完全に同一のライブラリは取り除かれており，重複するライブリは存在しないとする．ここで，$Packages(x)$ はあるソフトウェア x に含まれるすべてのパッケージのハッシュ値を返す．また，$Common(x, y)$ は集合 x, y に関して，完全に一致する要素だけでなく類似すると判定された要素も含まれる共通部分の集合を返す.

検出は主に二段階で構成されており，第一段階ではデータベース中から入力ソフトウェアと類似するパッケージを持つライブラリを全て選択する．この処理は Algorithm1 の 5-10 行目に該当する．第二段階では，選択されたライブラリ群から重複なくパッケージをなるべく多く含むような組み合わせを探索する．この処理は 15-17 行目に相当する．選択されたパッケージは探索対象のパッケージの集合 P_t から取り除き，$|P_t| = 0$ となるまでこれらの処理を繰り返す．それぞれの段階について詳述する．

3.5.1 共通するパッケージを持つライブラリの選択

まず，データベース $R = \{r_i|i = 0...n\}$ 中から入力ソフトウェア t と共通するパッケージを持つライブラリを選択する処理について述べる．P_t について R 中の各ライブラリ r_i に対して，$P'_{r_i} = P_t \cap Packages(r_i)$ であるような $Packages(r_i)$ の一部分を抽出する．加えて，P'_{r_i} を抽出する際に，パッケージ間の類似度が閾値以上のものも含める．このとき，r_i に関して，ライブラリ中に含まれる t との共通部分の割合を式 4 によって計算する．

$$IntersectionRate(t, r_i) = \frac{|Common(P_t, Packages(r_i))|}{|Packages(r_i)|} \tag{4}$$

全ての r_i に対して $IntersectionRate(t, r_i)$ を計算する．その値が最大であるような r_i をすべて選択し，R の部分集合 $R'_m = \{r_j|j = 1...m\}$ とする．そして，R'_m について再利用ライブラリの組み合わせ探索を行う.

例として，図 5 のような入力を考える．図は，入力ソフトウェアとして Input.jar が与えられたとき，検出アルゴリズムの第一段階が終了した状態である．この例では，Input.jar と一致または類似するパッケージを含むライブラリがデータベースから 4 つ選択された．入力とデータベースから検出された各ライブラリに含まれるパッケージについて，互いに完全に一致するパッケージを赤字で，類似するパッケージを青字で表示している．4 つのライブラリについて共通部分の割合を計算すると，A-1.0.jar，A-1.1.jar，B-1.1.jar は 1.0，C-1.0.jar は 0.67 となる．B-1.1.jar と C-1.0.jar を比較すると，Input.jar と一致するパッケージは双方とも二つだが，B-1.1.jar は類似度が 0.99 であるパッケージを持つため，B-1.1.jar は全て

