ソフトウェア開発における同時バージョン変更される併用ライブラリの推薦

桂川 大輝[1] 伊原 彰紀[2] ラウラ ガイコビナ クラ[1] 松本 健一[1]

概要：多くのソフトウェア開発は効率的な実装を実現するために，ソフトウェア機能の一部分を他人が作ったソフトウェア（特にライブラリ）を組み合わせて代用するソフトウェアの再利用が頻繁に行っている．しかし，再利用しているライブラリに問題が発生した場合，自身が実装していないため，検証や修正をすることが容易ではない．また，組み合わせて再利用しているライブラリ間の依存関係を把握することは難しく，ライブラリのバージョンを変更する場合は，ライブラリ間の依存関係を維持するために同時に利用しているライブラリのバージョン変更の必要性を検証する必要がある．本論文では，多種多様なライブラリを組み合わせて再利用しているソフトウェア開発プロジェクト 57,342 件を対象に，多数のプロジェクトにおいて同時にバージョン変更しているライブラリの組み合わせを明らかにする．さらに，ライブラリのバージョン変更において，同時にバージョン変更を検討すべきライブラリを推薦する手法を提案する．分析の結果，同時にバージョン変更されるライブラリを，提案手法により Recall Rate 47%で推薦できることを確認した．

1. はじめに

多くのソフトウェア開発プロジェクトでは，ソフトウェア機能の一部分を他人が作ったソフトウェア（特にソフトウェアライブラリ）を組み合わせて実装する開発（ソフトウェアの再利用）を頻繁行っている [11, 15]．ライブラリとは，汎用性の高い複数のプログラムを再利用可能な形でまとめたファイルであり，ライブラリを利用することでソフトウェアの一部の機能を容易に実装することができる．例えば，Web アプリケーション開発で利用されるライブラリには，プログラム実行時のログ出力，HTML 解析，SSH（セキュアシェル），暗号化の機能を実現する GWT [*1]，Spring [*2]，Hibernate [*3]がある．ライブラリの再利用は，開発期間の短期化に限らず，開発コストの低コスト化，高品質なライブラリの利用による製品の品質維持の実現において重要な役割を果たしている [16]．

今日，クラウド上でソフトウェアの管理，公開を支援するサービス GitHub の普及により，オープンソースソフトウェア開発プロジェクトだけでなく，ソフトウェア開発企業でさえもソフトウェアを構成するソースコードを広く公開するようになった．それに伴い，利用者は GitHub で公開されるソフトウェアを容易に再利用することが可能になった一方で，開発するソフトウェアに合わせてライブラリやそのバージョンを慎重に選択する必要となった．もし間違った選択をした場合，ライブラリ中に含まれる不具合やセキュリティ上の脆弱性，同時に利用するライブラリとの互換性の問題などに直面する可能性がある [1,3,4,10,12,18]．

従来研究では，ソフトウェア開発において高品質なライブラリを推薦するための手法が多数提案されている．多くの研究では，多数のプロジェクトにおいて同時期に利用されるライブラリの組み合わせを特定し，同時利用の頻度が高いライブラリを推薦する手法である [9,23]．従来手法は，推薦されたライブラリを利用することで開発効率を向上することができる可能性はあるが，推薦されたライブラリの利用は必須条件とは限らない．

本論文では，ライブラリのバージョン変更時に，同時に変更されるライブラリを推薦する手法を提案する．本提案手法は，従来手法に比べて同時に利用することを推めるライブラリの推薦手法である．本論文では，多種多様なライブラリを組み合わせて再利用しているソフトウェア開発プロジェクト 57,342 件を対象に，多数のプロジェクトにおいて同時にバージョン変更しているライブラリの組み合わせを明らかにし，従来手法 [23] を応用することで同時変更す

[1] 奈良先端科学技術大学院大学情報科学研究科
[2] 和歌山大学システム工学部

[*1] GWT: http://www.gwtproject.org
[*2] Spring: https://netbeans.org/kb/docs/web/quickstart-webapps-spring_ja.html
[*3] Hibernate: http://www.techscore.com/tech/Java/Others/Hibernate/index/

© 2018 Information Processing Society of Japan

べきライブラリを推薦する手法を提案する．

以降，2 章では関連研究，3 章では分析，実験を行うためデータセットの準備，複数ライブラリの同時バージョン変更が起こる頻度について述べる．4 章では同時バージョン変更される頻度の高い併用ライブラリの組み合わせについての分析，5 章では，推薦モデルの提案，評価について，6 章では分析結果の考察を述べる．7 章では本論文における制約を述べる．最後に 8 章では本論文でのまとめと今後の展望について述べる．

