コンテナ開発における
OSSの法的リスク特定自動化に関する研究

令和5年3月

和歌山大学大学院システム工学研究科

東　裕之輔

Automating OSS Legal Risk Identification for Container Development

March 2023

Graduate School of Systems Engineering
Wakayama University

Yunosuke Higashi

概 要

近年，コンテナ仮想化技術の急速な普及によりアプリケーションの配布方式が変化してきている．コンテナを活用したシステム開発では，アプリケーションと動作環境を一つに統合したファイル（以下，Docker イメージ）をリリースする．コンテナ仮想化環境はデファクトスタンダードとなっている Docker により提供される．Docker の環境さえあれば，Docker Hub 等のイメージ共有サイトから Docker イメージをダウンロードするだけでアプリケーションをすぐに起動できる．

従来よりコスト削減のため，オープンソースソフトウェア（以下，OSS）を再利用したシステム開発が盛んに行われてきた．OSS を再利用するには，ソースファイル，もしくはパッケージ単位に設定されているライセンスを遵守することが義務付けられている．OSS のライセンスに誤って違反する法的リスクを避けるためには，まず，利用する予定の複数の OSS の (1) ライセンスの特定，及び (2) OSS 間のライセンスの互換性を検証する必要がある．しかしながら，上記 2 種類の作業を行うためには課題がそれぞれ存在する．1 つはソースファイルのライセンス特定ツールは存在するが，ライセンス名を紐づけるためのライセンスルールの保守が複雑で，未知のライセンスに対応できないという課題である．もう 1 つは，OSS パッケージを 1 つの Docker イメージに統合する際に，ライセンス間で条項に矛盾が発生していないか（互換性）を検証する必要があるが，パッケージ情報が出揃わなければ検証できないという課題である．上述の 2 つの課題が未解決な現状においては，計画通りにアプリケーションをリリースできないという事態を生じさせている．

本研究では，上述の 2 つの課題を解決するための自動化技術を構築する．課題 (1) に対しては，今後，未知のライセンスに対応できるようにするため，ライセンス特定ツールに登録すべきルールを自動生成する仕組みを構築する．まず，ライセンス特定ツールにより，未知のライセンスと判定されたソースファイルからライセンス記述を抽出する．次に個々のライセンスにマッチするルールを抽出するため，ライセンス記述を階層クラスタリングでライセンス単位に分類する．次に，系列パターンマイニングにより各クラスタのライセンス記述に共通して頻出する英単語の系列パターンを抽出する．最後に系列パターンを元に正規表現に変換しライセンスルールとして出力する．提案手法を評価するために，FreeBSD-10.3.0, Linux-4.4.6, Debian-7.8.0 から未知のライセンスとして検出された 1,821, 3,561, 2,838 件のソースファイルを用いて実験を行った．その結果，提案手法は，既存のライセンス特定ツールには登録されていないライセンスルールを生成可能であることを示した．また，生成したライセンスルールは，1 つのルールがマッチするライセンス記述の数が多く，特定可能なライセンスが 2%–10%増加することが分かった．

課題 (2) に対しては，Docker イメージとして統合される複数の OSS パッケージ間のライセンス互換性検証を開発早期に実施できるよう自動化する．OSS のパッケージ情報は Docker イメージの開発ともに増加すると仮定し，その上で再利用する全てのパッケージの情報が出揃っていない開発早期の時点でも最終的なライセンス互換性検証結果を機械学習により予測する．まず，各 Docker イメージ内に含まれるパッケージの利用情報と互換性検証結果をベクトル化しデータセットを作成する．パッケージ情報が不十分な状態を再現するため，データセット全体の 0%〜90%まで 5%間隔の割合に相当するパッケージ情報をマスキングしながら，多層パーセプトロン（MLP）による学習を行う．Github から抽出した Dockerfile をビルドして生成した 598 件の Docker イメージをもとに評価実験を行った．その結果、開発進捗度 10%の時点でも 適合率 94%，再現率 96%，F 値 95%の精度でライセンス互換性検証結果を予測できることが分かった．

Abstract

In recent years, the rapid spread of container virtualization technology has changed the method of application distribution. In system development using container virtualization technology, the developer releases an image file (e.g., Docker image) that integrates an application and excution environment. Container virtualization technology is provided by Docker, which has become the de-facto standard. In a Docker environment, applications can be launched quickly after downloading a Docker image from an image-sharing site.

In order to reduce the development costs, open source software (OSS) is widely reused for application in software development. OSS is reusable as a part of a software product if it strictly complies with OSS licenses described in source files or software packages. In order to avoid the legal risk of OSS license, it is necessary to (1) identify licenses of multiple OSS components used and (2) verify the compatibility of licenses among OSS components. However, there are issues that need to be solved for the above two types of work. The first issue is that although there are tools for identifying source file licenses, the management of license rules is multipled and cannot be maintained, and as a result, it is not possible to deal with unknown licenses. Second, when integrating OSS packages into a single Docker image, it is necessary to verify whether there are any incompatibilities between licenses, but this cannot be verified until the package design is complete. The two issues are causing the failure to release the application as planned.

This study builds an automation technology to solve these issues. For issue (1), A method is proposed to automatically generate license rules in order to handle unknown licenses in the future. First, license descriptions are extracted from source files with unknown licenses detected by a license identification tool. Then, source files that are determined to be unknown licenses are classified by license name. Next, filtering by the Levenshtein distance is applied to license statements belonging to each cluster to extract the license statements for which license rules should be generated. After that, sequential pattern mining is used to extract patterns of the license statements. Finally, the patterns are converted to regular expressions. The proposed method is applied to 1,821, 3,561 and 2,838 source files with unknown licenses respectively collected from FreeBSD v10.3.0, Linux Kernel v4.4.6, and Debian v7.8.0, to confirm the usefulness of proposed method. The results show that the proposed method can generate license rules that are not registered in existing license identification tools. The proposed method generated rules with a large number of matches of license descriptions in a single rule, and the number of licenses that can be identified increases by 2%–10%.

For issue (2), A machine learning prediction method in the early development phase was proposed to automate the license compatibility verification for Docker images. First, Docker images are vectorized by their package information. Next, the package usage information is masked according to each development progress level. Finally, multi-layer perceptron (MLP) prediction model is built a for each progress level. Evaluation study is conducted using 598 Docker images generated by building Dockerfiles extracted from Github. The results show that the model can predict the license compatibility verification results with 94% of precision, 96% of reproducibility, and 95% of F-value even at 10% of development progress level.

目 次

第 1 章 はじめに **1**
1.1 研究の背景と目的 1
1.2 本論文の構成 3

第 2 章 コンテナ開発における OSS の法的リスク **4**
2.1 OSS の再利用とライセンス 4
2.2 OSS ライセンスにおける法的係争事例 7
2.2.1 著作者と利用者との関係 7
2.2.2 GPL の法的係争事例 9
2.3 コンテナ仮想化技術の台頭 9
2.4 コンテナの開発プロセス 10
2.5 コンテナ開発における法的リスク 12
2.6 コンテナ開発における法的リスク特定の課題 14
2.6.1 ライセンス特定に必要なライセンスルールの保守 14
2.6.2 課題１：手作業によるライセンスルールの保守 16
2.6.3 課題 2：コンテナ開発におけるライセンス互換性検証 18
2.7 本研究の目的 19

第 3 章 コンテナ開発における法的リスク予備調査 **21**
3.1 予備調査 1：未知ライセンスの発見的調査 21
3.1.1 調査方法 21
3.1.2 RQ1: 大規模 OSS ではどの程度未知ライセンスが検出されるか 23
3.1.3 RQ2: 未知ライセンスには何種類のライセンスが含まれているか 23
3.2 予備調査 2：コンテナのライセンス互換性分析 26
3.2.1 調査方法 26
3.2.2 RQ3: GPL と非互換なライセンスはパッケージにどの程度適用されているか 29
3.2.3 RQ4: ライセンスの互換性問題はどれくらいの Docker イメージに発生するか 30
3.3 OSS の法的リスク特定自動化のための技術的課題 31
3.3.1 T1-1：未知ライセンスが検出されたライセンス記述の分類 32
3.3.2 T1-2：生成する正規表現数の最小化 33
3.3.3 T2：開発早期におけるライセンス互換性検証結果の予測 34

第 4 章 ライセンスルール自動生成手法の開発 **36**
4.1 ライセンスルール自動生成手法 36
4.1.1 処理 1 : ライセンス記述の抽出 37
4.1.2 処理 2 : GPL/BSD family ライセンスの分類 38
4.1.3 処理 3：ライセンス記述の階層クラスタリング 40
4.1.4 処理 4：ライセンス記述のフィルタリング 42
4.1.5 処理 5 : BIDE による系列パターンマイニング 43

4.1.6 処理 6：記述パターンから正規表現への変換 44
4.2 ケーススタディ 45
4.2.1 データセット 46
4.2.2 閾値 46
4.3 実験結果 48
4.3.1 RQ5: GPL/BSD family ライセンスを何%の精度で分類できるか 48
4.3.2 RQ6: 単一のライセンスからなるクラスタは全体の何%か 50
4.3.3 RQ7: 編集距離を用いて単一のライセンスからなるクラスタにフィルタリングすることができるか 53
4.3.4 RQ8: どれくらいの正規表現を生成できるか 54
4.3.5 RQ9: 生成した正規表現はどれくらいの未知ライセンス文とマッチするか . 56
4.4 考察 58
4.4.1 GPL/BSD family の分類精度向上 58
4.4.2 ライセンスルール生成に適した階層クラスタリングの条件 58
4.4.3 ライセンスルール生成数の最小化に向けて 59
4.4.4 手作業によるライセンスルール作成の効率化 60
4.4.5 制約 60
4.5 関連研究 61
4.5.1 ライセンス特定ツール 61
4.5.2 ライセンス違反検出 61
4.5.3 OSS のライセンスとソフトウェア開発 62
4.5.4 系列パターンマイニング 62
4.6 まとめ 63

第 5 章 Docker イメージ開発時におけるライセンス非互換の予測 65
5.1 ライセンス互換性検証結果予測 65
5.1.1 パッケージ情報を用いた Docker イメージのベクトル化 66
5.1.2 開発進捗度に合わせたパッケージ利用情報のマスキング 66
5.1.3 多層パーセプトロンによる予測モデルの構築 67
5.2 実験方法 68
5.2.1 データセット 68
5.2.2 機械学習アルゴリズムの評価方法 69
5.3 実験結果 69
5.3.1 RQ10：開発進捗度 100%のパッケージ情報を学習した場合，どれくらい開発早期からライセンス互換性を予測できるか 69
5.3.2 RQ11：開発進捗度を考慮したパッケージ情報の学習により開発早期での予測精度は向上するか 73
5.4 考察 76
5.4.1 GPL 非互換予測精度と開発進捗度の関係 76
5.4.2 最適な予測モデル 77

5.4.3 GPL 非互換となるパッケージのインストール経路 79
5.4.4 制約 81
5.5 関連研究 81
5.5.1 Docker イメージの開発 81
5.5.2 コンテナ品質向上のための Dockerfile 分析 82
5.6 まとめ 83

第 6 章 おわりに **84**

謝辞 **85**

研究業績 **86**

第1章　はじめに

本章では，本研究の背景・目的について説明する．

1.1　研究の背景と目的

近年，コンテナ仮想化技術の急速な普及 [1] により OSS の再利用が加速化している．コンテナ仮想化は，OS カーネル部分はホストコンピュータと共有し，アプリケーションと動作環境のみをイメージ化することで，従来の仮想化技術より軽量でかつ複数の環境を跨いだシステム構築を可能にする．コンテナ仮想環境はデファクトスタンダードとなっている Docker[1]により提供される．Docker は Linux Container（LXC）をベースとした技術を採用 [2] しており，Linux ディストリビューション上でのみ動作する．そのため，ホストマシンとカーネルを共有する各コンテナには多数の OSS パッケージが含まれる．また，コンテナ仮想環境内で動作するアプリケーションやミドルウェアは，あらかじめイメージファイル（以下，Docker イメージ）に埋め込まれているため，Docker イメージさえあれば，構築作業を行わずにすぐにアプリケーションを起動することができる．この仕組みにより，従来ソフトウェア単体でリリースしていたところ，動作環境も合わせてユーザに提供することができる．作成した Docker イメージは，Docker Hub[2]などのイメージ共有サイトでリリースされるのが一般化しつつある．Docker Hub では現在 9 百万以上 (2022 年 9 月 26 日時点) のイメージがシェアされている．

以前よりアプリケーション開発では，開発コスト削減や最新技術の取り込みを主たる目的として，オープンソースソフトウェア（OSS）を再利用した開発が盛んに行われている．OSS はソースファイル，もしくはパッケージ単位にライセンスが設定されており，それらを遵守することで企業で開発するソフトウェアプロダクトに OSS を統合することができる．ただし，OSS のライセンスは全て同様に設定されているわけではない．OSS には，別の異なる OSS が含まれていることが多く，ファイル単位やパッケージ単位では例外的にライセンスが異なる場合がある [3]．そのため，OSS を再利用する場合は，OSS に含まれる全てのソースファイルとパッケージのライセンス特定に加えて，OSS 間でのライセンスの互換性の検証が必要となる．

コンテナ仮想化技術を用いたアプリケーション開発（以降，コンテナ開発と呼ぶ）では，OSS ライセンスの特定はアプリケーションに加え Docker イメージ内のパッケージに対しても行う必要がある．Linux Foundation が 2020 年に発表した Docker イメージのライセンスに関するホワイトペーパー [4] では，Docker イメージに含まれるソフトウェアのライセンスは全て遵守しなければならないとされている．Docker イメージ内に含まれるパッケージのライセンスを正しく遵守しなかった場合，最悪の場合訴訟問題に発展し，社会的信用を失い，多額の損害賠償を負う可能性がある．しかしながら，Docker イメージには多数のパッケージが含まれているため，コンテナ開発の

[1] https://www.docker.com/
[2] https://hub.docker.com/

中で発生する法的リスクを正しく把握することは容易ではない．OSS のライセンスに誤って違反する法的リスクを避けるため，アプリケーション，Docker イメージでそれぞれに発生する 2 つの課題を解決する必要がある．まず，ライセンスをソースファイル単位で自動で特定するライセンス特定ツールがあるものの，ライセンスルールの保守に労力がかかり，手作業で特定が必要な未知ライセンス（Unknown）が数多く出力されるという課題がある．パッケージでは，1 つの Docker イメージに統合されるため，ライセンス間で条項に矛盾が発生していないか（互換性）を検証する必要がある．ただし，パッケージ情報が確定しなければ検証できないという課題がある．リリースの直前でパッケージを変更することになった場合，リリース時期を遅らせるしかない．

本研究では，これらの課題を解決するため，OSS の法的リスク特定のための自動化技術を提案する．まず，ソースファイルのライセンス特定ツールの保守を自動化するため，ライセンスルール（正規表現）の自動生成手法を提案する．ライセンス特定のためのライセンスルール（正規表現）自動生成手法は，以下の 6 つの処理から構成される．

1. 未知ライセンスと判定されたソースファイルから，ライセンス記述を抽出する
2. 著名なライセンスのライセンス記述をフィルタリングする
3. ライセンスルールを抽出するのに適したクラスタを作成するため，ライセンス記述の階層クラスタリングを行う
4. 各クラスタ内で外れ値となるライセンス記述を編集距離を用いてフィルタリングする
5. 各クラスタに対し系列パターンマイニングアルゴリズムを適用し記述パターンを抽出する
6. 記述パターンを正規表現に変換しライセンスルールとして出力する

提案手法を評価するため，FreeBSD-10.3.0, Linux-4.4.6, Debian-7.8.0 から検出した 1,821, 3,561, 2,838 件の未知ライセンスを用いてケーススタディを行った．その結果、提案手法は最小限のライセンスルールでより多くのライセンスの文を識別できること，提案手法で作成したライセンスルールを Ninka に追加することで，ライセンスルールのパフォーマンスが 2%–10%向上することを示した．

また，Docker イメージ内のパッケージのライセンス互換性検証を機械学習を用いて開発早期に行えるように自動化する．Docker イメージのライセンス互換性検証自動化は，以下の 3 つの処理から構成される．

1. Docker イメージをそのイメージ内に含まれるパッケージ情報によりベクトル化する
2. 各開発進捗度に合わせて，パッケージ情報をマスキングする
3. 各開発進捗度ごとに多層パーセプトロンによる予測モデルを構築する

提案手法を評価するために，Github から抽出した Dockerfile をビルドして生成した 598 件の Docker イメージをもとに評価実験を行った．その結果、開発進捗度 10%の時点でも適合率 94%，再現率 96%，F 値 95%の精度でライセンス検証結果を予測できることが分かった．

1.2 本論文の構成

本稿の構成は以下の通りである．2 章では，OSS ライセンスの定義やコンテナの開発方法，開発中に発生する法的リスクについて説明する．3 章では，2 章で定義した法的リスクが実際にどの程度発生しているのかを確認するために行った予備調査について述べる．4 章では，本研究で提案するライセンスルール自動生成手法について説明し，有用性評価のために行ったケーススタディについて説明する．5 章では，ライセンス互換性検証結果予測手法について説明し，有用性評価のために行ったケーススタディについて説明する．最後に 6 章では，本研究のまとめと今後の展望について述べる．

第2章　コンテナ開発におけるOSSの法的リスク

本章では，本研究での問題点となるコンテナの開発過程で発生する OSS の法的リスクと本研究の目的について説明する．本章の構成は次のようになる．2.1 章では，OSS を再利用する際に遵守しなければならない OSS ライセンスついて説明する．2.2 章では，OSS ライセンスの違反に関する法的係争事例について説明する．2.3 章では，コンテナ仮想化技術と Docker イメージの構造について説明する．2.4 章では，本研究が想定するコンテナの開発プロセスについて説明する．2.5 章では，コンテナの開発プロセスで発生する法的リスクについて説明する．2.6 章では，コンテナ開発の過程で発生する法的リスク特定における課題について説明する．2.7 章では，本研究の目的について説明する．

2.1　OSS の再利用とライセンス

近年のソフトウェア開発では，コスト削減のためにソフトウェアを再利用することが一般的になりつつある [5]．ソフトウェアの再利用とは既存のソフトウェアを用いて新たなソフトウェアを構築することである．ソフトウェアを利用するには著作者からソフトウェアのライセンスを得なければならない．ライセンスとは著作物の利用許諾と再利用する際に遵守しなければならない条件である．ソフトウェアの再利用は以下の 3 つ行為から構成される [6].