のパッケージが Input.jar との共通部分であるとする．よって，この中で次の段階への入力となるのは，C-1.0.jar 以外の全てである．

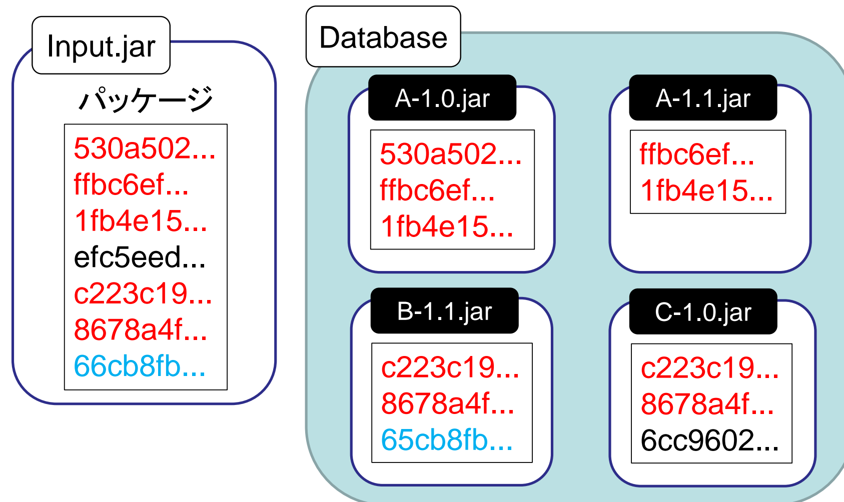

図 5: 第一段階の例

3.5.2 組み合わせ探索

次の段階として，前段階で選択された R'_m に関して，P_t の部分集合でありその要素数が最大となるような P'_{r_j} の組み合わせを探索する．本研究では，入力ソフトウェア t に含まれるあるパッケージは一つのライブラリからのみ再利用しているとする．そのため，最終的に検出されるライブラリ間には t と共通するパッケージに関して重なりがないように選ぶ必要がある．そこで，探索を行う R'_m の中から，他の再利用ライブラリの検出に影響を与えない r_j は省く．つまり，P'_{r_j} の全ての要素が R'_m 中の他の全ての P'_{r_j} に含まれないような r_j は事前に検出結果とする．

事前に検出した P'_{r_j} を取り除いた R'_m について P_t の部分集合として最大となるような r_j の組み合わせを深さ有線探索により全探索する．深さ優先探索とは，探索対象の要素をノードとして，あるノードから葉ノードまで探索を行い，その後未探索のノードがある場合は未探索のノードを子に持つノードまで戻り，また葉ノードまで探索を繰り返す手法である．本手法での探索では，あるノードは子ノードとして自身の持つパッケージを持たないような

ノードしか持てないため，全ての組み合わせを列挙する必要はない．

その後，列挙された r_j の組み合わせのうち，その和集合が持つパッケージの数が最大となるような組み合わせを選択し，その組み合わせに含まれる全ての r_j を検出結果に加える．検出されたライブラリに含まれるパッケージを P_t から取り除き，共通するパッケージを持つライブラリの選択をもう一度行う．この操作を $|P_t| = 0$ となるか，共通部分の割合の最大値 m が閾値 th 以下になるまで繰り返す．

図 5 の例での第二段階の処理が終わった状態を図 6 に示す．はじめに，候補として選択されたライブラリの中に固有のパッケージのみを持つようなライブラリを選ぶ．この例では B-1.1.jar が固有のパッケージのみを持つので，検出結果に加え，組み合わせ探索からは除外する．次に，残った A-1.0.jar と A-1.1.jar について組み合わせ探索を行うが，これらはお互いに共通するパッケージを持つため，片方を選ぶともう片方は選択できない．このとき，パッケージ数の多い A-1.0.jar が選択される．

その後，Input.jar の持つパッケージのうち一つが検出されていないため，第一段階に戻り処理を繰り返すが，全てのライブラリに関して共通部分の割合が 0 となるため，処理が終了する．よって，最終的な検出結果は図 7 のようになり，再利用しているライブラリとして赤枠で囲まれた A-1.0.jar と B-1.1.jar が検出される．