2. 再利用するライブラリの推薦に関する研究

安全なソフトウェアの再利用を実現するために，ライブラリ選択，また，ライブラリのバージョン選択を支援するための研究が多数取り組まれている．

Mileva らは，ライブラリの選択支援に向けて，“The Wisdom of Crowds”（みんなの意見は案外正しい） [22] の考えに基づき，大多数の開発者が再利用するライブラリは高品質であるという仮説のもとで，多くの開発者が再利用しているライブラリを推薦するソフトウェア AKTARI を開発している [17]．また，多数のソフトウェアにおいて併用されているライブラリの組み合わせに着目し，再利用できる可能性のあるライブラリを推薦する研究が取り組まれている．Thung ら [23] は，ソフトウェアで既に利用されているライブラリを入力とし，アソシエーションルールマイニング手法と協調フィルタリングを用いて，他のソフトウェアで頻繁に併用されるライブラリを推薦する手法を提案している．Ouni ら [20] は，多目的最適化を用いた検索ベースのアルゴリズム NSGA-II によってライブラリを推薦する手法を提案している．我々 [9] もまた，先行研究においてライブラリを用途別に分類できることに着目し，ソフトウェアで併用されている機能と該当するライブラリを推薦する手法を提案している．

さらに，ライブラリのバージョン選択を支援するために，ライブラリのバージョンの利用を分析した研究も取り組まれている．Kula らは，プロジェクトとの間に依存関係が存在するライブラリのバージョン更新では，最新版を導入する可能性は低いことを明らかにし，ライブラリの依存関係を可視化できるツールを開発した [13,25]．伊原らは，リリース頻度が低いライブラリを導入する際に、バージョンダウンする可能性が高いことを明らかにした [8]．

これら従来研究では，日常的に併用されるライブラリの組み合わせからライブラリ推薦，ライブラリのバージョン推薦を考察している．これら従来研究では，同時利用することでソフトウェア実装を効率化を目的としているため，依存関係をもつライブラリが推薦されるとは限らない．一方で，本論文が対象とするライブラリのバージョン変更においては，依存関係を持つライブラリも同時にバージョン変更する必要性が高く，たとえ依存関係をもたないライブラリでもバージョン変更している可能性がある．

表 1 同時バージョン変更されるライブラリ数

	最小値	平均値	中央値	最大値
ライブラリ数	1	2.45	1.0	87

本論文では，同時バージョン変更される頻度の高いライブラリの組み合わせを特定し，同時にバージョン変更するライブラリを推薦する手法を提案するために 2 つのリサーチクエスチョン（RQ）に回答する．

RQ1: 併用ライブラリはどのような組み合わせで同時バージョン変更されるのか？

RQ2: 同時バージョン変更される頻度の高い併用ライブラリをどれだけの精度で推薦できるのか？

3. 分析対象データセット

本論文では，Libraries.io [*4]においてオープンデータとして公開されているデータ [*5]を用いる．Libraries.io には，ソフトウェアとライブラリの依存関係に関するデータが保存されており，本論文では，分析対象ソフトウェア名，分析対象ソフトウェアのリリース時において再利用していたライブラリとそのバージョン，分析対象ソフトウェアがリリースされた日時を用いる．今回はケーススタディとして Java ライブラリとして Maven [*6]に管理されるライブラリを対象とし，ソフトウェア 57,342 件，対象ソフトウェアが利用しているライブラリ 45,044 種類を含む．

本論文が対象とする 57,343 件のソフトウェアは，合計 1,146,860 回の新しいバージョンがリリースしている．これらソフトウェアのバージョン変更において，ライブラリのバージョン変更が同時に行われたのは 228,104 回であった．表 1 は，分析対象ソフトウェアが再利用するライブラリのバージョン変更において，変更されたライブラリ数を示す．最小で 1 つ，最大で 87 つのライブラリが同時にバージョン変更されており，2 つ以上のライブラリが同時にバージョン変更されたのは 107,209 回（47%）であった．

また，表 2 は，再利用するライブラリのバージョン変更を実施したソフトウェア数を示す．本論文が分析対象とするソフトウェア 57,343 件のうち 29%は，ソフトウェアの新しいバージョンのリリースに伴い，複数のライブラリのバージョンを変更してりう．また，ライブラリのバージョンを変更したソフトウェアの中で，複数のライブラリのバージョンの変更が行われたのは 63% ($= \frac{16558}{26350}$) であった．

[*4] Libraries.io: `https://libraries.io/`
[*5] File name and digest : Libraries.io-open-data-1.2.0.tar.gz, Release Date : March 13
[*6] Maven: `http://search.maven.org/`