- 複製：ソフトウェアをコピーする
- 配布：ソフトウェアを他人に渡す
- 改変：ソフトウェアを修正する/一部を取り出す/新機能を追加する

OSS の場合，OSS の著作者により指示された OSS ライセンスを読んで理解し，遵守することでその OSS を利用することが可能となっている．一般的な商用ソフトウェアのライセンスと OSS ライセンスの大きな違いは，ソフトウェアライセンスは，そのソフトウェアの再利用を認める必要がないのに対し，OSS ライセンスは再利用を基本的に認めている点にある．

OSS ライセンスは，一般的に，Open Source Initiative（OSI）[1]や Free Software Foundation（FSF）[2]のようなライセンス承認機関に承認されているライセンスから選択されることが多い．選択された OSS ライセンスは，ソースファイルのコメント部分に記述される場合（図 2.1）と，パッケージのメタファイルに記述される場合（図 2.2）がある．メタファイルに記述されるライセンスは，具体的にパッケージ内のどのファイルを指しているかが不明なため，パッケージの一部を再利用するためのライセンス情報としての活用は期待できない．Wolter ら [7] は，Github リポジト

[1] https://opensource.org
[2] http://www.fsf.org

/*
This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2, or (at your option) any later
version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Place - Suite 330, Boston, MA 02111-1307, USA.
*/

図 2.1: GPLv2+のライセンス記述の例

$brew info wget
wget: stable 1.21.3 (bottled), HEAD
Internet file retriever
https://www.gnu.org/software/wget/
Not installed
From: https://github.com/Homebrew/homebrew-core/blob/HEAD/Formula/wget.rb
License: GPL-3.0-or-later
== Dependencies

図 2.2: wget パッケージのメタ情報として表示されるライセンス

リのメタファイルで指定しているライセンスとそのソースファイルのライセンス間の差異を調査したところ，約半数のリポジトリのメタファイルには，ソースファイルで見つかった全てのライセンスが記載されていないことを明らかにしている．そのため，メタファイルのライセンス情報は，個々のソースファイルのライセンスの管理ではなく，主にソフトウェアのサプライチェーン管理など，システム全体のライセンスを大まかに把握する場面での活用に向いているといえる．

一方，ソースファイルのライセンスは，そのファイル自体に直接記述されており，ライセンスの宣言範囲が明確なため，ソフトウェアの再利用時の情報として参照される．OSS ライセンスは，最も細かいレベルでは，ソースファイル単位で決定されている．そのため，OSS を再利用する場合，その OSS に含まれる全てのソースファイルのライセンス記述を確認する必要がある．特に本研究ではソースファイルでライセンスを指示している記述をライセンス記述と呼ぶ．

OSS ライセンスには，多くの種類が存在する．OSS ライセンスの承認機関の 1 つである OSI には，101 種類（2022 年 9 月 27 日時点）のライセンスが承認されている．他にも OSS のコンサル

ティング会社である Synopsys 社が所有する BlackDuck KnowledgeBase[3]には 2750 種類（2022 年 9 月 27 日時点）以上の OSS ライセンスが登録されている．その中でも，特に幅広く利用されている GNU General Public License (GPL)，GNU Lesser General Public License (LGPL)，BSD License (BSD)，Apache License (Apache) を紹介する．

GNU General Public License (GPL) 1989 年，Emacs や GCC の開発者である Richard M. Stallman により提唱され，Free Software Foundation により定められた歴史のあるライセンスである．GPL の大きな特徴は，二次著作物を配布する場合，同じ GPL ライセンスで配布する必要がある点である．一般的に，このような性質はコピーレフトと呼ばれ，ある OSS を再利用・改変して開発されたソフトウェアも OSS としてソースコードを公開することを要求する．さらに GPL は，GPL のソフトウェアとリンクして実行されるソフトウェアも同じソフトウェアとして見なすため，コピーレフトの適用範囲が広いライセンスとして知られている．したがって，GPL のソフトウェアを再利用して配布する場合，独自で開発した部分も GPL により公開しなければならなくなる可能性が高く，一般的な商用ソフトウェアとの相性は良くないと考えられている．

GNU Lesser General Public License (LGPL) GPL と同様に Free Software Foundation が定めたライセンスである．GPL と同じくコピーレフト条項を持ち，LGPL でライセンスされたソフトウェアの二次著作物を配布する場合は，LGPL ライセンスとする必要がある．しかし，LGPL でライセンスされたソフトウェアを動的リンクで利用する場合のみ，同一のライセンスにする必要はない．

BSD License (BSD) 1980 年代末に，カリフォルニア大学バークレー校（UC Berkeley）により提唱されたライセンスである．元々UNIX 互換 OS である BSD を配布するために用いられたライセンスである．著作者の名前を勝手に使用しないことや，無保証であることなどの条件は存在するが，GPL や LGPL とは異なり，コピーレフト条項は持たない．そのため，二次著作物を同一のライセンスにする必要がなく，商用ソフトウェアにもソースコードを取り込むことができる．

Apache License (Apache) Apache Software Foundation (ASF) が定めたライセンスである．主に Apache HTTP Server に適用されてきた．コピーレフト条項をもたないという点に関しては BSD ライセンスに似ているが，Apache License には特許に関する条項がある点で異なる．Apache License には，OSS の開発者が特許事項を含んだコードをコミットしていたとしても，OSS の利用者にはそれらを自由に利用できる特許ライセンスが付与されることが明記されている [3].

また，OSS ライセンスの条項は様々な理由により変更される．例えば，GPL は，自由なソフトウェアの利用を妨げるハードウェアによる制限やデジタル著作権管理（DRM）を避けるため，制約が厳しくなるようにライセンスの条項を修正した [8]．一方，BSD はユーザのニーズに対応するため，制約を緩和する方向に条項が 2 回に渡って修正されてきた．このような背景から，OSS ライセンスは複数のバージョンが作成されることがあり，それらは全て法的解釈が異なる独立した

[3]https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis/knowledgebase.html

表 2.1: 本稿で使用するライセンス名とその略称

略語.	説明
Apache	Apache License
AGPL	GNU Affero General Public License
BSD4	BSD 4-clause License
BSD3	BSD 3-clause License
BSD2	BSD 2-clause License
CeCill	Cea Cnrs Inria Logiciel Libre License
CDDL	Common Development and Distribution License
GPL	GNU General Public License
LGPL	GNU Lesser General Public License
LibraryGPL	GNU Library General Public License
MIT	MIT License
MPL	Mozilla Public License
publicDomain	パブリックドメインとされているライセンス
RSA-MD	RSA Message-Digest License
ZPL	Zope Public License

ライセンスとして扱われる．本論文では，ライセンスが複数のバージョンを持つ場合，特定のバージョンを示すために接尾辞 “v” を用いる．ただし，“v” を接尾辞に用いない場合は特定のバージョンを問わない．本論文では “or later” を接尾辞 “+” で表現する．“or later” とは，例えば，GPLv2 のライセンス記述に “either version 2 of the License, or (at your option) any later version” と記載されている場合，version 2 以降のバージョンのうちのどれかでの再頒布が可能ということを意味する．最後に，本稿で使用するライセンス名とその略称を表 2.1 に示す．

2.2 OSS ライセンスにおける法的係争事例

OSS の再利用は OSS ライセンスを遵守することが義務付けられている．もし，OSS ライセンスを正しく遵守しなかった場合，最悪の場合訴訟問題に発展し，社会的信用を失い，多額の損害賠償を負う可能性がある．本研究では，OSS を再利用することで企業に生じる損失の危険性を法的リスクとして扱う．

2.2.1 著作者と利用者との関係

本章では，法的係争事例について述べる前に，OSS ライセンスの遵守における著作者と OSS を利用する開発者との関係について説明する．

一般的なソフトウェアライセンスは、著作権者が提示する使用許諾書にその著作物の利用者が署名したときに成立すると規定されている [3]．一般的な個人向けソフトウェア商品でも，パッケー

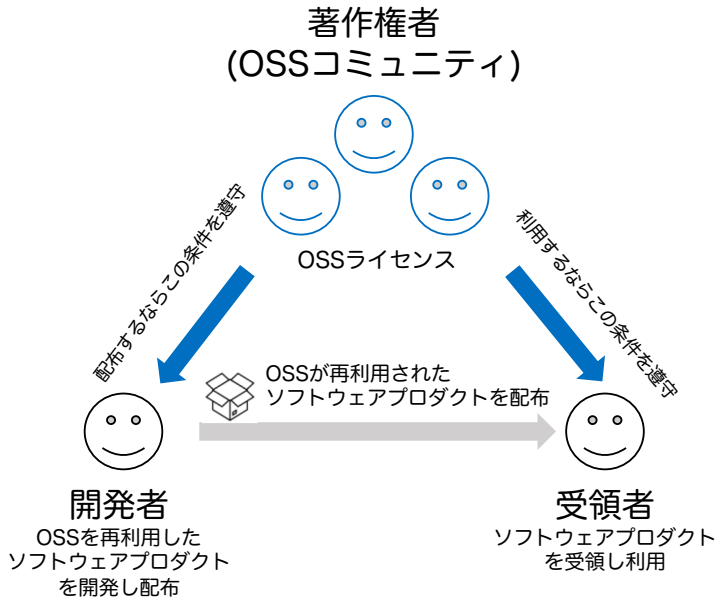


図 2.3: OSS ライセンスにおけるコミュニティ・開発者・受領者の関係
(上田理 (著), 岩井久美子 (監), OSS ライセンスの教科書, 2018 中の 図 2.4 を参考に作成)

ジの開封が署名と同等の行為を持つシュリンクラップ契約[4]や，ソフトウェアのダウンロード時に求められるライセンス文書への同意など，何らかの方法で著作者から利用許諾を得る手続きが必要である．

一方で OSS の場合は，2.1 章で説明した利用の 3 つの行為（複製，配布，改変）は基本的に認められているため，OSS の利用者は，OSS コミュニティ（著作者）に直接的に利用許諾を取る必要が無い．その OSS を利用するタイミングで，提示されたライセンス条件に合意したとみなすのが一般的である．ただし，企業の立場で OSS の法的リスクを扱う場合，配布のタイミングは重要である．図 2.3 に OSS コミュニティと，OSS を再利用してソフトウェアプロダクトを開発する企業の開発者やその受領者とのライセンスの遵守義務の関係を示す．OSS を再利用してソフトウェアプロダクトを開発した場合，ライセンスの遵守義務はソフトウェアプロダクトを配布する開発者，そのプロダクトの受領者の双方に発生する．しかしながら，一般的な OSS ライセンスでは，配布行為に制約や条件を課しているものが多い．例えば，GPL ライセンスは配布の際にソースコードを開示する義務を課している[5]．そのため，法的係争事例の多くは GPL ライセンスと配布に関するものがほとんどである．図 2.3 の場合，開発者が再利用している OSS が GPL ライセンスであった場合，ソースコード開示義務が発生する．反対に受領者は，単なる利用のみの場合，ライセンス遵守義務は発生するものの，事実上制約がないことが多いのが実態となる（但し，複製や改変でも受領者に制約を課すライセンスも存在する）．

[4]経済産業省:電子商取引及び情報財取引等に関する準則 https://www.meti.go.jp/press/2020/08/20200828001/20200828001-1.pdf
[5]GPLv3 全文日本語訳:https://mag.osdn.jp/07/09/02/130237

2.2.2 GPL の法的係争事例

本章では，過去発生した GPL のライセンス違反に関連する判例の中から以下の 2 つの事例について紹介する．

Skype の Linux 電話機販売における GPL 違反[6]

2007 年 7 月 24 日，Skype は，GPLv2 でライセンスされた Linux が使用されている VoIP 電話機を販売したが，ソースコードを公開していなかったため，GPL ライセンスに違反しているとの判決が下された．電話機はスペインの通信機器販売ベンダである SMC Network が製造したもので，法廷は Skype による SMC 製品の販売差し止めを命令した．

PS2 ゲームソフト「ICO」における GPL 違反[7]

2007 年 11 月 29 日，PS2 版の ICO のソースコードには GPL でライセンスされたライブラリ「libarc」が利用されていることがゲームのプレイヤーによって発見された．2008 年 2 月に ICO の生産を終了し，廃盤を決定した．

このように，OSS ライセンスの違反は，企業にとって経済的に大きな損害となる．また，社会的なイメージも低下することも懸念される．このような損害を被らないようにするという点においても，再利用する OSS に含まれる全てのソースファイルのライセンスを正しく遵守しなければならないと言える．

2.3 コンテナ仮想化技術の台頭

近年，新規システムのリリースや運用を迅速に行うため，コンテナ仮想化技術の利用が急速に普及してきている [9] [10]．従来のハイパーバイザ型の仮想化技術は，OS のカーネル部分を仮想化するのに対し，コンテナ仮想化技術は，OS カーネル部分をホストマシンと共有する．これにより，仮想化イメージが軽量化され仮想化環境の運用・管理が容易になるメリットがある．

コンテナ仮想化を行うソフトウェアとして Docker がある．Portworx 社，Aqua Security 社による調査[8]では，Docker は，IT 関連の会社の 87%で利用されていると報告されている．Docker を利用してアプリケーションと動作環境をイメージファイル化することで，クラウド，オンプレミスなどのプラットフォームを問わず，コンテナを高速で起動したり，運用したりすることができる．例えば，アプリケーションサーバが稼働するプラットフォームの変更や，必要なハードウェアリソースに応じてコンテナ数の調整（サーバ台数の増強）を自動で行うことができる．本研究ではこのイメージファイルのことを Docker イメージと呼び，Docker イメージの開発のことをコンテナ開発と呼ぶ．また，Docker 社は，作成した Docker イメージをインターネット上で共有する Docker Hub と呼ばれるサービスを展開している [11]．不特定多数の開発者は，Docker Hub から Docker イメージをダウンロードすることで，アプリケーションを構築作業なしにすぐに利用することができる．

---

[6] スカイプ、携帯電話販売で GPL 違反–ドイツの法廷で判決 (cnet.japan) : https://japan.cnet.com/article/20353432/

[7] PS2 ゲームソフト「ICO」における GPL 違反 (スラッシュドット) : http://slashdot.jp/story/07/12/02/1340209/.

[8] Portworx 社 Aqua Security 社による調査 : https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

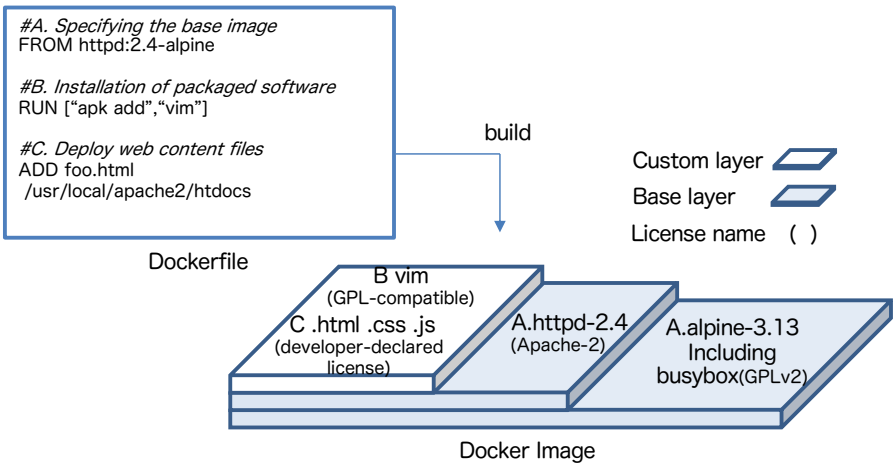


図 2.4: Dockerfile と Docker イメージのレイヤー構造との関係

Docker イメージは，Dockerfile と呼ばれるコンテナ構成ファイルを作成し，ビルドすることで作成することができる．図 2.4 に Dockerfile と Docker イメージの概要を示す．Dockerfile には主に以下の 3 つの処理命令の記述から構成される．

(a) **ベースイメージの継承:** 通常の Dockerfile の多くは，まず，FROM 構文で既存の Docker イメージ名を指定する．Docker イメージは，Dockerfile のビルド時の各工程をレイヤ化して保持する構造となっている．一部のコンテナ開発者はスクラッチで Dockerfile を記述する場合もあるが，一般的な開発者は既存のイメージに独自のレイヤ（必要なパッケージ等）を追加して新たにイメージを作成することが多い．本研究では継承する既存イメージをベースイメージと呼ぶ．そして，ベースイメージにより作成されたレイヤをベースレイヤと呼び，開発者自身が追加したレイヤをカスタムレイヤと呼ぶ．図 2.4 では，FROM 構文にて httpd:2.4-alpine をベースイメージとして指定している．httpd:2.4-alpine は，Linux distribution の一つである Alpine-linux[9]に，web サーバソフトウェアである Apache-httpd がインストールされたイメージである．

(b) **追加パッケージのインストール:** コンテナの中で任意のシェルコマンドを実行する際は RUN 構文を利用する．一般的には Linux ディストリビューションにバンドルされている OSS パッケージのインストール，環境パラメータの設定を RUN 構文であらかじめ行なっておく．図 2.4 では，RUN 構文にてメンテナンスのために利用する Vim のようなテキストエディタをインストールしている．

(c) **アプリケーションのデプロイ:** 開発したアプリケーションファイルをコンテナの中にデプロイするには ADD 構文を利用する．図 2.4 では，ADD 構文にて Web アプリケーションをデプロイしている．

2.4 コンテナの開発プロセス

コンテナを用いたシステム開発は，アプリケーションと Docker イメージの両方の開発が必要となるため，従来より多くの開発工程が存在する．2017 年に，Microsoft 社はコンテナを用いたシス