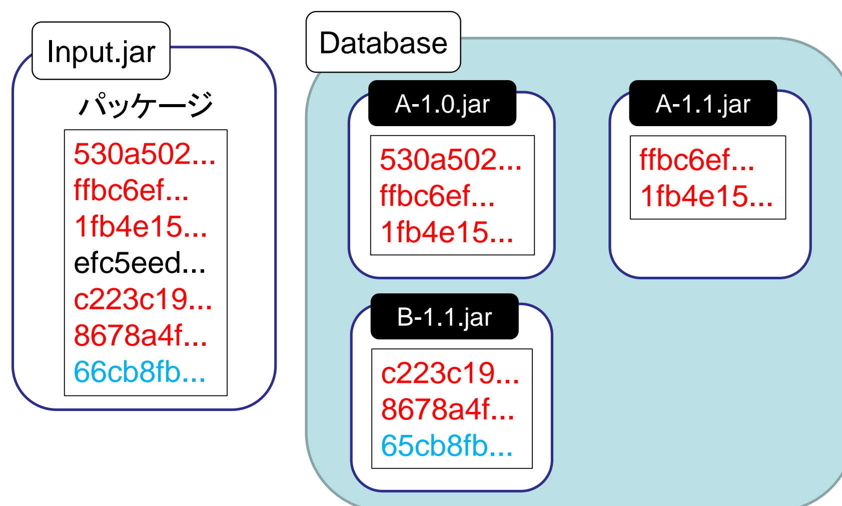

図 6: 第二段階の例

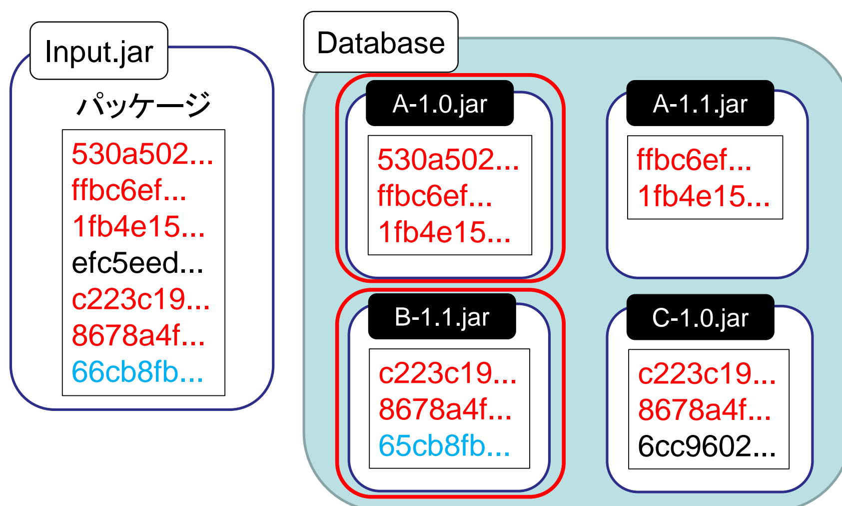

図 7: 検出されたライブラリ

4　実験

前章で述べた提案手法について，その特性を評価するために実験を行った．本手法ではパッケージ単位でのハッシュ値の導入について評価する．その結果について述べる．

4.1　実装

提案手法の実装は Java を用いて行った．Java バイトコード解析には，シンプルなバイトコード解析ツールである SOBA[21] を利用した．クラスファイルのシグネチャやパッケージのシグネチャには SHA-1 ハッシュを用いた．入力を Java で開発されたあるソフトウェアの Java アーカイブファイルとし，出力はデータベース中から検出された入力ソフトウェアに含まれるライブラリの一覧とした．

4.2　事前準備

評価実験を行うにあたり，インターネット上のリポジトリから Java ライブラリのデータを収集した．利用したデータは Maven Central Repository[6] の 2015 年 8 月のスナップショットで，破損したデータを除いて 172,038 個の Java アーカイブファイルを持つ．このスナップショットにはライブラリが 23,336 含まれ，ライブラリあたり平均　個のバージョンを持つ．このデータセットを用いて，データベースを構築し，人工的にライブラリを合成した Java アーカイブファイルを作成した．

4.2.1　データベースの構築

収集した 172,038 個のライブラリに関して，ライブラリごとにそれぞれの持つパッケージのハッシュ値を計算し，データベースとした．各ライブラリは以下の情報を持つ．

- ライブラリ名
- バージョン
- 含まれるパッケージのリスト

また，それぞれのパッケージは以下の情報を持つ．

- パッケージ名
- シグネチャ

[6] https://mvnrepository.com/

- b-bit MinHash の値
- 含まれるクラスファイルの数

データベース中のライブラリは平均して 13 パッケージ，150 クラスを持つ．また，パッケージは平均して 11 クラスを持つ．

4.2.2 実験用 Java アーカイブファイルの作成

一般に公開されているソフトウェアでは，ライブラリを再利用しているものは多い．しかし，正確に再利用したライブラリ名とバージョンを公開していることは少なく，評価で利用するには十分な数のデータセットを現実に存在するソフトウェアを用いて作成するのは難しい．そこで，本研究における評価では，矢野らの手法や Software Ingredients などと同様に，人工的に複数のライブラリを合成した Java アーカイブファイルを作成し，それらを入力とする．