© 2018 Information Processing Society of Japan

表 2　分析対象ソフトウェアが再利用するライブラリのバージョン変更回数

	変更されたライブラリ数	ソフトウェアの割合	ソフトウェア数
バージョン変更されたライブラリなし	—	0.54	30,993
バージョン変更されたライブラリあり	1 つ以上	0.46	26,350
	2 つ以上	0.29	16,558

4. RQ1: 併用ライブラリはどのような組み合わせで同時バージョン変更されるのか？

4.1 分析手法

本章では，同時にバージョン変更される頻度が高いライブラリの組み合わせを明らかにすることを目的として，ソフトウェア 57,342 件において再利用される 2 種類以上のライブラリが同時にバージョン変更された 107,209 回のソフトウェアのバージョン更新を分析対象とする．

RQ1 では，同時にバージョン変更をされる頻度の高いライブラリの組み合わせを明らかにするために，アソシエーションルールマイニング手法 [2, 7, 26] を用いる．アソシエーションルール（ルール）は，ある事象の発生に対して，別の事象が発生する，同時性や関係性が強い事象の組み合わせ，強い事象間の関係を示す．また，アソシエーションルールマイニング手法とは，多数の事象の組み合わせから，ルールを抽出する技術を指す．ルールは条件部（A）と結論部（B）で表す．

$$\{A\} \Rightarrow \{B\}$$

ルールは支持度，信頼度，リフト値の 3 つの指標値で評価する．支持度（support）は全データの中でルールの条件部と結論部が出現するデータの割合を示す．

$$support(\{A\} \Rightarrow \{B\}) = \frac{\sigma(A \cap B)}{all\ software\ version\ updates} \quad (1)$$

信頼度（confidence）はルールの条件部の項目が出現するデータの割合の中で条件部（A）と結論部（B）が同時に出現するデータの割合を示す．

$$confidence(\{A\} \Rightarrow \{B\}) = \frac{support(\{A\} \Rightarrow \{B\})}{support(A)} \quad (2)$$

リフト値（lift）は条件部と結論部を共に満たすデータの割合と全事象の中で結論部を満たす割合を比較した倍率を示す．

$$lift(\{A\} \Rightarrow \{B\}) = \frac{confidence(\{A\} \Rightarrow \{B\})}{support(B)} \quad (3)$$

アソシエーションルールマイニング手法および評価値を用いることで，異なるソフトウェアに貢献する開発者が共通して同時にバージョン変更するライブラリの組み合わせを明らかにすることができる．アソシエーションルールマイニング手法では，多数のルールが出力されるため，apriori アルゴリズムを用いて指標値に一定の条件を満たすルールを選定する．本章では，最小支持度，最小信頼度をそれぞれ 0.01，0.1 と設定する．本論文では，Python 言語における apriori アルゴリズムが実装されているライブラリ `Orange` [5] を用いてアソシエーションルールマイニング手法を実装する．

4.2 結果

表 3 は，分析対象のソフトウェアのバージョン変更に伴い，同時にバージョン変更をされる頻度の高いライブラリの組み合わせをアソシエーションルールマイニング手法によって出力した結果を示す．表 3 は，ルールを信頼度に基づき降順にソートしているため，条件部のライブラリのバージョン変更と同時に，結論部のライブラリをバージョン変更する頻度が高い組み合わせを示す．

また，比較対象として，表 4 に従来研究 [23] のライブラリ推薦手法において出力される併用ライブラリの組み合わせも共に出力する．最小支持度は 0.018，最小信頼度は 0.1 とする．従来手法は，ライブラリのバージョンを更新するか否かは考慮していないため，本論文では，ソフトウェアが再利用するライブラリの中で，バージョン更新を必要とするライブラリの組み合わせのリストとは異なる．

知見 1 — 同じ組み合わせのライブラリが常に同時にバージョン変更されるとは限らない． 表 3, ルール 18, ルール 42 より，`jcl-over-slf4j` をバージョン変更する際に 87%の確率で `slf4j-api` が同時バージョン変更されるのに対して，同じライブラリの組み合わせではあるが，`slf4j-api` をバージョン変更する際は 24%の確率で `jcl-over-slf4j` が同時バージョン変更されるということが明らかになった．つまり，`jcl-over-slf4j` と `slf4j-api` のライブラリは，他の組み合わせに比べて同時にバージョン変更される可能性が高いが，常に同時にバージョン変更されるとは限らない．それゆえ，開発者がこのような関係すべてを共通の知見として把握することは大変困難なことであることが予測される．