[9]https://www.alpinelinux.org/

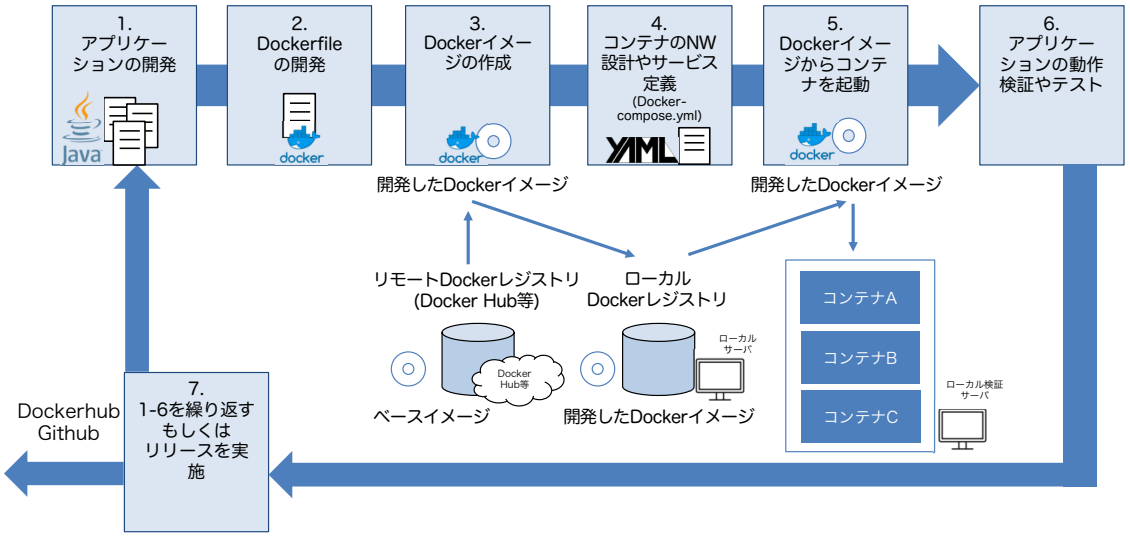


図 2.5: コンテナで稼働するアプリケーションの開発のための 7 つのステップ
(Microsoft 社: .Net Microservices: Architecture for Containerized .NET Applications 中の図 5-1 を元に作成)

テム開発を簡単に始めることができるよう，コンテナの開発環境のアーキテクチャのガイドラインを公開している [12]．本章では，このガイドラインを基に，本研究が前提としているコンテナ開発プロセスについて説明する．なお，本ガイドラインは，Microsoft 社製品である.NET アプリケーションとその統合開発環境である Visual Studio の利用が前提となっている．そのため，.NET は Java 言語，Visual Studio は単に統合開発環境として一般化して記述する．図 2.5 にコンテナ開発のプロセスを示す．本ガイドラインによれば，コンテナ開発は以下の 7 つのステップで構成される．

ステップ 1： アプリケーションの開発 統合開発環境上で，Java 言語等の標準的なプログラミング言語でアプリケーションのコーディングを行う．アプリケーションはテスト・検証単位で定期的にビルドし，実行ファイルをソフトウェアリポジトリに格納する．

ステップ 2： Dockerfile の開発 ベースイメージとそのバージョン，アプリケーションの実行ファイルのファイルパス，動作に必要なライブラリやパッケージのインストールコマンド，アプリケーションの実行コマンドが記述された Dockerfile を開発する．

ステップ 3： Docker イメージの作成 Dockerfile をビルドし，アプリケーションが埋め込まれた Docker イメージを作成する．このとき Dockerfile で指定されたベースイメージは，インターネット上の Docker Hub からダウンロードされる．作成された Docker イメージはローカル環境の Docker レジストリに格納される．

ステップ 4： コンテナのネットワーク設計やサービス定義 開発したいシステムが複数のコンテナで構成される場合，コンテナ間をネットワークで接続する必要がある．そのため，docker-compose と呼ばれる yaml 形式のファイルに，ステップ 3 で作成された Docker イメージ名，コンテナ間の通信に用いるポート番号や URL を記述する．

ステップ 5 : Docker イメージからコンテナを起動 docker-compose ファイルをビルドし，Docker イメージからコンテナを起動する．コンテナはローカルレジストリ内にある docker-compose ファイル内で指定された Docker イメージから起動される．コンテナは次のステップで検証やテストが必要なため，本ステップではローカルの検証用サーバで実行される．

ステップ 6 : アプリケーションのテスト 起動しているコンテナに対してアプリケーションの結合テストやセキュリティテストを実施し，各コンテナとシステム全体の品質を検証する．

ステップ 7 : ステップ 1-6 を繰り返す，もしくはリリースの実施 ステップ 1-6 を繰り返して，最終的にコンテナを完成させる．完成したコンテナは Docker Hub と Github 等[10]のコード共有サイトでリリースされる．コンテナには既にアプリケーションが統合されているため，Docker Hub 上でのみリリースすればよいが，従来の仮想マシン上で稼働させたいといった旧来のニーズに対応するため，Docker Hub に加えてアプリケーションのみを Github でリリースするといった形式もとられる．

本研究では，主にステップ 1〜3 で発生する OSS の法的リスクを特定することを目的としている．現在，OSS ライセンスの管理は利用する OSS が確定するステップ 6 のタイミングで行われると想定されるが，2.2.2 章で説明した GPL の係争事例が発生していることを鑑みると，ステップ 6 での OSS の法的リスクの管理だけでは限界がある．ステップ 6 でライセンス違反が発覚し，該当の OSS を変更した後に再度テストを行うことはリリーススケジュールに多大な影響を及ぼすため，ステップ 6 のチェックのみで完全に回避することは困難である．本研究では，ステップ 6 の最終的な品質チェックのタイミングより前に OSS の法的リスクを特定することを前提に，以降 2.5 章，2.6 章にかけて OSS の法的リスク特定の課題について述べる．

2.5 コンテナ開発における法的リスク

コンテナ仮想化技術は，ソフトウェアの実行とデプロイに必要な時間を削減する一方で，複雑な法的課題を生み出している．2020 年 4 月，Hemel ら [4] は，Linux Foundation より，Docker コンテナを配布・デプロイする際の OSS ライセンスの取り扱いについて記述したホワイトペーパーをリリースしている．Docker イメージ全体のライセンスに準拠するには，Docker イメージ内に含まれる各ソフトウェアのライセンスに遵守する必要があると述べている．Docker イメージには，アプリケーションとそれを動作させるための言語環境やライブラリ等，非常に多くのソフトウェアが含まれている．そのため，含まれているソフトウェアが多ければ多いほど法的リスクは増加するといえる．

法的リスクを排除するためには，まず，再利用する OSS の全てのライセンスを確認することが必要である．OSS ライセンスの確認は，ソフトウェアを統合する以下の 2 つのタイミングで実施すべきとされている [3].

- 外部から OSS を持ち込む時
- ソフトウェアをビルドする時

[10] https://github.co.jp/

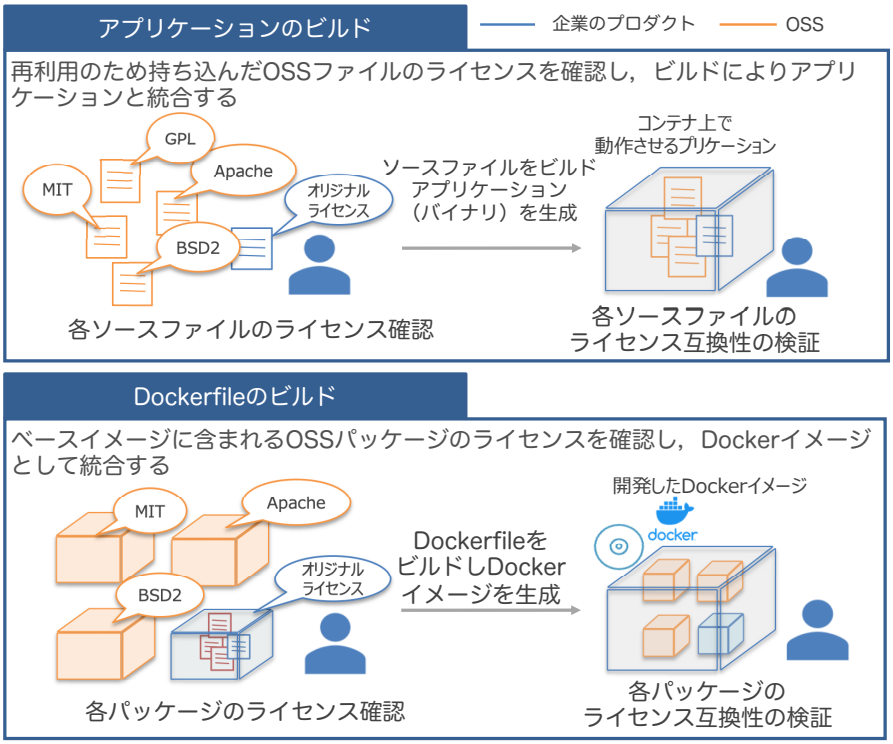


図 2.6: コンテナ開発で発生するソフトウェア統合と必要となる法的リスクの特定

外部から OSS を持ち込むタイミングでは，持ち込む OSS のライセンスを事前に確認することで，遵守できないライセンスの OSS の利用を取り止めることができる．ソフトウェアのビルドでは，アプリケーションやコンテナと依存関係にあるパッケージやライブラリが統合される．ビルド後に生成される成果物に対してさらにライセンスの確認を行うことでより正確にライセンスを管理することができる．

2.4 章で示したコンテナ開発を想定した場合，再利用される OSS は，ステップ 1 のアプリケーションに再利用される OSS と，ステップ 2-3 での Docker イメージに含まれる OSS パッケージの 2 つに分けられる．図 2.6 にソースファイルと Dockerfile のビルドでのライセンス確認内容を以下に示す．

ソースファイルのビルド：コンテナで稼働するアプリケーションを開発する場合，まず，Docker イメージで中で動作させたいアプリケーションを開発する必要がある．アプリケーションの一部に OSS を再利用することは一般的になりつつあるが，開発者は法的知識がない状態で Github などのソーシャルコーディングサービスからアドホックにソースコードを再利用することが知られている [13]．そのため，アプリケーションの開発コスト削減のため持ち込まれる OSS のライセンス管理が必要となる．具体的には，再利用する OSS に含まれる全てのソースファイルのライセンスを確認することが必要である．

Dockerfile のビルド：Dockerfile の開発では，既存のベースイメージを継承する開発が主流となっている [4]．ベースイメージは最終的に組み込まれるアプリケーションのライセンスと整合性が取れるイメージを選択する必要がある．Docker Hub にアップロードされている既存イメージには，アプリケーションを動作させるために必要なプログラム言語やテキストエディタ，アプリケーションサーバソフトウェアなどのミドルウェアが OSS パッケージとしてインストールされている．そ

のため，開発者はDockerfileに継承するベースイメージ名を記述する前に，利用するベースイメージ内に存在するソフトウェアパッケージや，追加するソフトウェアパッケージを洗い出しそのライセンスを確認する必要がある．

2.6 コンテナ開発における法的リスク特定の課題

現在，ソフトウェア工学の分野では，前述した2種類のライセンス管理を支援するため，ライセンス名を自動で出力するライセンス特定ツールが提案されている．しかしながら，技術的な課題により，コンテナ開発のライセンス管理は容易ではない状況である．以降2.6.1章，2.6.2章，2.6.3章でそれらの課題について説明する．

2.6.1 ライセンス特定に必要なライセンスルールの保守

ソースファイルのライセンス特定ツール

OSSのソースファイルを再利用してアプリケーションを開発する場合，再利用する全てのライセンス記述（図2.1）を確認する必要がある．しかしながら，再利用したいOSSのソースファイルが数多く存在する場合，ライセンス確認作業は大きな労力が必要である．そのため，ソースファイルのライセンス名を自動で特定するライセンス特定ツールが提案されている [14–18]．ライセンス特定ツールは，機械学習を用いた手法ではなく，ライセンス記述と既知のライセンスの対応関係を示すルール（ライセンスルール）をあらかじめ用意しておき，再利用対象となるソースファイルのライセンスを特定するものが主流となっている．ライセンス記述には，バージョンやGPLのオプションなどの細かな文字列の差異により法的意味が異なるものが存在するため，機械学習を用いて一般化することは難しいと考えられる．

ライセンスルールを用いてライセンスを特定する手法には，文字列の類似度を用いるもの [15] と正規表現を用いるもの [14, 16–18] がある．類似度を用いる場合，ライセンスルールとして登録されている既知のライセンス記述の類似度を算出する．そして，類似度が閾値以上であった場合にはライセンスルール内のライセンス記述と対応関係にあるライセンス名を特定結果として出力する．類似度を用いたライセンス特定ツールには，入力されるライセンス記述とライセンスルール内のライセンス記述が完全に一致していなくても特定結果を出力できるメリットがある．ただし，文字列を厳密に区別する必要のあるライセンスのバージョン等の特定には向いていないというデメリットがある．

正規表現を用いる場合，厳密な文字列の違いを区別できるため，ライセンスのバージョンなどの法的な意味が異なる細かな文字列の差異を区別できるメリットがある．その反面，法的に意味が同一な文字列の差異（以降，ライセンス記述のバリエーション）がライセンス記述に少しでも含まれていた場合，正規表現によるテキストマッチングに失敗するというデメリットを持つ．ライセンス記述のバリエーションに対応するには，同じライセンスを特定するライセンスルールを別に作成する必要がある．さらに，たとえライセンス名が同じであってもバージョンが異なれば，ライセンス条項が異なるため別のライセンスとして特定する必要がある．

ライセンス特定ツールの1つにNinka [14] がある．Ninkaは，OSSの分析に基づいて作成した正規表現をライセンスルールとして用いる．Ninkaの概要を図2.7に示す．Ninkaは，その他のラ

図 2.7: Ninka の概要

イセンス特定ツールと同じように，入力されたライセンス記述に対応するライセンス特定結果（ライセンス名）を出力する．一方で，対応するライセンスルールが搭載されていないライセンス記述が入力された場合，Ninka は未知のライセンスであることを出力する．Ninka は，未知ライセンスの出力のため，ソースファイル中のライセンス記述を抽出する．これにより，Ninka に搭載されているライセンスルールにマッチしなかったとしても，ライセンス記述の抽出に成功していれば，ライセンス特定結果として未知ライセンスを出力することができる．ライセンス記述は，ソースファイルのコメント部分に “copyright”，“redistribute”，“conditions” などの法的なキーワード（計 82 語）が存在する文として抽出される．ライセンス記述が 1 文以上抽出された場合，ライセンス記述がソースファイルに存在すると判定され，後続のライセンスルールとのマッチングに移行する．未知ライセンスの出力は現状 Ninka のみで提案されており，Ninka 以外のツールは，ライセンス記述が存在しないと誤判定するしかない．したがって，他のツールに比べ Ninka はライセンスの適合率が最も高くなっている [14]．

その一方で，未知のライセンスが多数存在する場合，全て目視により人手で判定する必要がある．未知のライセンスが大量に検出されるような OSS に対しては，Ninka によるライセンス特定効率化の効果が限定的になる場合がある．German ら [14] が行なったライセンス特定ツール評価では，再現率は 82.3%であり 4 つのライセンス特定ツールの中で最も低いことが分かった．そのため，Ninka の高い適合率を維持しつつ目視作業を減らすためには，Ninka によって未知ライセンスが検出されたソースファイルからライセンスルールを作成し，Ninka に適宜追加することが望ましいと考えられる．

Ninka のライセンスルール

未知ライセンスが検出されたソースファイルのライセンスを特定できるようにするには，新たなライセンスルールを作成し追加する必要がある．ライセンス特定ツールの代表として，Ninka を例にライセンスルールの構成について説明する．Ninka のライセンスルールは大きく分けて次の 2 種類から構成される．

ライセンス文特定のための正規表現 ライセンス記述を文単位で特定するための正規表現である．ライセンス記述における文とは，先頭からピリオドまでの一文を指す．例えば，BSD2 ライセンスは，計 5 種類のライセンス文が存在する．それぞれの文を特定するための 5 種類の正規表現が搭載されている必要がある．表 2.2 に正規表現の例を示す．BSD2 のライセンス文 “BSDcondSource” を特定するルールに正規表現のメタ文字が利用されていることが分

表 2.2: Ninka の BSD2 のライセンス文特定のための正規表現（一部抜粋）

ルール名	ライセンスルール	ライセンス記述
BSDpre	Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:	Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
BSDcondsource	Redistributions? of source code must retain the (above)?copyright notice, this list of conditions(,)? and the following disclaimer (, without modification)?:	1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
BSDbinarycond	Redistributions in binary form must reproduce the above copyright notice(*s*)?, this list of conditions(,)? and the following disclaimer(s)? in the documentation and/or other materials provided with the distribution:	2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
BSDasIsExtrict	THIS SOFTWARE IS PROVIDED BY (.+)“AS IS” AND ANY EXPRESS(ED)? OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED:	THIS SOFTWARE IS PROVIDED BY (COPYRIGHT HOLDER) ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
BSDWarrExtrict	IN NO EVENT SHALL (.+) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE:	IN NO EVENT SHALL (COPYRIGHT HOLDER) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

かる．例えば，“Redistribution” と “Redistributions” の両方のバリエーションには，0 回または 1 回の繰り返しを意味する正規表現のメタ文字 “?” を用いて，1 つの正規表現で両方に対応できるよう作成されている．デフォルトでは，Ninka 作成時の調査の際に発見された表記揺れを 1 つのルールで特定できるよう実装されている．

ライセンス名の出力ルール 1 つ以上の正規表現の組み合わせから，ライセンス名を特定するためのルールである．ライセンスルールの組み合わせとマッチするルールが Ninka に登録されている場合はそのライセンス名を，Ninka に登録されていなかった場合は未知のライセンス（Unknown）を出力する．BSD3 を出力するためのライセンスルールを（ライセンス名）:（ライセンス文名）の形式で以下に例を示す．

BSD3 : BSDpre, BSDcondSource, BSDcondBinary, BSDcondEndorseRULE, BSDasIs, BSDWarr

2.6.2 課題１：手作業によるライセンスルールの保守

本章では，検出された未知ライセンスからライセンスルールを人手で作成する場合，どのような作業が必要なのかを明らかにし，ライセンスルール作成を自動化する意義を説明する．ライセンスルール作成を自動化する前提として，人手によるライセンスルール作成は，以下の 4 つのタスクからなると想定する．