実験用のデータセットは，データベースに利用した Maven Central Repository のスナップショットから，N 個のライブラリをランダムに選択し展開し一つの Java アーカイブファイルとして圧縮しなおすことで作成する．選択するライブラリの個数 N は 10 から 100 まで 10 刻み，つまり $N = 10, 20, 30, ..., 100$ とする．ランダムに選択したライブラリに含まれるパッケージ構造とクラスファイルはすべてコピーするが，ファイルの衝突が起きた場合は置換を行わずコピーしない．全てのファイルが衝突した場合は，改めて別のライブラリを選択しコピーする．

4.3 検出精度の評価

提案手法について，入力されたソフトウェア中に含まれるライブラリの検出精度を評価する．入力には前節で述べたデータベースと人工的に作成した Java アーカイブファイルを用いる．本節では評価方法とその結果を詳述する．

4.3.1 評価方法

4.2.2 節で述べた方法で，入力としてデータセットを用意する．これらのデータを提案手法に入力として与え，入力された Java アーカイブごとに検出結果の集合 $Result$ を得る．その後，データセット作成時に記録した合成に用いられたライブラリの集合 $Answer$ と $Result$ を比較する．$Result$ 中の，$Answer$ に含まれるライブラリ名が含まれている数を数え，式 5 と式 6 で適合率と再現率を評価する．

$$Precision = \frac{|Result \cap Answer|}{|Result|} \tag{5}$$

$$Recall = \frac{|Result \cap Answer|}{|Answer|} \tag{6}$$

4.3.2 結果

入力する Java アーカイブファイルを合成するライブラリの数 N ごとに 10 個ずつ，合計で 100 個の Java アーカイブファイルを作成した．いくつかの入力データでは，提案手法の第二段階での組み合わせの探索処理が終了しなかった．これは，組み合わせの取りうる数が探索にかける個数 m に関して最大 $2^m - 1$ 個となり，実用的な時間内に検出が不可能となるためである．本研究では検出時間が 30 分を超えた場合検出不能とした．

入力に対して得られた結果について，合成したライブラリ数に対する適合率と再現率の分布を図 8 と図 9 に示し，全体の適合率と再現率を表 1 に示す．また，検出時間を 10 に示す．

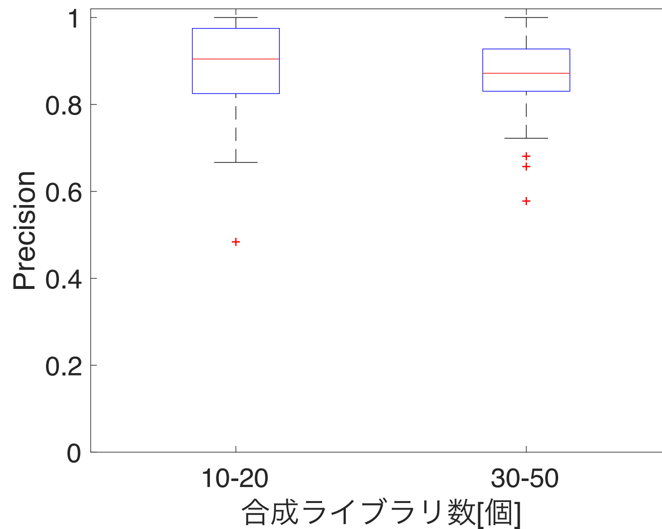

図 8: 提案手法で検出された結果の適合率の分布

Precision	Recall
0.87	0.89

表 1: 提案手法の適合率と再現率

また，本手法の組み合わせ探索をせず，共有部分の割合が大きいかつパッケージ数が多いものを優先的に検出する貪欲法を用いた場合についても評価した．このときの適合率と再現率を図 11 と図 12 に示し，全体の適合率と再現率を表 2 に示す．また，検出時間を 13 に示す．