知見 2 — 併用ライブラリとは異なる組み合わせが上位に存在する． 表 3，表 4 から，同時バージョン変更されるライブラリのルールは，併用 されるライブラリのルールと異なる組み合わせが上位に存在することが明らかになった．例えば，表 3 でもっとも信頼度が高いルールは表 4 に存在しない．常用的に併用されるライブラリのルールには，多

© 2018 Information Processing Society of Japan

表 3 同時バージョン変更される併用ライブラリ (信頼度によってソート)

ID	条件部		結論部	支持度	信頼度	リフト値
1	{jackson-annotations}	⇒	{jackson-databind}	0.01	0.97	48.33
2	{ spring-beans, spring-core}	⇒	{spring-context}	0.01	0.95	53.13
3	{spring-beans, spring-context}	⇒	{spring-core}	0.01	0.94	72.00
4	{spring-context, spring-core}	⇒	{spring-beans}	0.01	0.94	74.22
5	{jackson-core}	⇒	{jackson-databind}	0.01	0.94	46.50
6	{spring-beans}	⇒	{spring-context}	0.01	0.93	51.65
7	{spring-beans}	⇒	{spring-core}	0.01	0.92	69.99
8	{spring-core}	⇒	{spring-context}	0.01	0.90	50.11
9	{specs2-scalacheck_2.10}	⇒	{specs2-core_2.10}	0.01	0.89	50.00
10	{specs2-matcher-extra_2.10}	⇒	{specs2-core_2.10}	0.01	0.89	49.92
11	{specs2-matcher-extra_2.10,specs2-scalacheck_2.10}	⇒	{specs2-core_2.10}	0.01	0.89	49.74
12	{specs2-matcher-extra_2.11}	⇒	{specs2-core_2.11}	0.02	0.89	40.15
13	{spring-core}	⇒	{spring-beans}	0.01	0.88	69.99
14	{specs2-matcher-extra_2.11}	⇒	{specs2-scalacheck_2.11}	0.02	0.88	50.22
15	{specs2-scalacheck_2.11}	⇒	{specs2-core_2.11}	0.02	0.88	39.79
16	{spring-beans}	⇒	{spring-context, spring-core}	0.01	0.88	74.22
17	{specs2-matcher-extra_2.11, specs2-scalacheck_2.11}	⇒	{specs2-core_2.11}	0.01	0.87	39.51
18	{jcl-over-slf4j}	⇒	{slf4j-api}	0.01	0.87	20.99
19	{specs2-matcher-extra_2.11, specs2-core_2.11 }	⇒	{specs2-scalacheck_2.11}	0.01	0.86	49.42
20	{specs2-scalacheck_2.11}	⇒	{specs2-matcher-extra_2.11}	0.01	0.86	50.22
...		...		...	...	...
42	{slf4j-api}	⇒	{jcl-over-slf4j}	0.01	0.24	20.99

くのソフトウェアで利用されているライブラリが上位に存在し，それらは必ずしも同時にバージョン変更すべきライブラリではないため，表 3 で示すような同時バージョン変更される併用ライブラリのルールおよびその信頼度を用いて同時バージョン変更される併用ライブラリを推薦するのが望ましい．

5. 同時バージョン変更される頻度の高い併用ライブラリをどれだけの精度で推薦できるのか？

5.1 概要

RQ2 では，RQ1 で特定したルールに基づいて，ソフトウェアが再利用するライブラリのバージョン変更に伴い，同時にバージョン変更を必要とするライブラリを推薦するモデルを提案する．アソシエーションルールマイニング手法を用いて同時バージョン変更される頻度の高いライブラリを信頼度に基づき推薦する．

図 1 に，提案手法の概略図を示す．本論文が提案する手法では，モデル構築前に同時にバージョン変更される頻度の高いライブラリの組み合わせをルールとして保持する．ソフトウェアが新しいバージョンをリリースにおいてバージョン変更されたライブラリが存在した場合に，バージョン変更されたライブラリを入力として蓄積されたルール群と比較し，3 つの条件を満たすルールを選出する．

- ルールの条件部にあたるライブラリがバージョン変更するライブラリに含まれる．
- ルールの結論部にあたるライブラリがソフトウェアで併用されているライブラリに含まれる．
- ルールの結論部にあたるライブラリがバージョン変更するライブラリに含まれない．

次に，条件を満たす全てのルールの中で最も信頼度の高いルールの結論部を同時バージョン変更される頻度の高い併用ライブラリとして出力する．ただし，条件を満たすルールが存在しない場合（例えば，利用例の少ないライブラリのみを利用している，蓄積されるルール，トレーニングデータセットのサイズが不十分である）は，ルールを選出できないため，ランダムにライブラリを推薦する．