Task 1. 未知ライセンスの分類

全ての未知ライセンスが検出されたソースファイルを人手で確認し，それぞれのライセンス名を特定する．特定されたライセンス名に基づき分類する．

表 2.3: ライセンスルール作成タスクと本研究の自動化範囲

	Ninka [14] (正規表現)	FOSSology [15] (類似度の閾値)	提案手法 (正規表現)
1. 未知ライセンスの分類	手作業	手作業	**自動化**
2. ライセンス記述の表記ゆれの調査	手作業	なし (※)	**自動化**
3. ライセンスルール作成	手作業	手作業	**自動化**
4. ライセンスの名前づけ	手作業	手作業	手作業

(※) Fossology はライセンス特定のためにライセンス記述の類似度の閾値を採用しており，記述の細かな差異をライセンスルールに反映する必要がない．ただし，この仕様によりライセンス特定の精度が低くなると考えられる．

Task 2. ライセンス記述のバリエーション調査

タスク 1 により得られた分類から，ライセンスルールに反映すべき複数のライセンス記述で出現する記述（頻出フレーズ）を抽出する．加えて，頻出フレーズからライセンス記述による表記揺れを調査する．例として，“This **software** is free” と “This **program** is free” という頻出フレーズが得られたとすると，“software” と “program” の表記ゆれを確認できる．

Task 3. ライセンスルールの作成

Step2 で得られた頻出フレーズと表記揺れから正規表現を作成する．例えば，“This **software** is free” と “This **program** is free” の両方の頻出フレーズにマッチする正規表現は，“This **(software| program)** is free” とすればよい．

Task 4. ライセンスの名付け

正規表現の作成完了後，その正規表現に適切な名称をつける．未知ライセンスに一般的に知られている名称が存在する場合はその名称をつけるが，新規ライセンスや開発者オリジナルのライセンス等，特定の名称が存在しないライセンスであった場合は他のライセンスと区別できるよう名称を考える必要がある．

一般的な開発者が，以上のステップで正規表現を作成する場合，相当な労力がかかることが予想される．まず，Task1 では，未知ライセンスが検出されたファイル数に応じて，ライセンスを特定にかかる労力が増加してしまう．German [14] らが行った Ninka の評価実験では，実験対象となった 250 のソースファイル中，43 ファイル (17.2%) に対し，未知のライセンスが検出されたことを示している．また，Task2 では，意味的には同じだが表記揺れが多数存在するライセンス記述には，多くの頻出フレーズが存在する．それらを全て人手で抽出するには，文字単位・単語単位の比較を行わなければならず，人為的ミスも発生しやすい．さらに，未知ライセンスが検出される原因である，新しいライセンスや未知の表記揺れは今後増加していく傾向にある．

このような背景から，ライセンス特定の適合率維持のために，ライセンスルールの作成を自動化することが望ましいといえる．そのため，本研究では，検出された未知ライセンスからライセンスルールを自動的に抽出する手法を提案する．表 2.3 に本研究で自動化するライセンスルール作成タスクを示す．本研究では，Task1–Task3 までを自動化の対象とする．Task4 のライセンスの名前付けは自動化の対象外としている．Task4 のライセンスの名前付けは，他のライセンスと区別するために法的専門知識が必要であるため，機械的に行うのは限界があると考えている．

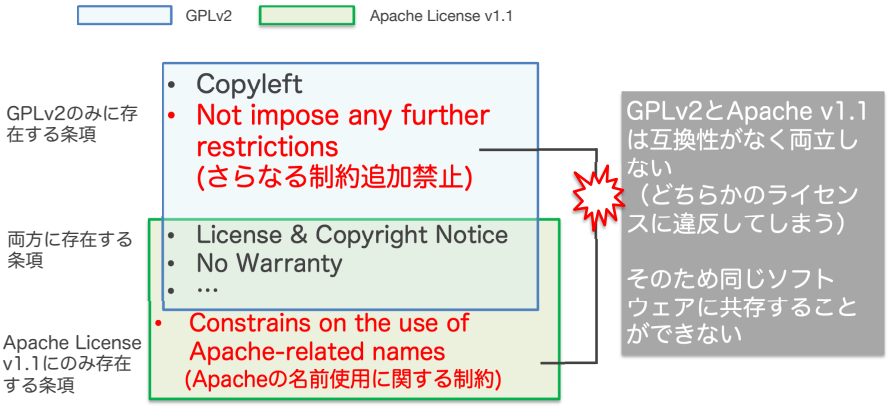


図 2.8: GPLv2 と Apache License v1.1 のライセンス互換性の問題

2.6.3 課題 2：コンテナ開発におけるライセンス互換性検証

Docker イメージにはアプリケーションとアプリケーションの動作に必要な OSS パッケージが統合されている．複数のソフトウェアを 1 つのソフトウェアとして統合する場合，個々のソフトウェアのライセンス確認に加え，それぞれのソフトウェアのライセンス条項が矛盾しないかの確認も必要である．例えば，GPL ライセンスには，GPL の OSS を再利用して開発したものは同じ GPL ライセンスとしてリリースしなければならないというコピーレフト条項がある．これは，仮に GPL 以外のライセンスでリリースしなければならないソフトウェアに，GPL の OSS を再利用する場合，ライセンスを同時に満たすことができないことを意味する．本研究では，この状態をライセンスの互換性が無い状態として扱う．

図 2.8 に GPLv2 と Apachev1.1 に互換性の問題が生じる例を示す．GPLv2 には，さらなる制約を追加してはならないという条項がある．一方で Apache License v1.1 には，同ライセンスの OSS を再利用して開発された二次著作物に Apache の名前を許可なしで使用してはならないという制約を課す．この 2 つの条項は同時には成り立たないため，互換性が無いライセンスとして認識されている[11]．

Tern によるコンテナ内の OSS パッケージのライセンス特定

複数の OSS パッケージ間のライセンスの互換性を検証するには，まず個々のパッケージのライセンスを特定することが必要である．図 2.2 に示したように，OSS パッケージのライセンスは各パッケージのメタデータに記載される．しかしながら，Docker イメージ内の OSS パッケージを抽出，それぞれのライセンスを特定するのは手間がかかる．Docker イメージ内の OSS パッケージのライセンス特定を支援するため，VMware 社が Tern というツールをリリースしている．Tern 概要を図 2.9 に示す．Tern は Docker イメージに導入されている OSS パッケージの一覧とそのライセンス名を出力する．

[11] https://www.gnu.org/licenses/license-list.ja.html#GPLIncompatibleLicenses

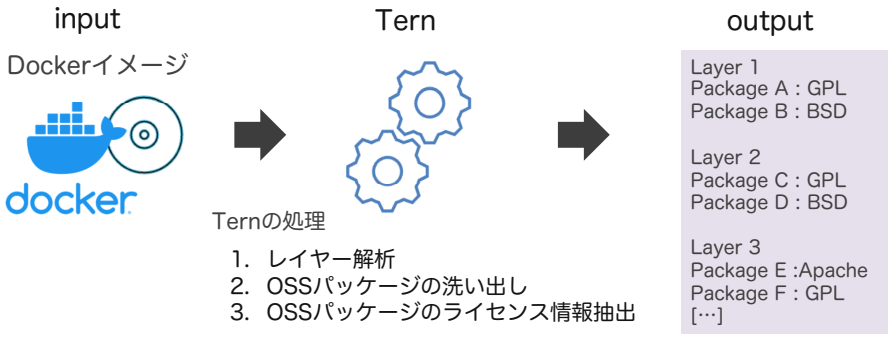


図 2.9: Tern の概要

コンテナ開発初期でのライセンス互換性検証

一般的な開発者が非常に多くの種類がある各 OSS ライセンスを理解することも容易ではないが，OSS ライセンス間に互換性があるかどうかを網羅的かつ間違いなく判断するのはそれ以上に容易ではない．そのため，ソフトウェア工学の分野では，OSS ライセンスの互換性の検証を支援するための研究がおこなわれている．German ら [19]，Manabe ら [20] は，ソフトウェアパッケージに適用されている OSS ライセンスとそのソースファイルに適用されている OSS ライセンスの互換性に関する分析を行なっている．Kechagia ら [21] は，FreeBSD Ports/Packages Collection からソフトウェアパッケージの依存関係を取得し，依存関係があるパッケージ間でどのように OSS ライセンスが適用されているかを分析する手法を提案した．Mlouki ら [22] は，Android アプリに適用されている OSS ライセンスの種類と互換性を分析した．Golubev ら [23] や [24] は，コードクローン生成に伴う，ライセンス互換性を含むライセンス違反について分析している．Liu ら [25] は，ソースコードを変更する際，その変更内容とライセンス互換性の関係から変更後のライセンスを予測する手法を提案している．Xu ら [26] は，ライセンス条項を自然言語処理で解析し，ライセンス互換性を確認する手法を提案している．

しかしながら，Docker イメージ内における OSS ライセンスに関する研究は行われておらず，現在のライセンスの互換性における法令遵守状況には大きな懸念がある．ライセンスの互換性のない複数の OSS が含まれている Docker イメージを再利用した場合，ライセンス違反となるリスクがある．また，OSS パッケージはソースファイルとは異なり，ライセンス互換性に問題が見つかった場合の手戻りコストは大きい．OSS パッケージの依存関係は多重構造で複雑なため，開発後期でのシステムアーキテクチャの変更は，問題が見つかったパッケージ以外にも波及する可能性がある．

本研究では，Docker イメージ内のソフトウェアパッケージの OSS ライセンスの互換性について調査を行い，ライセンスの互換性がない OSS の開発中の Docker イメージ内への混入を開発初期段階で防ぐ手法を提案する．

2.7 本研究の目的

2.6.2 章，2.6.3 章では，アプリケーション，Docker イメージそれぞれの法的リスクの特定について述べた．本研究では，それらの法的リスク特定で生じる問題を自動化により解消することを

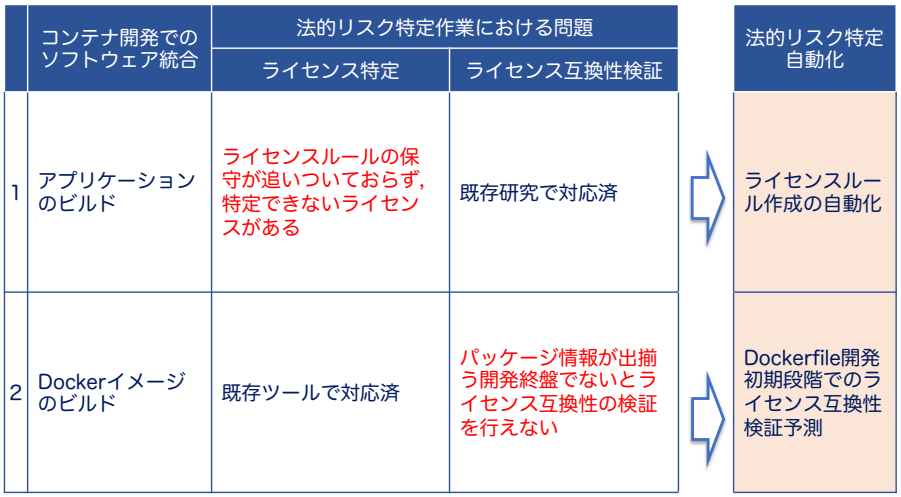


図 2.10: 本研究で取り組む課題の概要

最終目標とする．本研究で取り組む課題の概要を図 2.10 に示す．

アプリケーションのライセンス管理では，ソースファイル統合時のライセンス互換性検証技術は複数の既存研究 [19,20] で提案されている．しかしながら，個々のソースファイルのライセンス特定においては，正規表現作成の負荷が高く，新たなライセンスに対応するためのライセンスルールが作成されないという課題がある．本研究では，**特定できなかったライセンスからライセンスルールを自動生成する研究に取り組む．**

Docker イメージのライセンス特定では，ソフトウェアパッケージ管理基準である SPDX (Software Package Data Exchange) が国際標準化[12]されたことにより，個々のパッケージのライセンスは機械判読可能になりつつある．しかしながら，パッケージ間のライセンス互換性検証技術 [27,28] は提案されているものの，Docker イメージに着目した研究や，ライセンス互換性検証後のパッケージの変更対応がコンテナのリリースに与える影響を考慮した研究はまだない．本研究では，ライセンス互換性検証後のプロジェクトの影響を軽減するため，**Docker イメージ開発初期でのライセンス互換性検証を予測する手法を提案する．**

[12]https://www.iso.org/standard/81870.html

第3章　コンテナ開発における法的リスク予備調査

本章では，コンテナ開発での法的リスク特定を自動化するにあたり，現時点で判明している法的リスク特定における問題が，実際にどの程度発生するかを把握するための予備調査を行う．

3.1　予備調査1：未知ライセンスの発見的調査

2.7章では，ソースファイルをビルドする際に，個々のソースファイルのライセンスを特定しておく必要がある点について述べた．ライセンス特定ツールNinka [14]は，搭載されているライセンスルールに存在しないライセンス記述が入力された場合，未知ライセンスであることを出力する．未知ライセンスが多く検出されるほど，作成しなければならないルール数が増加し，手作業によるライセンスルール作成は現実的ではなくなる．本章では，以下の2つのリサーチクエスチョンについて取り組む．

- **RQ1: 大規模OSSではどの程度未知ライセンスが検出されるか**
- **RQ2: 未知ライセンスには何種類のライセンスが含まれているか**

3.1.1　調査方法

データセット

本研究では，2013年にリリースされたNinka-1.1を用いて検出された未知ライセンスを扱う．未知ライセンスの調査対象とするOSSプロジェクトの概要を表3.1に示す．対象プロジェクトは，Ninka-1.1リリースから2-3年後にリリースされた大規模なOSSプロジェクトである，FreeBSD-10.3.0，Linux-4.4.6，Debian-7.8.0の3つとする．Linux-4.4.6とFreeBSD-10.3.0はそれぞれのソースファイルから検出された未知ライセンスを扱うが，Debian-7.8.0はより幅広く未知ライセンスを収集するためにDebianプロジェクトが管理するソフトウェアパッケージのソースファイルから検出される未知ライセンスを扱う．FreeBSD-10.3.0，Linux-4.4.6はオープンソースのOSとして幅広く利用されている．また，主要なライセンスはそれぞれ，FreeBSD-10.3.0はBSD2，Linux-4.4.6はGPLv2とそれぞれ異なっており，両データセットのライセンスの分布が異なっていると考えられる．それぞれのプロジェクトからNinkaが対応できるC，C++，Java，Python，Perl，Lisp言語で書かれたソースファイルを対象に抽出したところ，FreeBSD-10.3.0では，20,759ファイル，Linux-4.4.6では，22,310ファイルとなった．Debianは厳格なソフトウェアパッケージの管理を行なっているLinuxディストリビューションとして知られている．Debianプロジェクトのパッケージ管理者らはDebianポリシーと呼ばれる文書[1]に基づき，ソフトウェアパッケージを再配布可能

[1] https://www.debian.org/doc/debian-policy/index.html#

表 3.1: 対象プロジェクト

	ソースファイル数	種別	リリース
FreeBSD-10.3.0	20,759	UNIX 系 OS ディストリビューション	2016 年 3 月
Linux-4.4.6	22,310	Linux カーネル	2016 年 3 月
Debian-7.8.0	12,725	ソフトウェアパッケージ集合	2015 年 1 月

なライセンスなパッケージ (main) と，再配布不可なライセンスであり，Debian 上で使用できるが Debian の一部ではないパッケージ (contrib/non-free) に区別している．本予備調査では，OSS ライセンスを主に対象にしているため，再配布可能なライセンス (main) に属する全 12,725 の各ソフトウェアパッケージから 1 ファイルずつソースファイルをランダム抽出してデータセットを作成した．

目視によるライセンス特定

予備調査では，まず，表 3.1 の 3 つのプロジェクトのソースファイルを Ninka に入力し，未知ライセンスが検出されたファイルを取得する．次に，取得したファイルのライセンスについて目視による特定を行う．目視によるライセンスの特定は，Linux Foundation が運営しているライセンス承認機関，SPDX が Web 上に公開しているライセンスリストのライセンス名とそのライセンス記述を参考に行う．なお，同一のライセンス名であってもライセンスのバージョン，オプションなどの違いにより法的な意味が異なる場合がある．しかしながら，SPDX のライセンスリストだけ目視を行うと，一部分類できないライセンスが出現することが予想される．例えば，Melocaら [29] が行った調査では，1,058,554 プロジェクトのうち約 22 %のプロジェクトでは，OSI に承認されていないライセンスが適用されたソースファイルを 1 つ以上保有していたと報告している．このような OSI に承認されていないライセンスは以下のように取り扱い分類する．

著作権の記述　Di Penta ら [30] は，ソースコードの著作権の管理は，最終的にはそのライセンス条項を強制することができる法人であるため重要と考えており，ソースコードの著作権のオーナーを追跡する手法を提案している．本研究でも著作権記述の有無は法的リスクがあると考えるため，目視時は他のライセンスと同様に “Copyright” ライセンスとして扱う．

ライセンスファイルへのリンク，ライセンスなし　Ninka では，LICENSE.txt 等のライセンスが示しているファイルへのリンクを示すようなライセンス記述（SeeFile）や，ライセンス記述がそもそも存在しないソースファイル（NONE）もライセンス特定結果として出力する．ライセンスが存在しないファイルは単に著作権でのみ保護されるため，著作者の許可がない限り再配布することができない．Debian ポリシーでもライセンスが無いソフトウェアは，配布することができないソフトウェアとして区別されている[2]．そのため，Ninka でも未知のライセンスを出力することと同様に重要だと考えライセンス特定結果として出力するよう実装されている．そのため，今回の目視による調査でも同様にライセンスファイルへのリンクのあるライセンス記述を “SeeFile” ライセンス，ライセンス記述が存在しない場合は “NONE”” ライセンスとして扱う．

[1] https://spdx.org/licenses/
[2] https://www.debian.org/legal/licenses/index.ja.html