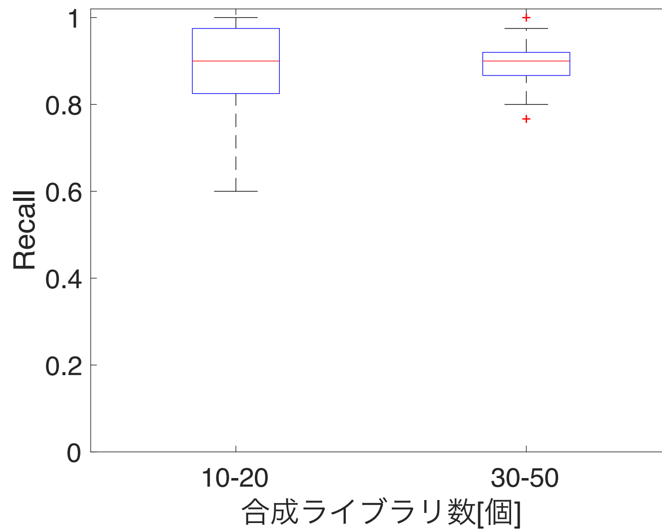

図 9: 提案手法で検出された結果の再現率の分布

Precision	Recall
0.87	0.89

表 2: 貪欲法の適合率と再現率

既存手法では，適合率が 0.87，再現率が 0.99 と報告されており，精度に関しては若干の低下がみられた．しかし，計算コストという点で見れば，クラス単位で検出する場合と比べて，パッケージ単位であれば比較回数が大幅に削減される．本研究で用いたデータセットでは，平均して一つのライブラリあたり 150 のクラスファイルが存在する一方，パッケージは 13 程度だった．検出時間に関しては，大部分が組み合わせ探索の時間であり，入力ファイルの解析には平均して 2 秒しかかかっていない．矢野らの手法と同様，精度に関しては貪欲法と組み合わせ探索を行う場合とでは差がなかった．