5.2 評価方法

本論文では，提案手法を評価するために，テストデータにおけるソフトウェアが再利用するライブラリの中で，バージョン変更されたライブラリをランダムに 2 分割し，一方をバージョン変更するライブラリとして推薦モデルへの入力，他方を入力に対して同時にバージョン変更をする併用ライブラリとして正解集合を作成する．これにより，開発者がバージョン変更を試みるライブラリに対して他の同時バージョン変更される頻度の高い併用ライブラリを発見するために貢献する状況を模倣する．

本論文では，提案手法の推薦精度を評価するために，既存研究でも利用されている評価指標として Recall Rate を定義する [19–21, 23, 24]．推薦されたライブラリが正解集合に存在するバージョン更新の割合を Recall Rate として算出する．

$$\text{Recall Rate} = \frac{\text{正解が存在するバージョン更新}}{\text{全バージョン更新}}$$

© 2018 Information Processing Society of Japan

表 4 常用的に併用されるライブラリ (信頼度によってソート)

ID	条件部		結論部	支持度	信頼度	リフト値
1	{jul-to-slf4j, commons-lang3, commons-io, slf4j-api}	⇒	{commons-codec, log4j-core}	0.01	1.00	84.68
2	{jul-to-slf4j, commons-lang3, commons-io, log4j-slf4j-impl}	⇒	{commons-codec, guava, log4j-core, slf4j-api}	0.01	1.00	84.68
3	{jul-to-slf4j, commons-lang3, commons-io, log4j-slf4j-impl}	⇒	{commons-codec, guava, log4j-core}	0.01	1.00	84.68
4	{commons-codec, guava, log4j-core}	⇒	{jul-to-slf4j, commons-lang3, commons-io, log4j-slf4j-impl}	0.01	1.00	84.68
5	{commons-codec, guava, log4j-core, slf4j-api}	⇒	{jul-to-slf4j, commons-lang3, commons-io, log4j-slf4j-impl}	0.01	1.00	84.68
6	{commons-codec, guava, log4j-core}	⇒	{jul-to-slf4j, commons-lang3, commons-io, log4j-slf4j-impl, slf4j-api}	0.01	1.00	84.68
7	{commons-lang3, guava, commons-io, log4j-core, slf4j-api}	⇒	{jul-to-slf4j, commons-codec, log4j-slf4j-impl log4j-slf4j-impl}	0.01	1.00	84.61
8	{disruptor, jul-to-slf4j, guava, commons-io, log4j-core, slf4j-api}	⇒	{og4j-jcl , javassist, log4j-api, log4j-slf4j-impl}	0.01	1.00	84.35
9	{disruptor, jul-to-slf4j, guava, commons-io, log4j-slf4j-impl, log4j-core}	⇒	{log4j-jcl , javassist, log4j-api, slf4j-api}	0.01	1.00	84.35
10	{disruptor, jul-to-slf4j, guava, commons-io, log4j-slf4j-impl, log4j-core, slf4j-api}	⇒	{log4j-jcl, javassist, log4j-api}	0.01	1.00	84.35
11	{disruptor, jul-to-slf4j, commons-io, log4j-core}	⇒	{log4j-jcl, javassist, log4j-api, log4j-slf4j-impl, slf4j-api}	0.01	1.00	84.35
12	{disruptor, jul-to-slf4j, guava, log4j-api, commons-io, slf4j-api}	⇒	{log4j-jcl, javassist, log4j-slf4j-impl}	0.01	1.00	84.35
13	{disruptor, jul-to-slf4j, guava, log4j-api, commons-io, log4j-core}	⇒	{log4j-jcl, javassist, log4j-slf4j-impl, slf4j-api}	0.01	1.00	84.35
14	{disruptor, jul-to-slf4j, guava, log4j-api, commons-io, log4j-slf4j-impl, log4j-core, slf4j-api}	⇒	{log4j-jcl, javassist}	0.01	1.00	84.35
15	{disruptor, jul-to-slf4j, guava, log4j-api, commons-io, log4j-slf4j-impl, log4j-core}	⇒	{log4j-jcl, javassist, slf4j-api}	0.01	1.00	84.35
16	{disruptor, guava, log4j-api, commons-io, log4j-core, slf4j-api}	⇒	{log4j-jcl, javassist, jul-to-slf4j, log4j-slf4j-impl}	0.01	1.00	84.35
17	{disruptor, guava, log4j-api, commons-io, log4j-core}	⇒	{log4j-jcl, javassist, jul-to-slf4j, log4j-slf4j-impl, slf4j-api}	0.01	1.00	84.35
18	{disruptor, guava, log4j-api, commons-io, log4j-slf4j-impl, log4j-core, slf4j-api}	⇒	{log4j-jcl, javassist, jul-to-slf4j}	0.01	1.00	84.35
19	{disruptor, guava, log4j-api, commons-io, log4j-slf4j-impl, log4j-core}	⇒	{log4j-jcl, javassist, jul-to-slf4j, slf4j-api}	0.01	1.00	84.35
20	{disruptor, guava, commons-io, log4j-core, slf4j-api}	⇒	{log4j-jcl, javassist, jul-to-slf4j, log4j-api, log4j-slf4j-impl}	0.01	1.00	84.35