カスタムライセンス OSS ライセンスには，OSI や SPDX のようなライセンス承認機関以外にも開発者によって作成されたライセンスも存在し，そのほとんどは既存ライセンスを少しカスタマイズして作成されたものであることが知られている．既存ライセンスをカスタマイズしたライセンスは，既存ライセンスに対する差異や表記揺れの内容が既存ライセンスと法的に意味が異なるかどうかに依存する．もし既存ライセンスと法的に意味が異なると判断する場合は “can not name” ライセンスとして扱う．

不正なライセンスバージョン また，“GPLv2.1” など，本来存在しない（ライセンス記述を書いた開発者が LGPLv2.1 と混同した）バージョンが記述されている場合がある．ただし，LGPLv2.1 もしくは GPLv2 のどちらとして扱ってよいか断定できないため，本予備調査では “GPLv2.1” として記述されたバージョンをそのまま扱う．

3.1.2 RQ1: 大規模 OSS ではどの程度未知ライセンスが検出されるか

動機： Ninka は入力されたライセンス記述に対応するライセンスルールが存在しなかった場合，ライセンス特定結果として未知ライセンスであることを出力する．未知ライセンスが検出された場合，そのライセンス名を知るには目視で確認する必要がある．さらに Ninka がそのライセンス名を特定できるようにするには，利用者自らライセンスルールを作成し Ninka に追加する必要がある．現状，大規模な OSS での未知ライセンスに関する調査事例はないため，どの程度未知ライセンスが検出されるのか検証する．

アプローチ： 表 3.1 で示した，3 プロジェクトのソースファイル集合に対して Ninka でライセンス特定を行う．検出された未知ライセンスの数を集計し，そのソースファイル総数に対する未知ライセンスの検出割合を算出する．

調査結果： 各プロジェクトのソースファイルのライセンスを Ninka で特定し，検出された未知ライセンス数を表 3.2 に示す．FreeBSD-10.3.0 では，1,821 ファイルに検出され，全体のソースファイルに対する割合は 8.8%であった．最もソースファイルが多い Linux-4.4.6 では，3,561 ものファイルに未知ライセンスが検出された．未知ライセンスの割合は 16.0%であった．また，Debian-7.8.0 では，2,838 ファイルに未知ライセンスが検出され，その割合は 22.3%であった．これは Debian-7.8.0 のデータセットは各パッケージから 1 つずつソースファイルを抽出して作られたデータセットであることが起因していると考えられる．以上の結果から RQ1 の回答は以下とする．

RQ1 への回答：

未知ライセンスは FreeBSD-10.3.0 のソースファイルのうち 8.8%，Linux-4.4.6 では 16.0%で検出された．また，Debian-7.8.0 のソフトウェアパッケージから抽出したソースファイルでは，全体の 22.3%で検出された．

3.1.3 RQ2: 未知ライセンスには何種類のライセンスが含まれているか

動機： ライセンスルールの自動作成へ向けて，Ninka が検出する未知ライセンスにはどのようなライセンスが存在するか調査し，その傾向を知る必要がある．

表 3.2: 未知ライセンス検出数

	ソースファイル数	未知ライセンス数	未知ライセンスの割合
FreeBSD-10.3.0	20,759	1,821	8.8%
Linux-4.4.6	22,310	3,561	16.0%
Debian-7.8.0	12,725	2,838	22.3%

```
Copyright (c) ****.  All rights reserved.
Subject to the following obligations and disclaimer of warranty use and redistribution of this
software in source or object code forms with or without modifications are expressly permitted by
**** provided however that:

1.  Any and all reproductions of the source or object code must include the copyright notice above and
the following disclaimer of warranties; and
2.  No rights are granted in any manner or form to use ****.  trademarks including the mark ‘‘****’’
on advertising endorsements or otherwise except as such appears in the above copyright notice or in
the software.

THIS SOFTWARE IS BEING PROVIDED BY ****‘‘AS IS’’ AND TO THE MAXIMUM EXTENT PERMITTED BY **** MAKES
NO REPRESENTATIONS OR WARRANTIES EXPRESS OR IMPLIED REGARDING THIS SOFTWARE INCLUDING WITHOUT
LIMITATION ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY FITNESS FOR A PARTICULAR PURPOSE OR
NON-INFRINGEMENT. **** DOES NOT WARRANT GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF OR
THE RESULTS OF THE USE OF THIS SOFTWARE IN TERMS OF ITS CORRECTNESS ACCURACY RELIABILITY OR OTHERWISE.
IN NO EVENT SHALL **** BE LIABLE FOR ANY DAMAGES RESULTING FROM OR ARISING OUT OF ANY USE OF THIS
SOFTWARE INCLUDING WITHOUT LIMITATION ANY DIRECT INDIRECT INCIDENTAL SPECIAL EXEMPLARY PUNITIVE OR
CONSEQUENTIAL DAMAGES PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES LOSS OF USE DATA OR PROFITS HOWEVER
CAUSED AND UNDER ANY THEORY OF LIABILITY WHETHER IN CONTRACT STRICT LIABILITY OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE EVEN IF **** IS ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

図 3.1: カスタムライセンス (BSD-like) の例
****は固有名詞に対するマスキングの意

アプローチ： RQ1 の調査結果で検出された全ての未知ライセンスについて，3.1.1 章で説明した点に留意し目視でライセンス名を特定する．その結果について集計し，最も多く含まれていたライセンスを調査する．

調査結果： それぞれの未知ライセンスの内訳を表 3.3 に示す．未知ライセンスについて目視した結果，Debian-7.8.0 のソフトウェアパッケージが最も多く 194 種類の未知ライセンスが見つかった．また，FreeBSD-10.3.0 では 69 種類，Linux-4.4.6 では 33 種類の未知ライセンスが見つかった．RQ1 では，Linux-4.4.6 のほうが FreeBSD-10.3.0 より 2 倍近くの未知ライセンスが検出されたが，反対に未知ライセンスの種類は FreeBSD-10.3.0 の方が多く存在していることが分かった．

FreeBSD-10.3.0 では，全種類の BSD ライセンスが見られた．これらは既知のライセンスであるが，そのライセンス記述に未知の表記揺れが含まれていた．また，BSD ライセンスの書き方で作成されたライセンス（BSD-like）も 53 ファイルで発見されている．実際に見つかったカスタムライセンスの例を図 3.1 に示す．条項の内容は BSD2（表 2.2）とは異なるが，書き方は類似している．Ninka は未知のライセンスとして出力しているものの，その他のライセンス特定ツールが誤検知しやすい例の 1 つと考えられる．

Linux-4.4.6 では著作権のみの記述が未知ライセンスとして多数検出された．ライセンス記述は通常 Copyright の宣言が始まるものが多いが，Ninka には著作権のみを記述した場合のルールが無いため未知ライセンスとして出力されたことが原因である．著作権以外では，やはり GPLv2 のライセンスが最も多く占めている．Debian-7.8.0 でも，Linux-4.4.6 と同様に著作権のみの記述が最も多く検出されている．一方で，SeeFile や NONE など，ソースファイルではライセンスを宣言していないソースファイルが他のプロジェクトより多く見られた．これは Ninka によるライセンス記述抽出の誤りによるものである．SeeFile は一般的な自然言語でライセンスファイルへのリン

表 3.3: 各プロジェクトで検出された未知ライセンスの内訳

	FreeBSD-10.3.0（全 69 種類）		Linux-4.4.6（全 33 種類）		Debian-7.8.0（全 194 種類）	
	ライセンス名	ファイル数	ライセンス名	ファイル数	ライセンス名	ファイル数
1	BSD2	416	Copyright	1778	Copyright	377
2	SeeFile	265	GPLv2only	1116	SeeFile	354
3	BSD4	174	GPL	176	NONE	338
4	BSD3	157	NONE	123	Perl	288
5	PublicDomain	98	SeeFile	113	GPL	158
6	MIT-CMU	98	GPLv2+	101	GPLv2+	146
7	Copyright	61	MIT	33	GPLv2only	131
8	OpenSSL	60	GPLv2only and BSD3	20	MIT-CMU	64
9	Custom(BSD-like)	53	GPLv2+ and BSD3	20	PublicDomain	63
10	NONE	40	Unicode-TOU	11	MIT	52
	Other(59)	399	Other(23)	70	Other(184)	867

クが記述されているため，既存の正規表現では対応できなかったことによるものだと考えられる．また，NONE はソースコード中のコメントを誤検知してしまったことが原因である．一方，ライセンス自体では Perl ライセンスや GPL が多く含まれていることがわかった．

全プロジェクトに共通しているのは，未知ライセンスのほとんどは新規のライセンスではなく，既知のライセンスの記述のバリエーションに対応できていないことであった．Debian-7.8.0 では，Ninka に反映されていない MPLv2.0（2012 年 1 月リリース）の新規ライセンスも 5 ファイル含まれていたが，多くは既知のライセンスの記述バリエーションが正規表現で対応できてないことが原因であることが分かった．以上の結果から，RQ2 は以下の回答とする．

RQ2 への回答：

未知ライセンスを目視した結果，FreeBSD-10.3.0 では 69 種類，Linux-4.4.6 では 33 種類，Debian-7.8.0 では 194 種類のライセンスが発見された．ほとんどは既知のライセンスの未知の表記揺れであり，ルール（正規表現）を作成することで解決できると考えられる．

3.2　予備調査 2：コンテナのライセンス互換性分析

2.6.3 章では，Docker イメージは多くの OSS パッケージが統合されているため，ライセンス互換性の検証が必要となる点について述べた．しかしながら，現状，インターネット上に存在する Docker イメージはどの程度ライセンス互換性が保証されているかは不明である．本章では，以下の 2 つのリサーチクエスチョンについて取り組む．

- **RQ3: GPL と非互換なライセンスはパッケージにどの程度適用されているか**
- **RQ4: ライセンスの互換性問題はどれくらいの Docker イメージに発生するか**

3.2.1　調査方法

Docker コンテナに含まれるソフトウェアパッケージと OSS ライセンスを特定し，調査・分析するための手順の概要を図 3.2 に示す．本分析方法は OSS のコンテナ解析ツールである Tern と OSS ライセンス特定ツール Ninka [14] によるライセンス特定と，非互換ライセンスの対応付けにより構成されている．各処理の詳細については以下に示す．

Tern による OSS パッケージ分析

まず，コンテナ解析ツール Tern[3]を用いて Docker イメージに含まれるパッケージ名とそのライセンス情報を各レイヤーごとに抽出する．例えば，tomcat:10.0-jdk11-corretto イメージの場合の出力は図 3.3 のようになる．

Tern は Docker イメージをレイヤーごとに解析する．その概要を図 3.4 に示す．まず，Docker イメージ内に保持されている各レイヤのデータを取得する．まず，最低層のレイヤ 0 のデータのファイルシステムを展開し chroot でホストマシンにマウントする．そして，コンテナ内の OS（ベース OS）に対応するパッケージ管理システムを通してパッケージのメタ情報からライセンス名を取得する．レイヤ 0 でのパッケージ情報取得完了後，次のレイヤ 1 のデータをレイヤ 0 と結合し再度パッケージのメタ情報を取得する処理を繰り返していく．これを最も上位のレイヤまで繰り返した後，最終的に抽出できたパッケージをレポートとして出力する流れとなっている．

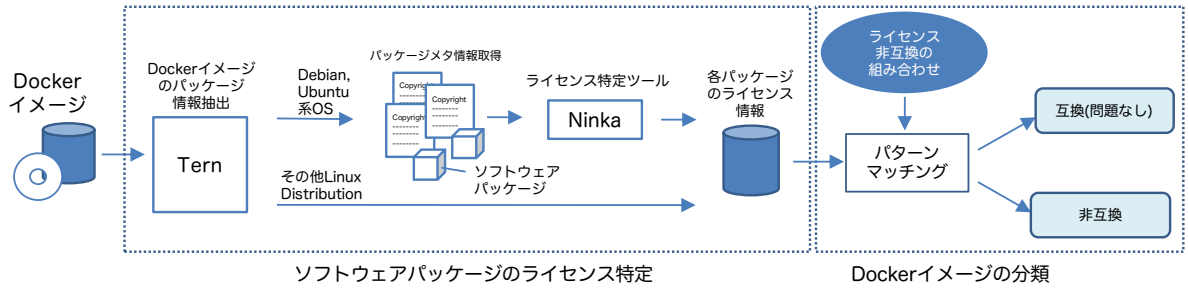


図 3.2: Docker イメージのライセンス互換性検証結果による分類

[3]https://github.com/tern-tools/tern

```
Docker image: tomcat:10.0-jdk11-corretto:
        Layer 1:
                info: Layer created by commands: /bin/sh -c #(nop) ADD file:a0918d3dfd5c1b4e0594c53d0dfc4c97f1ebaf14085279c79f36f06cf7ed95ee in /
                info: Found 'Amazon Linux 2' in /etc/os-release.
                info: Retrieved package metadata using rpm default method.