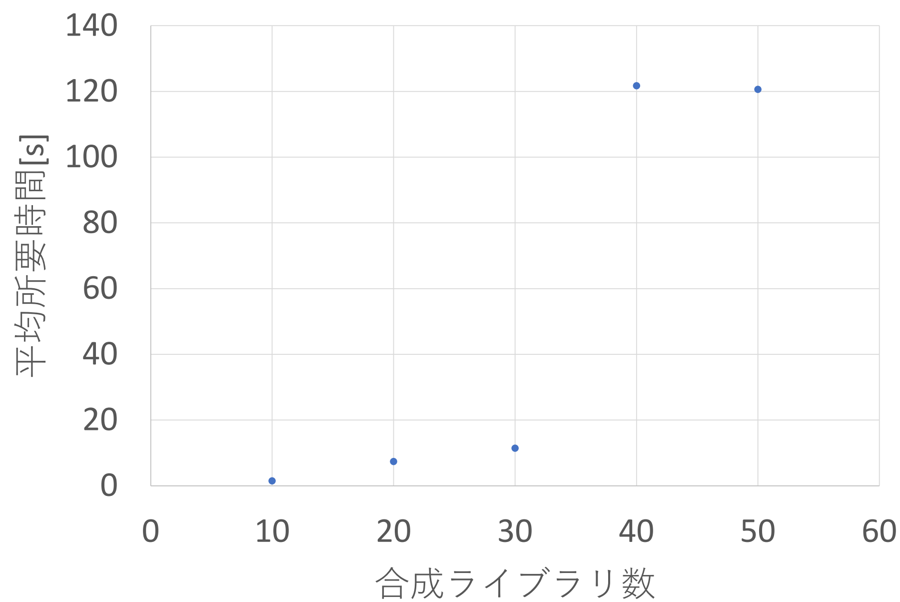

図 10: 提案手法の検出時間

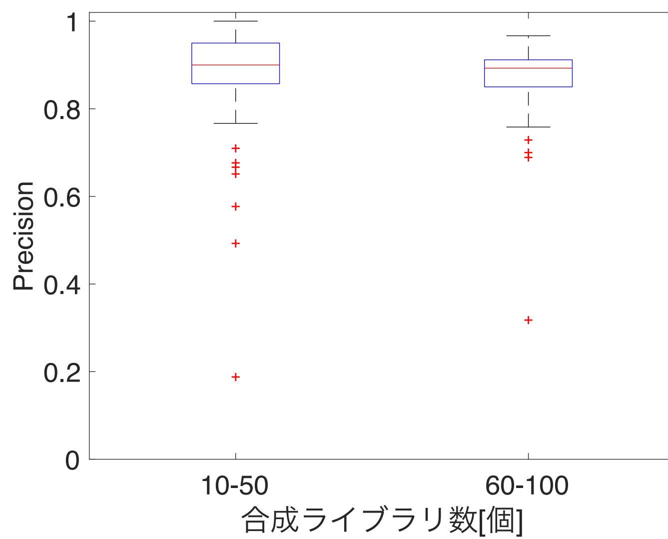

図 11: 貪欲法で検出された結果の適合率の分布

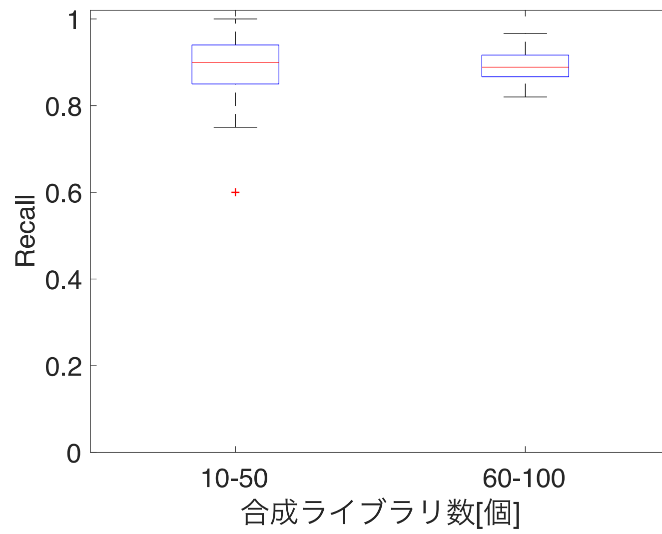

図 12: 貪欲法で検出された結果の再現率の分布

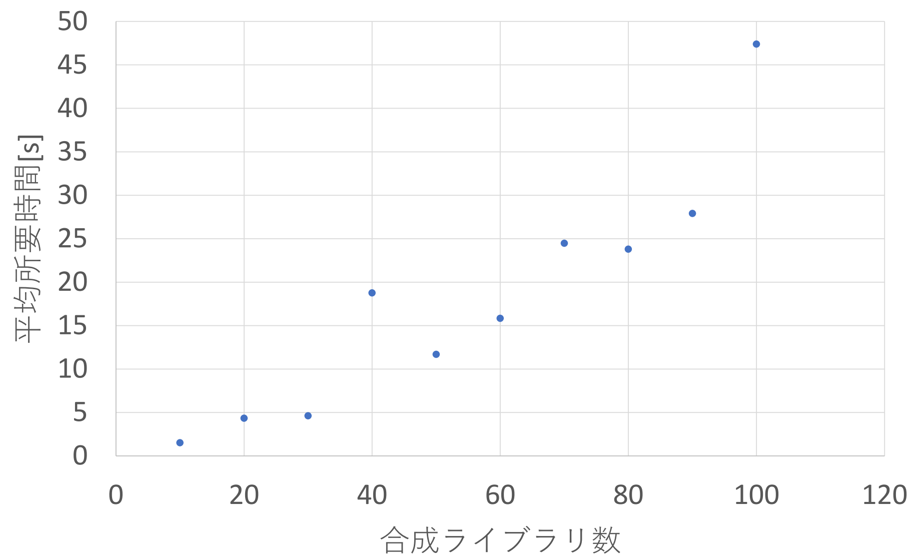

図 13: 貪欲法の検出時間

5　まとめ

本研究では，Java バイトコードから抽出した情報から計算したシグネチャを元に計算した b-bit MinHash の値を用いて，Java アーカイブ内部に含まれるライブラリの再利用を検出する手法を提案した．提案手法では，クラスファイルのシグネチャを計算する際にパッケージ名に関する情報を取り除くことによって，パッケージのリネームが行われていた場合でも再利用の検出を可能にした．さらに，Java パッケージに対して b-bit MinHash を用いることで，パッケージに含まれるクラスファイルが一部異なっていたとしても再利用の検出を可能にした．また，提案手法を実装したツールに複数個のライブラリを含む jar ファイルを入力することによって，再利用元を特定することが可能であるかどうか実験を行った．その結果，既存手法と比較して若干精度が低下したが，計算コストを削減することができた．