本論文では，提案手法の評価として Recall Rate を測定するために，10 分割交差検定を行う [6,14]．はじめに，ソフトウェアをランダムに 10 つの同じサイズのセットに分割する．1 回の施行で 1 つのセットに存在するソフトウェアのバージョン変更をテストデータセット，他 9 つをトレーニングデータセットとして推薦を実施し，その Recall Rate を算出する．以上の施行を 10 回繰り返し，Recall Rate の平均，中央値，最小値，最大値を評価する．

本論文では，極力多様なソフトウェアのバージョン更新に対応できるよう，最小支持度（minsup）を 0.01，結果的に高い信頼度のものが推薦リストに加えられるため，最小信頼度（minconf）を 0.1 と設定する．

5.3 評価結果

本論文では，57,342 件のソフトウェアに対して 10 分割交差検定をし， 4.1 節の条件を満たすバージョン更新合計 107,209 件に対して実験を実施した．本論文では，提案手法の精度を評価するために，以下の条件を満たすライブラリをランダムに 1 つ推薦するモデル（ランダム推薦）と比較する．

- 併用ライブラリに属する．
- バージョン変更されない．

© 2018 Information Processing Society of Japan

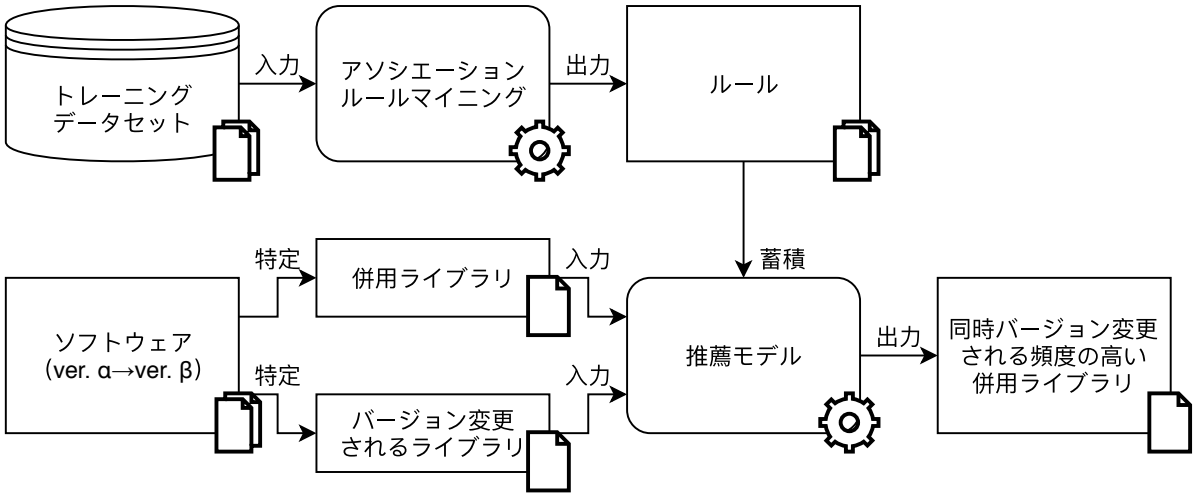

図 1 提案手法の概略図

表 5 Recall Rate による評価

	最小値	平均値	中央値	最大値
ランダム推薦	0.37	0.40	0.40	0.44
提案手法	0.40	0.44	0.44	0.47
差分	0.01	0.03	0.03	0.04

表 5 は，提案手法とランダム推薦における Recall Rate の平均，中央値，最小値，最大値，そしてそれらの差分（提案手法−ランダム推薦）を示す．

表 5 より，ランダム推薦と比較し，提案手法の Recall Rate が 44%（中央値）であることがわかる．また，差分の最小値が 1%であり，本実験において提案手法がランダム推薦よりも高いことが明らかとなったが，本提案手法の効果は小さいため，考察において議論する．