        File licenses found in Layer:  None
        Packages found in Layer:
        +-------------------------+-----------+-----------------------------------------------------------------------------------------+------------+
        | Package                 | Version   | License(s)                                                                              | Pkg Format |
        +-------------------------+-----------+-----------------------------------------------------------------------------------------+------------+
        | tzdata                  | 2022a     | Public Domain                                                                           | rpm        |
        | basesystem              | 10.0      | Public Domain                                                                           | rpm        |
        | glibc-common            | 2.26      | LGPLv2+ and LGPLv2+ with exceptions and GPLv2+                                          | rpm        |
        | pcre                    | 8.32      | BSD                                                                                     | rpm        |
        | info                    | 5.1       | GPLv3+                                                                                  | rpm        |
        | xz-libs                 | 5.2.2     | LGPLv2+                                                                                 | rpm        |
        | libcap                  | 2.54      | BSD or GPLv2                                                                            | rpm        |
        | libgpg-error            | 1.12      | LGPLv2+                                                                                 | rpm        |
        | libacl                  | 2.2.51    | LGPLv2+                                                                                 | rpm        |
        | libxml2                 | 2.9.1     | MIT                                                                                     | rpm        |
        | keyutils-libs           | 1.5.8     | GPLv2+ and LGPLv2+                                                                      | rpm        |
        | libuuid                 | 2.30.2    | BSD                                                                                     | rpm        |
        | libdb-utils             | 5.3.21    | BSD and LGPLv2 and Sleepycat                                                            | rpm        |
        | cpio                    | 2.11      | GPLv3+                                                                                  | rpm        |
        | findutils               | 4.5.11    | GPLv3+                                                                                  | rpm        |
        | libtasn1                | 4.10      | GPLv3+ and LGPLv2+                                                                      | rpm        |
        | openssl-libs            | 1.0.2k    | OpenSSL                                                                                 | rpm        |
        | libssh2                 | 1.4.3     | BSD                                                                                     | rpm        |
        | cyrus-sasl-lib          | 2.1.26    | BSD with advertising                                                                    | rpm        |
        | nss-tools               | 3.67.0    | MPLv2.0                                                                                 | rpm        |
        | rpm                     | 4.11.3    | GPLv2+                                                                                  | rpm        |
        | libblkid                | 2.30.2    | LGPLv2+                                                                                 | rpm        |
        | yum-metadata-parser     | 1.1.4     | GPLv2                                                                                   | rpm        |
        | python2-rpm             | 4.11.3    | GPLv2+                                                                                  | rpm        |
        | yum-plugin-priorities   | 1.1.31    | GPLv2+                                                                                  | rpm        |
        | vim-minimal             | 8.2.5172  | Vim and MIT                                                                             | rpm        |
        | ncurses-base            | 6.0       | MIT                                                                                     | rpm        |
        | filesystem              | 3.2       | Public Domain                                                                           | rpm        |
        | libgcc                  | 7.3.1     | GPLv3+ and GPLv3+ with exceptions and GPLv2+ with exceptions and LGPLv2+ and BSD        | rpm        |
        | libstdc++               | 7.3.1     | GPLv3+ and GPLv3+ with exceptions and GPLv2+ with exceptions and LGPLv2+ and BSD        | rpm        |
        | zlib                    | 1.2.7     | zlib and Boost                                                                          | rpm        |
        | libdb                   | 5.3.21    | BSD and LGPLv2 and Sleepycat                                                            | rpm        |
        | libcom_err              | 1.42.9    | MIT                                                                                     | rpm        |
        | elfutils-libelf         | 0.176     | GPLv2+ or LGPLv3+                                                                       | rpm        |
```

図 3.3: tomcat:10.0-jdk11-corretto イメージにおける Tern 出力の例

ただし，パッケージ管理システム APT のサポートが完全でないという Tern の仕様により、ライセンス情報の出力は、コンテナ内のベース OS 層で使用されている Linux ディストリビューションのパッケージ管理システムに依存する．APT は主に Debian や Ubuntu などの Linux ディストリビューションで使用されている．CentOS や Amazon Linux，Alpine Linux などの APT 以外を利用するベース OS では，その Docker イメージに含まれているパッケージとライセンス名を出力する．APT を持つ Docker イメージの場合は，パッケージソフトウェア内のライセンスファイル（COPYRIGHT ファイル等）の内容を出力する．COPYRIGHT ファイルにはライセンス記述や特定のフォーマットで記述されたソフトウェアパッケージのライセンス情報が記述されている．COPYRIGHT ファイルの内容からライセンス名を特定するため，2.6.1 章で説明したライセンス特定ツール Ninka [14] を利用する．COPYRIGHT ファイルの内容を Ninka に入力することでそのライセンス名を取得する．Ninka は，入力されたソースファイルのライセンス記述に対応するライセンス名を出力する．パッケージのメタ情報にライセンス記述が含まれている場合はライセンス名を特定することができる．

ライセンス互換性の検証

本研究で対象とするライセンスの互換性に問題がある OSS ライセンスの組みわせは，表 3.4 に示す既知の組み合わせ計 17 種類とする．これらは，FSF (Free Software Foundation) が提供する文献 [31] および，既存研究 [32] をもとに選定した．OSS ライセンスの互換性に関する既存の研究は，GPL を中心に議論しているものが多い [32] [22]．GPL は，その条項の中で，GPL にない制

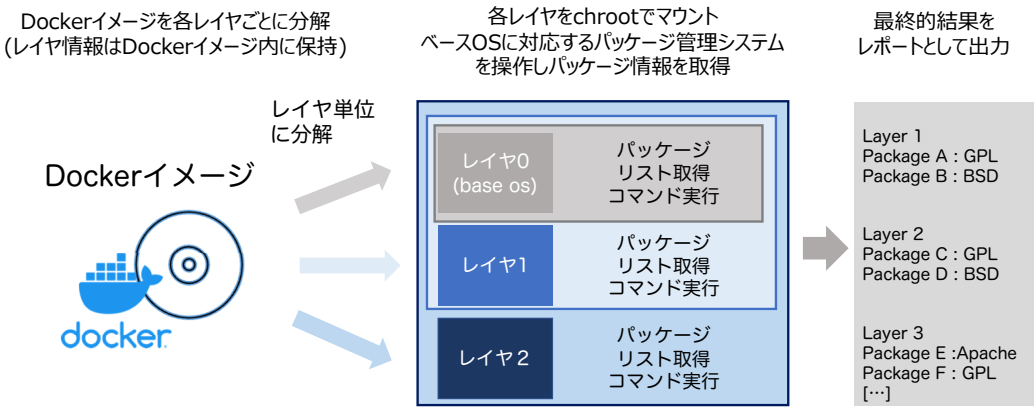


図 3.4: Tern のパッケージ解析の仕組み

表 3.4: 本研究で扱う GPL と非互換の OSS ライセンス

ライセンス	GPL v2	GPL v3	非互換の概要
Apache v1	✓	✓	GPL では課されていない無許可での Apache 関連の製品名使用禁止の制約が存在する[1]
Apache v2	✓		GPL v2 では課されていない特許の終了と保障に関する条項が存在する[2]
BSD4	✓	✓	GPL では課されていない宣伝条項が存在する[3]
CDDL v1.0	✓	✓	弱いコピーレフトのライセンス．ファイルに限定したコピーレフト条項を持つ[4]
EPL v1.0	✓	✓	弱いコピーレフトのライセンス．モジュールに限定したコピーレフト条項を持つ[5][6]
LPPL	✓	✓	弱いコピーレフトのライセンスであり，GPL では課されていない複数の制約を持つ[7][8]
MPL v1.1	✓	✓	弱いコピーレフトのライセンス．ファイルに限定したコピーレフト条項を持つ また，マルチライセンスを許容しており，二次著作物作成時のライセンスの選択が可能[9].
OpenSSL	✓	✓	再頒布の際に OpenSSl toolkit 使用の旨を記載する条項が含まれている[10]
GPL v3	✓		GPL v2 には無い複数の制約を持つ[11]．ただし，version 2 or later でリリースされている GPL v2 とは互換性がある[12]
LGPL v3	✓		GPL v2 にはない複数の制約を持つ [13]

✓ は互換性がない（非互換）であることを示す

約を課すことを禁止することを明記している．そのため，GPL では課されていない著作物や組織名の記載を課しているライセンスとは両立しない．

また，GPL 以外のコピーレフト条項を持つライセンスとも互換性がない．コピーレフトとは，一定の条件下で再利用した場合，その二次著作物のソースコードを公開しなければならないという概念である．GPL 以外のライセンスの多くは，GPL のようにソフトウェア全体のソースコードの公開を強制するものではなく，ファイル単位やモジュール単位などその範囲を明示的に限定している．本分析では，特に 1 つのソフトウェアパッケージにつき OSS ライセンスが 1 つ付与されているシングルライセンスのソフトウェアパッケージの組み合わせのみを対象として分析を行う．1 つのソフトウェアパッケージにつき，2 つ以上の OSS ライセンスが付与されている状態はマルチライセンスと呼ばれ，マルチライセンスは他のソフトウェアと互換性を持たせるために付与されている．また，Ninka によってあるソースファイルに対して複数の OSS ライセンスが特定されることがある．この場合，そのソースファイルを含むソフトウェアパッケージがどの OSS ライセンスを採用しているのかを正確に判断することができない．そのため，1 つのソフトウェアパッケージに対して 2 つ以上の OSS ライセンスが特定されたソフトウェアパッケージは分析の対象外としている．

表 3.5: ライセンス互換性分析に用いるデータセット

	Docker イメージ数
公式コンテナ	156
一般コンテナ	620
合計	776

データセット

Schermann ら [33] は，GitHub archive として公開されているデータから，Dockerfile が存在する 15,000 の Github プロジェクトを分析した結果を公開している．本研究では，Schermann らによって公開されているデータセットから，プロジェクトの重複を排除した上で，Dockerfile を取得し，分析対象となるデータセットを作成する．データセットの概要を表 3.5 に示す．本研究では，利用可能な 2,745 の Dockerfile の内，(1) Docker イメージにベースレイヤ，カスタムレイヤがそれぞれ存在する Dockerfile を分析対象とする．また，様々な理由で Dockerfile がビルドできない場合があるため，(2) ビルドが成功し Docker イメージを生成できた Dockerfile のみを分析対象とする．(1), (2) のフィルタリングを行った結果，最終的に合計 776 件の Dockerfile からビルドした Docker イメージを分析対象とする．776 件のうち，Docker Hub で配布されている公式 Docker イメージが 156 個存在した．その他の 620 個は一般に公開されている通常の Docker イメージであった．

3.2.2 RQ3: GPL と非互換なライセンスはパッケージにどの程度適用されているか

動機： ライセンス互換性問題を検証するためには，まず，Docker イメージに含まれているパッケージに適用されている OSS ライセンスを調査する必要がある．本研究で取り組むライセンス互換性検証結果の予測は，あくまでパッケージに GPL との互換性がないライセンスが多く採用されていることが前提となる．しかしながら，Docker イメージに統合されているソフトウェアパッケージに関する調査は存在しない．そのため，Docker イメージのソフトウェアパッケージにどのような OSS ライセンスが含まれているかを調査する．

アプローチ： 3.2.1 章で説明した，Tern と Ninka を利用し，776 の Dockerfile をビルドして作成された Docker イメージに含まれるパッケージのライセンスを集計する．

調査結果： 表 3.6 に GPL と互換性の無いライセンスのソフトウェアパッケージの数を示す．776 の Docker イメージから，全 2,167 のソフトウェアパッケージが見つかった．表 3.6 から，GPLv2+