謝辞

本研究において，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします．本研究において，適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします．本研究において，研究の方針や論文の書き方など，様々な御指導を頂きました奈良先端科学技術大学院大学情報科学研究科石尾隆准教授に深く感謝いたします．本研究において，様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 特任助教に深く感謝いたします．本研究に限らず，様々なご指導およびご助言をいただきました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 嶋利一真氏に深く感謝いたします．最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします．

参考文献

[1] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pp. 280–289, Washington, DC, USA, 2013. IEEE Computer Society.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pp. 368–377, Washington, DC, USA, 1998. IEEE Computer Society.

[3] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pp. 21–29, Washington, DC, USA, 1997. IEEE Computer Society.

[4] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pp. 380–388, New York, NY, USA, 2002. ACM.

[5] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software bertillonage: Finding the provenance of an entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pp. 183–192, New York, NY, USA, 2011. ACM.

[6] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pp. 25–34, Washington, DC, USA, 2013. IEEE Computer Society.

[7] C. Ebert. Open source software in industry. *IEEE Software*, Vol. 25, pp. 52–53, 05 2008.

[8] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. On the extent and nature of software reuse in open source java projects. In *Proceedings of the 12th International Conference on Top Productivity*

Through Software Reuse, ICSR'11, pp. 207–222, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pp. 604–613, New York, NY, USA, 1998. ACM.

[10] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go? - integrated code history tracker for open source systems -. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 331–341, Piscataway, NJ, USA, 2012. IEEE Press.

[11] Takashi Ishio, Raula Gaikovina Kula, Tetsuya Kanda, Daniel M. German, and Katsuro Inoue. Software ingredients: Detection of third-party component reuse in java software release. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pp. 339–350, New York, NY, USA, 2016. ACM.

[12] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. Source file set search for clone-and-own reuse analysis. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pp. 257–268, Piscataway, NJ, USA, 2017. IEEE Press.

[13] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue. Identifying source code reuse across repositories using lcs-based source code similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 305–314, Sept 2014.

[14] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *Proceedings of the 2014 Second IEEE Working Conference on Software Visualization*, VISSOFT '14, pp. 127–136, Washington, DC, USA, 2014. IEEE Computer Society.

[15] Ping Li and Christian König. b-bit minwise hashing. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pp. 671–680, New York, NY, USA, 2010. ACM.

[16] Parastoo Mohagheghi, Reidar Conradi, Ole M Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the*

26th international conference on software engineering, pp. 282–292. IEEE Computer Society, 2004.

[17] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. Finding file clones in freebsd ports collection. pp. 102–105, 05 2010.

[18] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *J. Softw. Evol. Process*, Vol. 26, No. 11, pp. 1030–1052, November 2014.

[19] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Computer Software*, Vol. 30, No. 4, pp. 98–104, 2013.

[20] Yuki Yano, Raula Gaikovina Kula, Takashi Ishio, and Katsuro Inoue. Verxcombo: An interactive data visualization of popular library version combinations. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pp. 291–294, Piscataway, NJ, USA, 2015. IEEE Press.

[21] 秦野智臣, 石尾隆, 井上克郎. Soba:シンプルな java バイトコード解析ツールキット. コンピュータ ソフトウェア, Vol. 33, No. 4, pp. 4–15, 2016.

[22] 矢野裕貴, Raula Gaikovina Kula, 石尾隆, 井上克郎. Java バイトコード比較を用いたライブラリ再利用検出ツールの提案. 情報処理学会研究報告, 2017.

[23] 坂口雄亮, 石尾隆, 伊藤薫, 井上克郎. ソースファイル群の類似性を用いたソフトウェア再利用元の検索. 情報処理学会研究報告, 2017.