6. 考察

RQ2 より，提案手法の Recall Rate がランダムと比較して約 3%（中央値）高いことが明らかになった．しかし，推薦手法や推薦内容など場面は異なるものの，既存研究で提案されているアソシエーションルールマイニング手法を用いた推薦手法と比べ，本提案手法の効果は小さい．本章では，提案手法の推薦精度がランダム推薦と変わらない理由を考察する．

提案手法の推薦精度が低い原因として，多くのソフトウェアのバージョン更新に対して，条件を満たすルールが全く選出されなかったことが問題であった．5.1 節より，提案手法では，全てのルールの条件部が当てはまらず，ライブラリを併用ライブラリからランダムに推薦されていた．具体的には，本実験においてルールの条件が当てはまりライブラリが推薦できた割合は 6%（5,708 件），条件に当てはまるルールが全く存在せず，ランダムに推薦した割合は 94%（101,501 件）であり，多くのソフトウェアではルールに基づいてライブラリを推薦できていないことを確認した．

表 6 条件を満たすルール存在する場合の Recall Rate による評価

	最小値	平均値	中央値	最大値
ランダム推薦	0.95	0.99	0.99	1.00
提案手法	0.33	0.40	0.41	0.48
差分	0.52	0.59	0.58	0.66

表 7 推薦におけるルール数

	最小値	平均値	中央値	最大値
ルール数	32	41.8	38	78

条件を満たすルールが存在し，ライブラリが推薦できた 6%（5,708 件）ソフトウェアから算出した Recall Rate を表 6 に示す．提案手法の Recall Rate は 99%（中央値）であり，一方で，ランダム推薦の Recall Rate が 41%（中央値）であるため，条件を満たすルールを抽出できたソフトウェアについては高い精度で同時にバージョン変更されたライブラリを特定することができている．

本論文が対象とする多くのソフトウェアのバージョン更新に対して，条件を満たすルールが存在しなかった原因として，ルールの数，種類が十分でなかったことが考えられる．表 7 に今回の実験（RQ3）においてトレーニングデータセットから出力されたルールの数を示す．表 7 より，38（中央値）というルール数は，今回対象となったライブラリが 45,044 種であるのに対して少ないと考えられる．ライブラリ自体が多種多様に存在していること，ソフトウェアで利用されているライブラリもそれぞれ異なり多種多様であること，バージョン更新の際，バージョン変更されるライブラリはその中でも一部であることから，正しい推薦に貢献するルールの支持度がより小さくなり，設定した最小支持度を上回らず，除外されていると考えられる．

この問題を解決するために，対象ソフトウェアのバージョン更新に十分なルールが蓄積されるように必要がある．例えば，最小支持度下げる，設定しないということで，

© 2018 Information Processing Society of Japan

十分なルールを蓄積できるかと考えられるが，ライブラリの種類が多いことによって計算量が大きくなる，利用される頻度が極めて小さいルールがノイズとして生じるという問題もある．このように，提案手法を実現するために十分なルールの蓄積が課題である．

7. 制約

内的妥当性： 本論文では，アソシエーションルールマイニング手法によって同時バージョン変更される併用ライブラリの組み合わせを特定した．しかし，これらライブラリ同士の実際の依存関係を明らかにしていないため，これらを今後考慮する必要がある．

外的妥当性： 本論文では，分析及び実験対象を Maven プロジェクトのみとした．しかし，それ以外プロジェクトに対しても，本実験と同様の傾向が得られるかを調査する必要があり，今後の課題とする．

8. おわりに

本論文では，効率的なソフトウェア開発を実現するために，同時バージョン変更される頻度の高い併用ライブラリを分析した．さらに，これらを形式化することで，プロジェクトがライブラリのバージョンを変更する場合に，同時バージョン変更される頻度の高い併用ライブラリを推薦するモデルを提案し，その精度を検証した．分析結果として，ライブラリのバージョンを変更する際，47%の確率で複数ライブラリを同時にバージョン変更するということ，複数ライブラリの同時バージョン変更は 67%のプロジェクトで行われていたことを明らかにした．そして，提案手法が正しい推薦を実現する可能性が示唆された．特に，6%のバージョン更新に対して，99%という Recall Rate で正しい推薦ができることが明らかになった．今後は，提案手法が多種多様なソフトウェアのバージョン更新に対応できるよう十分なルールの蓄積を目標とした課題に取り組む．さらに，推薦されたライブラリの導入後に発生する問題の有無について考察し，開発者間で安全なライブラリの再利用に向けた知識共有方法について検討する．

謝辞

本論文の一部は，文部科学省科学研究補助費（基盤 A: 17H00731，若手 B: 課題番号 16K16037）による助成を受けた．

参考文献

[1] Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S. and Shihab, E.: Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm, *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, pp. 385–395 (2017).