[1] https://www.gnu.org/licenses/license-list.html#apache1.1
[2] https://www.gnu.org/licenses/license-list.html#apache2
[3] https://www.gnu.org/licenses/license-list.html#OriginalBSD
[4] https://www.gnu.org/licenses/license-list.html#CDDL
[5] https://www.gnu.org/licenses/license-list.html#EPL
[6] https://directory.fsf.org/wiki/License:EPL-1.0
[7] https://www.gnu.org/licenses/license-list.html#LPPL-1.2
[8] https://www.gnu.org/licenses/license-list.html#LPPL-1.3a
[9] https://www.gnu.org/licenses/license-list.html#MPL
[10] https://www.gnu.org/licenses/license-list.html#OpenSSL
[11] https://www.gnu.org/licenses/license-list.html#GNUGPLv3
[12] https://www.gnu.org/licenses/gpl-faq.html#v2v3Compatibility
[13] https://www.gnu.org/licenses/license-list.html#LGPLv3
[14] https://cloud.google.com/bigquery/public-data/github

表 3.6: ソフトウェアパッケージとライセンス

	パッケージ種類数	(%)
GPL 非互換のライセンス（計）	**621**	**28.7**
Apache v1	6	0.3
Apache v2	31	1.4
BSD4	20	0.9
CDDL v1.0	2	0.1
EPL	12	0.6
GPL v2	100	4.6
GPL v2+	208	9.6
GPL v3+	101	4.7
LGPL v3+	26	1.2
LPPL	105	4.8
MPL v1.1	6	0.3
OpenSSL	4	0.3
GPL 互換のライセンス（計）	**1,546**	**71.3**

“+” は “or later” オプションありのライセンスを示す.

が最も多く，208 種類のソフトウェアパッケージに採用されていることが分かった．次いで多いのは LPPL と GPLv3+であった．また，ソフトウェアパッケージ全種類のうち 71.3%は GPL と互換性のあるパッケージであることが分かった．以上の結果から RQ3 は以下の回答とする．

RQ3 への回答:

Docker イメージから見つかった全種類のソフトウェアパッケージのうち 71.3%は，GPL ライセンスと互換性があることが分かった．反対に 28.7%のパッケージは GPL ライセンスと互換性がないライセンスであることが判明した

3.2.3 RQ4: ライセンスの互換性問題はどれくらいの Docker イメージに発生するか

動機： ライセンス互換性に問題のある Docker イメージを継承しコンテナ開発を行うとライセンス違反となる可能性がある．Docker イメージに含まれるソフトウェアパッケージの OSS ライセンスの研究は行われていないため，インターネット上にある Docker イメージのライセンス互換性の状況を調査する．

アプローチ： 表 3.4 記載のライセンス互換性に問題のあるパターンが 776 の Docker イメージにどれくらい見つかったかを集計する．GPL と互換性のないパッケージが 1 つ以上見つかったイメージを互換性に問題のある Docker イメージとして分類する．

調査結果： 表 3.7 に RQ4 の結果を示す．表 3.7 から，156 のオフィシャルイメージのうち 119 のイメージに，620 の通常イメージのうち 338 のイメージにそれぞれ GPL と互換性のないライセンスのパッケージが含まれていた．全体では 776 のイメージのうち，457(58.9%) の Docker イメージにライセンス互換性に問題があったことが分かった．オフィシャルイメージのほうがレギュラーイメージの方がライセンス互換性に問題のある割合が高くなっていた．これはオフィシャルイメー

表 3.7: GPL 互換性問題が発生している Docker イメージ

	GPL 非互換	GPL 互換	合計
Docker 公式イメージ	119 (76.3%)	37 (23.7%)	156
一般イメージ	338 (54.5%)	282 (45.5%)	620
合計	457 (58.9%)	319 (41.1%)	776

表 3.8: GPL 非互換ライセンス別のパッケージ種類数

	GPL v2	GPL v2+	GPL v3+
Apache v1	81	81	79
Apache v2	126	NA	NA
BSD4	**321**	**397**	**387**
CDDL v1.0	0	2	0
EPL v1.0	24	24	23
LPPL	2	2	2
MPL v1.1	82	82	82
OpenSSL	9	12	7
GPL v3+	**354**	NA	NA
LGPL v3	65	NA	NA

ジの方がレギュラーイメージより多くのソフトウェアパッケージが含まれている傾向にあり，ライセンス互換性の問題が発生しやすい状況であったためと考えられる．表 3.8 に GPL との互換性に問題があったライセンスの内訳を示す．表 3.8 から GPLv2+と BSD4 の組み合わせによる非互換が多く発生していたことが分かった．以上の結果から RQ4 には以下のように回答する．

RQ4 への回答:

主に GPL と互換性に問題のあるソフトウェアパッケージが存在する Docker イメージは 58.9%存在していた．特に，Docker の公式イメージのうち 76.3%は GPL と問題のあるパッケージを含んでいた．OSS ライセンス互換性に関する問題のほとんどは GPLv2+と BSD4 のライセンスの組み合わせによるものであった．

3.3 OSS の法的リスク特定自動化のための技術的課題

本章では，予備調査の結果を受け，OSS の法的リスク特定自動化のためにクリアしなければならない技術的課題について議論する．予備調査 1 では，大規模 OSS に対して Ninka を用いてライセンス特定を実施した場合，多くの未知ライセンスが検出されることが分かった．そのため，ライセンスルールを生成する前に著名なライセンスを分類する処理（T1-1）が必要である．さらに，各未知ライセンスで検出されたライセンスには偏りがあり，FreeBSD-10.3.0 では BSD2，Linux-4.4.6 では GPLv2，Debian では Perl ライセンスなどが多く検出された．このままライセンスルールを生成すると同一のライセンスを特定する正規表現が大量に生成されることになる．そのため，同一

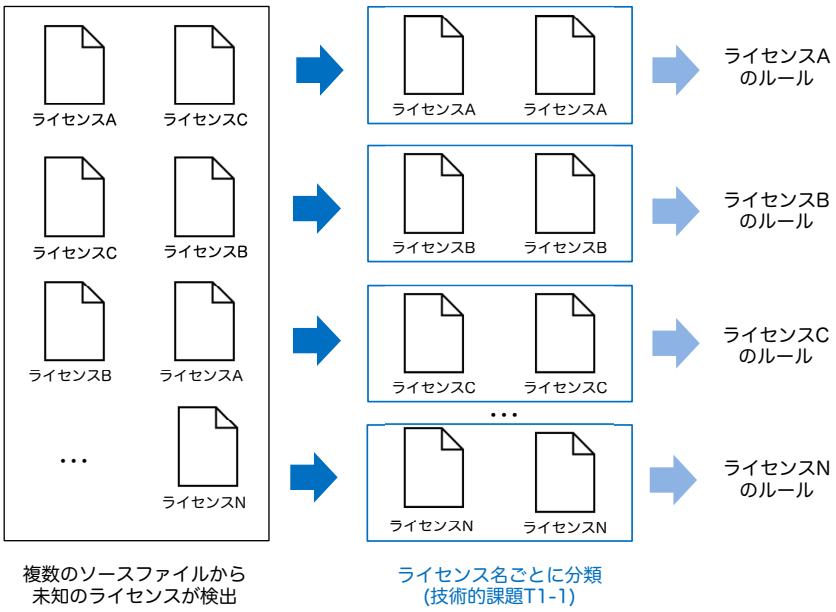


図 3.5: 技術的課題 T1-1:ライセンス名による未知ライセンスの分類

ライセンスのライセンスルールはできるだけ重複を排除しルール生成数を最小限に抑える（T1-2）ことが望ましい．予備調査 2 では，Docker Hub のイメージの 58.9%に，OSS パッケージのライセンス互換性に問題があることが確認できた．しかしながら，OSS パッケージの変更は動作環境の変更を意味するため，影響範囲が大きい．そのため，開発早期にパッケージ検証を行える仕組みが必要となる（T2）

以降，本研究で定義する技術的課題について 3.3.1 章，3.3.2 章，3.3.3 章にてその詳細を順に説明する．

3.3.1 T1-1：未知ライセンスが検出されたライセンス記述の分類

予備調査 1 の RQ1 では，各データセットから多くの未知ライセンスが検出されることが分かった．FreeBSD-10.3.0 では 1,821 ファイル，Linux-4.4.6 では 3,561 ファイル，Debian-7.8.0 では 2,838 ファイルに未知ライセンスが検出された．しかしながら，手作業によるライセンス名の確認は容易ではない．実際は未知ライセンスが何種類のライセンスから検出されたかは不明なため，各ファイルのライセンス記述を順番に確認していく必要がある．本研究では，未知ライセンスが検出されたライセンス記述の分類を技術的課題（T1-1）として定義する．技術的課題（T1-1）の概要を図 3.5 に示す．ライセンスルールを自動生成するには，ルール生成の元となるライセンス記述がライセンス名ごとに分類されている必要がある．これはライセンスルールが最終的に 1 つのライセンスのみにマッチしなければならないという要件を満たすためである．もし，ライセンス記述を分類せずにライセンスルールを生成した場合，複数のライセンスにマッチする不適切な正規表現が生成されてしまい，1 つのライセンス記述に対し，1 つのライセンス特定結果を出力することができなくなる．

検出された未知ライセンスを確認し分類することは容易ではない．まず，図 3.1 に示すライセン

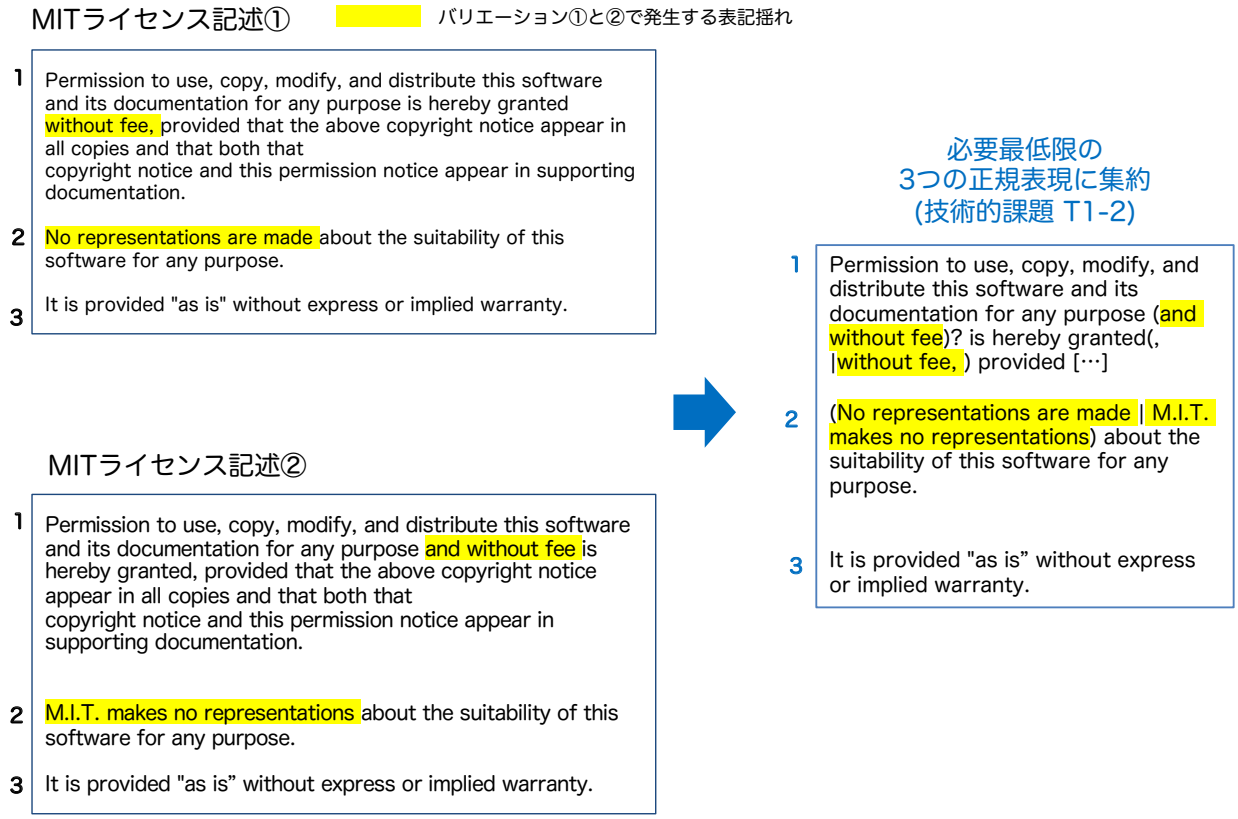


図 3.6: 技術的課題 T1-2:ライセンスルール生成数の最小化

ス名の記載がないライセンス記述が多く存在する．このようなライセンス記述は，基本的には，他のライセンス記述の文字列の差異の大きさやその内容から相対的に分類していく必要がある．また，ライセンスにはバージョンが存在する．ライセンス名が同一であってもバージョンが異なれば分類する必要がある．例えば，LGPLv2.1 と LGPLv3 のライセンス記述の文字列の差異は，記述者によるバリエーションを除けば，バージョン部の数字のみとなる．目視でこれらの違いを一つ一つ確認しながら分類するのは難しいと考えられる．

3.3.2 T1-2：生成する正規表現数の最小化

予備調査 1 の RQ2 では，未知ライセンス FreeBSD-10.3.0 では 69 種類，Linux-4.4.6 では 33 種類，Debian-7.8.0 では 194 種類のライセンスが含まれていることが分かった．さらに，図 3.1 に示すような BSD ライセンスに類似したライセンス記述も発見された．これらのライセンス記述から手作業で正規表現を作成するには，まず，ライセンス名ごとに表記揺れを調査する必要がある．もし，表記揺れを調査しない，つまり表記揺れを考慮した正規表現を作成しない場合，表記揺れごとに正規表現を分けて作成するしかない．しかしながら，表記揺れごとに正規表現を分けて作成すると，正規表現を多く作成することになり，後続の名付け作業に大きな労力がかかってしまう．正規表現の名付けにかかる労力を削減するためにも，表記揺れが考慮された必要最低限の数の正規表現を作成することは重要といえる．本研究では，必要最低限の正規表現の作成を技術的課題

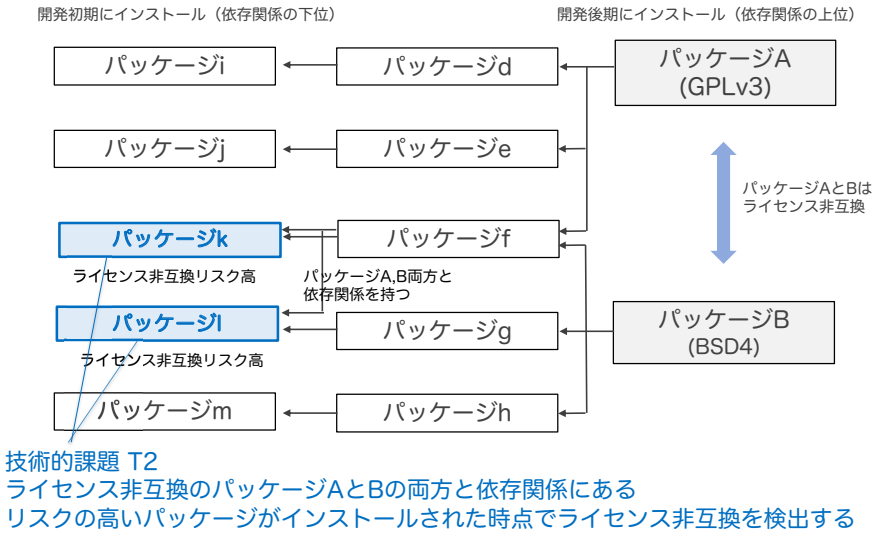


図 3.7: 技術的課題 T2:パッケージ依存関係を利用した開発早期でのライセンス非互換の検出

（T1-2）として定義する．

技術的課題（T1-2）の詳細を説明するため，図 3.6 に 3 つの文からなる MIT のライセンス記述の 2 種類のバリエーションを示す．2 つのライセンス記述間で文字列が異なる部分（表記揺れとなる部分）を黄色線で示す．この 2 つのライセンス記述から正規表現を手作業で作成する場合，1 つ目と 2 つ目のライセンス文の表記揺れを正規表現のメタ文字として反映することで計 3 つの正規表現にまとめることができる．しかしながら，表記揺れを調査するのは容易ではない．図 3.6 は 2 つのライセンス記述間の例であるが，実際は多くのライセンス記述を調査・比較し，ベースとなるライセンス記述のパターンとその表記揺れを区別する必要がある．この作業は膨大な文字数を目視で確認するため，手作業では非常に労力がかかる．ライセンスルールを自動生成する場合でも，計算量が大きくなると想定されるため，効率よく記述パターンを抽出する仕組みが必要と考えられる．

3.3.3 T2：開発早期におけるライセンス互換性検証結果の予測

3.2 章では，Docker イメージのライセンス互換性を検証したところ，全体の 58.9%で GPL と互換性のないパッケージが含まれていたことが分かった．

しかしながら，この互換性を解消するために，Docker イメージの OSS パッケージを変更するのは難しい．一般的な OSS パッケージには，暗黙的にも明示的にも複雑な依存関係があり，これらの関係は、特定のパッケージに対し，インストール・削除作業を行うとシステムが動作不能に至るリスクがあることが知られている [34]．

一般的な OSS パッケージの依存関係は方向を持つ．図 3.7 に示すような，依存関係の上位にあるパッケージ A と B には，下位の依存関係にあたるパッケージがあり，またそのパッケージがさらに下位の依存関係を持つパッケージが続くような構造となる場合が多い．そのため，必然的に下位にあるパッケージが最初にインストールされている必要がある．ライセンス互換性の問題が依存関係下位のパッケージに発生し，当該パッケージを変更する場合，システム動作確認のため，

依存関係上位の全てのパッケージについて変更要否を検討する必要がある．そのため，開発者は最後のライセンスの互換性検証後の多くの時間を消費していると想定される．

セキュリティやソフトウェアテストの分野では [35] シフトレフトという概念がある．従来のウォーターフォールモデルでは，開発工程の後半にテストなど品質保証活動（QA）を実施するため，致命的なバグや脆弱性が多く発見されると設計の修正後に再度プログラミングとテストをおこなう必要があることから手戻りコストが高くなり，リリーススケジュールに影響を及ぼす．近年では，アプリケーションをできるだけ迅速に提供するために，高頻度でリリースを行うアジャイル開発が注目されている．アジャイル開発では，開発プロダクトを頻繁にバージョンアップ，リリースするのが一般的である．その中で注目を浴びているシフトレフトは，品質保証活動を開発初期段階で頻繁に実施し，致命的なバグや脆弱性による手戻りコストを最小化することを意図した概念である．

ライセンスの互換性検証にもシフトレフトの概念を取り入れることが可能と考える．開発後期でのライセンスの互換性検証は，依存関係上位となるパッケージが増加するため，ライセンス互換性検証後のパッケージ変更要否検討コストが増加する構造はソフトウェアテストと同じである．本研究では，開発早期でライセンス互換性検証を行うことを技術的課題（T2）とする．開発早期でのライセンス互換性検証は容易ではない．開発早期では入力される OSS パッケージが少ないことが想定される．入力されるパッケージが少なければ少ないほど，ライセンス互換性の検証も十分に行うことができなくなる．そのため，図 3.7 にあるパッケージ k,l ようなライセンス互換性が発生するパッケージと依存関係となりやすいパッケージのインストールを検知するなどの方法を検討する必要がある．

第4章　ライセンスルール自動生成手法の開発

本章では，手作業によるライセンスルール作成を支援を目的とし，3.3 章で説明したライセンス名によるライセンス記述の分類（T1-1），ライセンスルール生成数の最小化（T1-2）の技術的課題をクリアするライセンスルール自動生成手法について説明する．

4.1　ライセンスルール自動生成手法

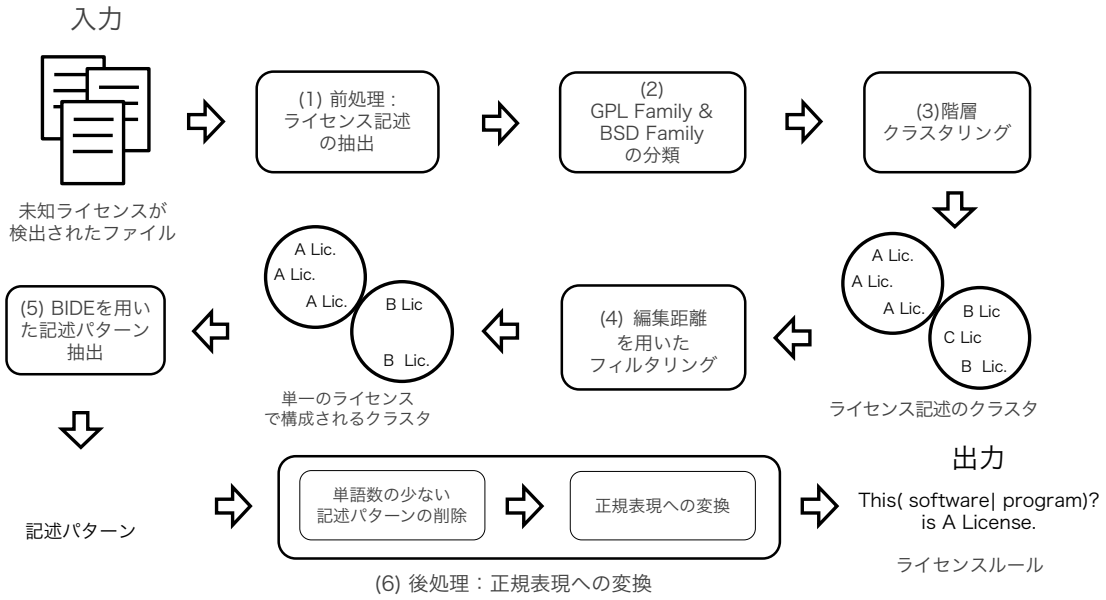


図 4.1: ライセンスルール自動抽出手法の概要

本章では，ライセンス特定ツールにより未知ライセンスが検出されたライセンス記述から，ライセンスルールを自動抽出する手法を提案する．

提案手法の概要を図 4.1 に示す．また，表 4.1 に各処理と技術的課題とのマッピングについて示す．提案手法は，大きく分けて 6 つの処理から構成される．まず前処理として，(1) 未知ライセンスと検出されたソースファイルからライセンス記述を抽出する．次に技術的課題（T1-1）に対応するため未知ライセンスの分類を行う．まず，(2) キーフレーズを用いて著名でかつ 1980 年代後期から普及しているライセンスである GPL 系（以下，GPL family）ライセンスと BSD 系（以下，BSD family）ライセンスを分類し，(3) 未知ライセンスのライセンス記述の集合に対して階層クラスタリングを適用し，類似するライセンス記述からなるクラスタを作成する．(4) にてテキスト間の類似度を用いて各クラスタに存在する外れ値となるライセンス記述をフィルタリングす

表 4.1: 各処理と技術的課題との対応

技術的課題	ライセンスルール自動生成手法	
-	処理 1	(前処理) ライセンス記述の抽出
T1-1 ライセンス名によるライセンス記述の分類	処理 2	GPL/BSD family ライセンスの分類
	処理 3	階層クラスタリング
	処理 4	編集距離によるフィルタリング
T1-2 生成する正規表現数の最小化	処理 5	BIDE を用いた記述パターン抽出
-	処理 6	(後処理) 正規表現への変換

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following **conditions** are met:

1. **Redistributions** of source code must retain the above **copyright** notice,
this list of **conditions** and the following **disclaimer**.
2. **Redistributions** in binary form must **reproduce** the above **copyright**
notice, this list of **conditions** and the following **disclaimer**
in the documentation and/or other materials provided
with the **distribution**.

図 4.2: ライセンス記述抽出の際に用いるキーワード（キーワードは太字で表示）

る．GPL/BSD family ライセンス以外にも，バージョン間でライセンス記述が酷似したライセンスは少なからず存在する．各クラスタに異なる未知ライセンス記述が混在しないようにする．

次に技術的課題（T1-2）に対応するため，記述パターンを抽出しライセンス記述の抽象化を行う．(5) 各クラスタに対し系列パターンマイニングアルゴリズムの 1 つである BIDE を適用し，ライセンス記述における語句の系列を記述パターンとして抽出する．

最後に後処理として，抽出した記述パターンを (6) 正規表現に変換する．記述パターンを抽出したライセンス記述からパターンの各要素間に該当する語句を探し，それらに対応できる正規表現に変換する．以降では，それぞれの処理の詳細を説明する．

4.1.1 処理 1 : ライセンス記述の抽出

ライセンス記述が存在するソースファイルのヘッダー部には，ソースコードの説明などのライセンスに無関係な文章も記述されている．本処理では，まず，ライセンスルール抽出に必要なライセンス記述を，ソースコードのヘッダー部から Ninka を用いて抽出する．2.6.1 章に記載したように，Ninka には，正規表現とのマッチングの前に，ライセンス記述がソースコードにあるかどうかを判定するために “copyright”，“redistribute”，“conditions” などのライセンス記述に頻出するキーワード（計 82 語）が含まれている文をライセンス記述として抽出する．図 4.2 に，例として BSD2 のライセンス記述に存在する頻出キーワードを示す．図 4.2 から，BSD2 のライセンス記述の全ての文に頻出するキーワードが含まれていることが分かる．

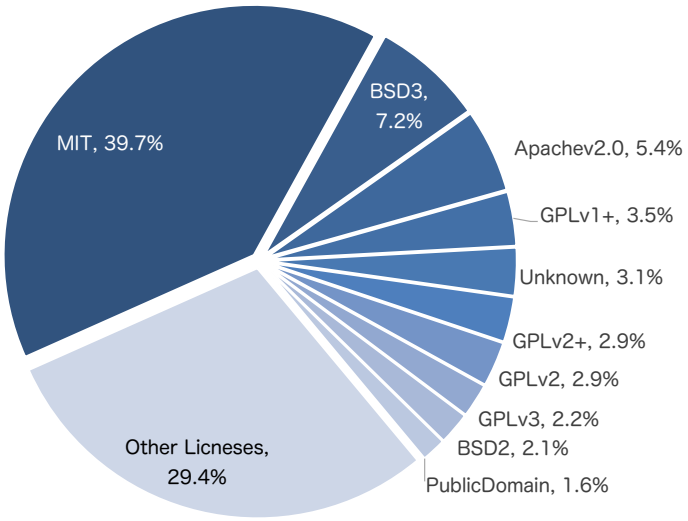


図 4.3: Zacchiroli ら [36] によるソース共有サイト内で使用されている OSS ライセンスの内訳（論文中図 3 を元に作成）

4.1.2 処理 2：GPL/BSD family ライセンスの分類