[2] Agrawal, R. and Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pp. 487–499 (1994).

[3] Bogart, C., Kästner, C., Herbsleb, J. and Thung, F.: How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'16)*, pp. 109–120 (2016).

[4] Decan, A., Mens, T. and Claes, M.: An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems, *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pp. 2–12 (2017).

[5] Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M. and Zupan, B.: Orange: Data Mining Toolbox in Python, *Journal of Machine Learning Research*, Vol. 14, pp. 2349–2353 (2013).

[6] Efron, B.: Estimating the error rate of a precision rule: improvements on cross-validation, *Journal of the American Statical Association*, Vol. 78, No. 382, pp. 316–331 (1983).

[7] Han, J., Pei, J. and Yin, Y.: Mining Frequent Patterns Without Candidate Generation, *Proceedings of the International Conference on Management of Data (SIGMOD'00)*, pp. 1–12 (2000).

[8] Ihara, A., Fujibayashi, D., Suwa, H., Kula, R. G. and Matsumoto, K.: Understanding When to Adopt a Library: A Case Study on ASF Projects, *Proceedings of the International Conference on Open Source Systems (OSS'17)*, pp. 128–138 (2017).

[9] Katsuragawa, D., Ihara, A., Kula, R. G. and Matsumoto, K.: Maintaining Third-Party Libraries through Domain-Specific Category Recommendations, *SoHeal18:IEEE/ACM 1st International Workshop on Software Health* (2018).

[10] Kikas, R., Gousios, G., Dumas, M. and Pfahl, D.: Structure and Evolution of Package Dependency Networks, *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*, pp. 102–112 (2017).

[11] Krueger, C. W.: Software Reuse, *ACM Computing Surveys (CSUR)*, Vol. 24, No. 2, pp. 131–183 (1992).

[12] Kula, R. G., German, D. M., Ouni, A., Ishio, T. and Inoue, K.: Do developers update their library dependencies?, *Empirical Software Engineering*, pp. 1–34 (2017).

[13] Kula, R. G., Roover, C. D., German, D., Ishio, T. and Inoue, K.: Visualizing the Evolution of Systems and their Library Dependencies, *Proceedings of the 2nd IEEE Working Conference on Software Visualization (VISSOFT'14)*, pp. pp.127–136 (2014).

[14] Lee, T., Nam, J., Han, D., Kim, S. and In, H. P.: Micro interaction metrics for defect prediction, *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pp. 311–321 (2011).

[15] Lim, W. C.: Effects of Reuse on Quality, Productivity, and Economics, *IEEE Software*, Vol. 11, No. 5, pp. 23–30 (1994).

[16] McCarey, F., Ó Cinnéide, M. and Kushmerick, N.: Knowledge Reuse for Software Reuse, *Web Intelligence and Agent Systems*, Vol. 6, pp. 59–81 (2008).

[17] Mileva, Y. M., Dallmeier, V., Burger, M. and Zeller,

© 2018 Information Processing Society of Japan

A.: Mining Trends of Library Usage, *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol'09)*, pp. 57–62 (2009).

[18] Mirhosseini, S. and Parnin, C.: Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?, *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, pp. 84–94 (2017).

[19] Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., Lo, D. and Sun, C.: Duplicate bug report detection with a combination of information retrieval and topic modeling, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pp. 70–79 (2012).

[20] Ouni, A., Kula, R. G., Kessentini, M., Ishio, T., German, D. M. and Inoue, K.: Search-based software library recommendation using multi-objective optimization, *Information and Software Technology*, Vol. 83, pp. 55 – 75 (2017).

[21] Sun, C., Lo, D., Khoo, S.-C. and Jiang, J.: Towards more accurate retrieval of duplicate bug reports, *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, pp. 253–262 (2011).

[22] Surowiecki, J.: *The wisdom of crowds*, Anchor Books (2005).

[23] Thung, F., Lo, D. and Lawall, J.: Automated library recommendation, *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*, pp. 182–191 (2013).

[24] Wang, X., Zhang, L., Xie, T., Anvik, J. and Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information, *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE'08)*, pp. 461–470 (2008).

[25] Yano, Y., Kula, R. G., Ishio, T. and Inoue, K.: VerXCombo: An Interactive Data Visualization of Popular Library Version Combinations, *Proceedings of the IEEE 23rd International Conference on Program Comprehension (ICPC'15)* (2015).

[26] Zhang, C. and Zhang, S.: *Association rule mining: models and algorithms*, Springer-Verlag (2002).

© 2018 Information Processing Society of Japan