本処理では，キーフレーズを用いてライセンス記述が酷似している著名なライセンスを分類する．OSS ライセンスには，ライセンス記述が酷似しており，クラスタリングなど機械学習アルゴリズムでは区別することが難しいライセンスが存在する．Free Software Foundation によって定められた GPLv3，AGPLv3，などの GPL family ライセンスのライセンス記述は酷似している．図 4.4 に GPLv3，図 4.5 に AGPLv3 のライセンス記述のテンプレートを示す．この 2 種類のライセンスのライセンス記述は，ライセンス名のみが異なっており，ライセンス記述が酷似しているといえる．また，BSD2，BSD3 などの BSD family ライセンスでも同様にライセンス記述が酷似している．

本処理では，これらの GPL/BSD family ライセンスを，クラスタリングを適用すべきではないライセンス記述が酷似しているライセンスとして扱い，キーフレーズによる分類を行う．分類を正しく行うことができれば，未知ライセンスを削減することができるため，ライセンスルールを作成する必要がなくなる．

Zacchiroli [36] らは，Github と Gitlab のソース共有サイトで公開されている 1 億 5000 万のリポジトリで使用されている OSS ライセンスを調査し集計した．その結果を図 4.3 に示す．最も使用頻度が高いライセンスは MIT (39.7%) であり，それ以降 BSD3 (7.2%)，Apachev2.0 (5.4%)，GPLv1+(3.5%) と続くことが報告されている．これらの多く使用されているライセンスのうち，バージョンを持つライセンスには Apachev2.0 も含まれるが，複数バージョンにわたって幅広く利用されているライセンスは GPL family ライセンスと BSD family ライセンスのみであることが分かる．GPL/BSD family ライセンスは，共に 1980 年代末に作成されたライセンスであり，Linux や，FreeBSD などで広く使用されているライセンスである．そのため，本研究では，他のライセンスの中でも GPL family ライセンスと BSD family ライセンスについてキーフレーズを用いて分

This program is free software: you can redistribute it and/or modify
it under the terms of the **GNU General Public License** as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the **GNU General Public License** for more details.

You should have received a copy of the **GNU General Public License**
along with this program. If not, see <http://www.gnu.org/licenses/.>

図 4.4: GPLv3 のライセンス記述

This program is free software: you can redistribute it and/or modify
it under the terms of the **GNU Affero General Public License** as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the **GNU Affero General Public License** for more details.

You should have received a copy of the **GNU Affero General Public License**
along with this program. If not, see <http://www.gnu.org/licenses/.>

図 4.5: AGPLv3 のライセンス記述

表 4.2: GPL/BSD family ライセンス

ライセンスの系列	ライセンス
GPL	AGPLv3(+) GPLv1(+), GPLv2(+), GPLv3(+) LGPLv2.1(+), LGPLv3(+) LibraryGPLv2.0(+)
BSD	BSD2, BSD3, BSD4

表 4.3: GPL family ライセンスの検出条件

GPL family ライセンス	キーフレーズ	バージョン
GPLv1	GNU General Public License	version 1
GPLv2	GNU General Public License	version 2
GPLv3	GNU General Public License	version 3
LibraryGPLv2	GNU Library General Public License	version 2
LGPLv2.1	GNU Lesser General Public License	version 2.1
LGPLv3	GNU Lesser General Public License	version 3
AGPLv3	GNU Affero General Public License	version 3

類する．

表 4.2 に，GPL/BSD family ライセンスに属する 17 種類のライセンスを示す．GPL family ライセンスには，or Later 条項が存在する．or Later 条項とは，指定されているバージョンより新しいバージョンの GPL family ライセンスでの再配布を許可するかどうかを選択できる条項である．本稿では，or Later 条項が，ライセンス記述に存在する場合を接尾辞 “+” で表す．また，表 4.2 中の (+) は，or Later 条項が存在する場合と存在しない場合の両方の意味で記載している．表 4.2 中にある GPL family ライセンスには，“or Later” 条項の有無による違いを含めると，14 種類のライセンスが属している．GPL family ライセンスの検出には，表 4.3 に示しているキーフレーズと，バージョン名の組み合わせを用いる．また，or Later 条項の有無は，“or Later” をキーフレーズとして用いる．

BSD family ライセンスにおけるバージョンは，数値での表記ではなく，記述されている条項によって決定される．そのため，ライセンス記述に存在する条項を検出することによってそのバージョンを決定する．BSD family ライセンスの検出は，図 4.6 に示している条件を用いる．

4.1.3 処理 3：ライセンス記述の階層クラスタリング

本処理では，ライセンス記述をライセンス名によって分類するため，ライセンス記述に対して階層クラスタリングを適用し，類似するライセンス記述からなるクラスタを作成する．このとき，単一のライセンスのライセンス記述で構成されているクラスタを可能な限り多く作成することを最も重要と考える．異なるライセンスのライセンス記述で構成されているクラスタからライセンスルールを作成すると，複数のライセンスに適合するライセンスルールを抽出してしまい，ライセンス記述ごとのライセンス特定を実現できなくなるためである．

階層クラスタリングを行うには，まず，入力となるライセンス記述間の類似度を全組み合わせについて求める必要がある．ライセンス記述間の類似度算出するために，各ライセンス記述の Bag of Words ベクトルを作成する．Bag of Words ベクトルは，全文書を現れた全単語の頻出回数でベクトル表現したものであり，総単語数を N，総文書数を M とすると $N \times M$ の行列となる．その

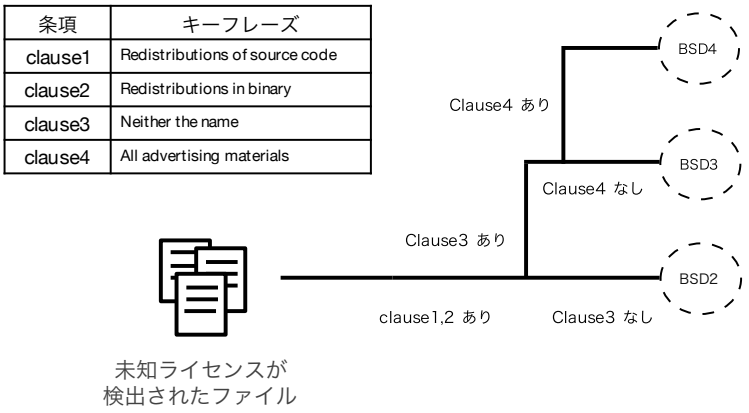


図 4.6: BSD family ライセンスの検出条件

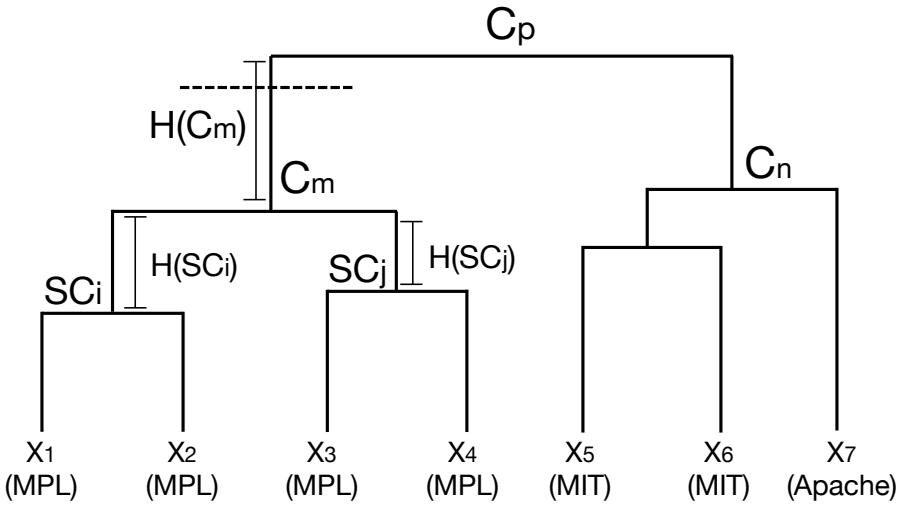


図 4.7: 樹形図の例

$[N_i, M_j]$ の要素は，単語 N_i が文書 M_i に現れる回数に対応する．このとき N_i の単語には，英語のどのような文書にも出てくる単語は特徴量がないため採用しない．例えば，“i”，“me”，“my” などの人称代名詞や “and”，“but” などの接続詞などをストップワードの設定を行う．ストップワードには，R のテキストマイニングパッケージ[1]で利用できる $stopwords(kind = "en")$ で表示される単語 174 語を設定する．

次に，ライセンス記述の Bag of Words ベクトルに対して階層クラスタリングを行い，その過程から図 4.7 のような樹形図を得る．樹形図とは，複数のデータを類似している順にクラスタとしてマージし，最終的に全てのデータが 1 つにマージされる様子を表したものである．樹形図の高さは，クラスタ間の距離を示しており，データとデータの類似関係，または，クラスタとクラスタの類似関係を読み取ることができる．階層クラスタリングでは，類似している順にクラスタを結合するためのクラスタ間の距離の定義が必要となる．このときクラスタ間の距離はウォード法 [37] を用いる．ウォード法は統合前のクラスタ内の個々のデータの分散と統合後の個々のデータの分

[1]http://cran.r-project.org/web/packages/tm/tm.pdf

散の差をクラスタ間の距離とする．クラスタ間の距離の考え方は，他にも，最も近いデータ間の距離を 2 つのクラスタ間の距離として代用する最短距離法（単リンク法）や最も遠いデータ間の距離を採用する最遠距離法（完全リンク法）などがある．最短距離法や最遠距離法では，最も近いデータ間の組み合わせとして，クラスタ内の重心から最も外れたデータが採用されやすいというデメリットがあり，外れ値に弱い特性を持つ．クラスタリングの対象となるライセンス記述は様々な種類のライセンスがあり，それぞれのライセンス記述の数や文字数も異なるため，外れ値が含まれる可能性がある．ウォード法は，ライセンス記述の偏りの影響が平均化され，外れ値の影響を受けにくい性質を持つ．また，ウォード法は空間拡散と呼ばれる性質を持ち，クラスタのマージが進むにつれて，クラスタが他のクラスタをマージしにくくなる性質を持つ．つまり，枝刈り時にデータ数の少ないクラスタを多く生成するような性質を持つ．各クラスタから正規表現を生成するため，生成されるクラスタが多いと，多くの正規表現が生成され，名前づけにかかる手間が増加してしまう．しかしながら，適切な正規表現を生成するという観点では，異なるライセンスがマージされたクラスタの生成を避けるため，データ数が少ないクラスタを多く生成する方が合理的であると考えられる．以上から本処理ではウォード法を採用している．

次に，樹形図から類似したライセンス記述からなるクラスタを得る．一般的な階層クラスタリングでは，閾値となるクラスタ間の距離（高さ）を樹形図から決定し，複数のクラスタに分割する．ただし，未知ライセンスと判定されたライセンス記述の中には，ライセンス名は異なるが酷似しているライセンス記述も含まれる．図 4.7 では，MPL が含まれている C_m と MIT と Apache が含まれている C_n がマージされ C_p が作成されている．この場合，MPL と Apache のライセンス記述は，異なるライセンスであるため C_p を作成すべきではない．このようなライセンス記述を別のクラスタにするため，ライセンス記述の階層クラスタリングでは，式 4.1 を満たす樹形図内のクラスタ C_m をクラスタとして切り出す．ここで，C_p は，C_m と次にマージされる対象のクラスタ C_n とする．また，C_m を分割して得られる最も小さいクラスタを SC_i と SC_j，$H(C_p)$ をクラスタ C_m と C_p 間の距離とする．

$$H(C_m) > H(SC_i) \ \ and \ \ H(C_m) > H(SC_j) \tag{4.1}$$

4.1.4 処理 4：ライセンス記述のフィルタリング

本処理では，クラスタの中で外れ値となるライセンス記述をフィルタリングする．GPL/BSD family ライセンス以外にも，バージョン間でライセンス記述が酷似したライセンスは少なからず存在する．各クラスタに異なる未知ライセンス記述が混在しないようにする．また，処理 3 でのクラスタリングも完全ではないため，図 4.8 に示す本フィルタリング処理にて吸収する．

外れ値とみなせるライセンス記述を検出するにあたって，まず，各クラスタにおいてクラスタの中心となるライセンス記述 X_d を決定する．クラスタの中心となるライセンス記述 X_d とは，ある閾値以上の類似度となるような他のライセンス記述 X_n と最も多くのペアとなり得るライセンス記述を指す．そのため本処理では，各クラスタに属する全てのライセンス記述 X_n のペアの類似度を求める．類似度には編集距離 [38] を用いる．編集距離とは，文字列 A を文字列 B と同値にするのに必要な文字列の編集（削除，挿入，置換）回数で，2 つの文字列の距離を定義するものである．ただし，そのまま用いるとライセンス記述の長さにより編集距離に偏りが生じる．そのため，本処理では，以下の式 4.2 で定義される 2 つのライセンス記述 (A,B) 間の編集距離を 2 つのライ

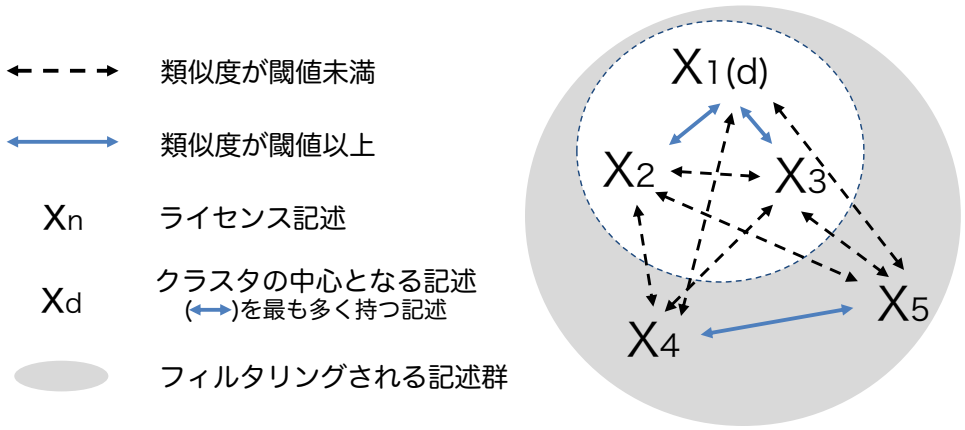


図 4.8: 編集距離によるクラスタの中心となるライセンス記述の選出

センス記述のうち長い方の文字数で割ることで正規化したものを用いる．図 4.8 では，X_1 が類似度が閾値以上のライセンス記述を 2 つ持っており，X_1-X_5 の中では最も多いため，X_1 がクラスタの中心となる記述として選出される．

$$Normalized\ Levenshtein\ Distance(A,B) = 1 - \frac{Levenshtein\ Distance(A,B)}{max\ length(A,B)} \tag{4.2}$$

クラスタの中心となるライセンス記述を決定した後，その記述との類似度が閾値を満たしているものを同一のライセンス記述のクラスタとして取り出す．図 4.8 では，X_1 と，X_1 が類似度が閾値以上である X_2，X_3 の計 3 つのライセンス記述が取り出される．同一のライセンス記述のクラスタとして取り出した後，クラスタの中心となるライセンス記述がまだ存在した場合，もう一度同じ処理を繰り返す．クラスタの中心となるライセンス記述がクラスタに複数存在した場合，1 つのクラスタから複数のクラスタを取り出すことになるため，結果としてクラスタが分割されることもある．最終的に類似度が閾値を満たしているライセンス記述がないライセンス記述はライセンスルール生成の対象外として除外される．

4.1.5 処理 5 : BIDE による系列パターンマイニング

表 4.4: 文章における語句の系列

	文章
1	This program is free
2	This library is free
3	This library is OSS

表 4.5: BIDE により抽出された飽和系列パターン

	パターン	支持度
1	(This),(is)	3
2	(This),(is),(free)	2
3	(This),(library),(is)	2

本処理では，ライセンス毎に分類されたクラスタに頻出する語句の出現順序（系列パターン）を抽出するため，系列パターンマイニングを行う．系列パターンマイニングは，複数の系列データに一定回数以上出現する系列を全て列挙するアルゴリズムである．系列パターンマイニングのアルゴリズムは，SPADE [39]，PrefixSpan [40] などいくつか提案されているが，その多くは計算コストを問うアルゴリズムであり，出力されるものは基本的には同一である．ただし，本処理では，

高速に飽和系列パターンを抽出する系列パターンマイニングアルゴリズム BIDE [41] を用いて行う．飽和系列パターンとは，同じ支持度（文章中に出現する回数）の系列パターンの中で他の系列パターンの部分集合にならない系列パターンを指す．

例えば，表 4.4 では，“This program is free” と “This library is free” という文章には，

“(This),(is),(free)” と “(is),(free)” という 2 つの系列パターンが共通して確認できる．ただし，“(is),(free)” は，“(This),(is),(free)” の部分集合の関係にあるため飽和系列パターンではない．BIDE は飽和系列パターン抽出するが，異なる支持度の飽和系列パターン間においては他の系列パターンの部分集合となることを許容するため，ライセンスルールの作成に特化しない系列パターンを抽出してしまうことがある．例えば，表 4.5 において，“(This),(is)” は，“(This),(is),(free)” の部分集合である．しかし，“(This),(is)” の系列パターンは，一般的な英文にも対応してしまうためライセンス特定のためのライセンスルール作成に用いるのは不適切である．本研究では，ライセンス記述以外の記述に対応する系列パターンの出力を防ぐために，各クラスタで抽出された系列パターンに対して，他の系列パターンの部分集合となる系列パターンをマージする．また，本研究では，著作権を示す文（“Copyright (c) [yyyy] *** inc. All rights reserved.” 等）をライセンス記述ではないとした．そのため，著作権を示す文を削除後，各クラスタに対し BIDE を適用する．

4.1.6 処理 6 : 記述パターンから正規表現への変換

単語数の少ないライセンス記述のフィルタリング

本処理では，正規表現に変換すべきではない，単語数の少ない記述パターンのフィルタリングを行う．前ステップにおいて，BIDE よる系列パターンマイニングにおいて出力される他の部分集合となる記述パターンを削除しているが，本処理では，実際に記述されているライセンス記述から生成される単語数の少ない記述パターンのフィルタリングを行う．

筆者が予備調査を行ったところ，ライセンス記述に，“GPL” や “License” などといった 1 単語のみの記述や数単語程度で記述された短い文が多数見つかった．単語数の少ない記述パターンをこれらの文が多数存在するクラスタに対し BIDE を適用すると，要素が “(GPL)”，“(License)” だけの記述パターンが出力されてしまう．これらのパターンは，1 種類のライセンス記述にのみ適合するものではないので，ライセンスルールに変換すべきではない．そこで，本処理では，要素数が閾値以下の記述パターンをフィルタリングする．

正規表現への変換

本処理では，記述パターンをライセンスルールとして利用できるようにするため，前の処理で抽出された記述パターンを正規表現に変換する．

記述パターンの各要素は連続してるとは限らないため，要素間に該当する語句を，記述パターンを抽出した文から探索する．要素間に該当する語句が存在しなかった場合，つまり，要素間が連続している場合はスペースを要素間に挿入する．もし，要素間に該当する語句が存在した場合，その語句を表記揺れとして抽出し，それらに対応できるようメタ文字を利用した正規表現を作成する．

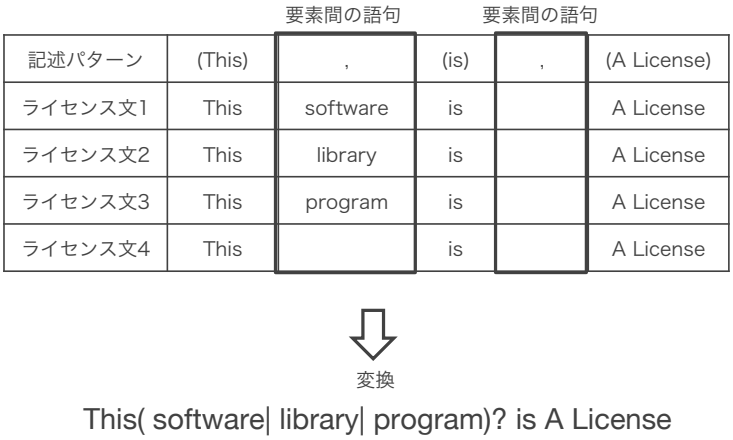


図 4.9: メタ文字を使用した正規表現への変換

メタ文字を利用した正規表現への変換を図 4.9 に示す．要素間に該当する語句が存在する文が複数存在し，かつ，その語句が異なっていた場合，正規表現のメタ文字 “ | ” を利用し，直前の括弧内のいずれかの語句であれば適合する正規表現に変換する．正規表現のメタ文字 “ | ” は，直前の文字と直後の文字のどちらかを表す．また，要素間に該当する語句が存在する文と，存在しない文が共に存在した場合，メタ文字 “?” を利用した正規表現に変換する．正規表現のメタ文字 “?” は，直前の文字の 0 または 1 回の繰り返しを表す．

4.2 ケーススタディ

本章では，提案手法の有用性を評価するために行ったケーススタディとその結果について説明する．本ケーススタディでは，計 5 つのリサーチクエスチョンについて取り組む．これらの RQ は，各技術的課題をどれくらい達成しているかを観点に設定している．各 RQ と技術的課題との対応を表 4.6 に示す．

表 4.6: 技術的課題（T1-1，T1-2）とリサーチクエスチョンとの対応

技術的課題		リサーチクエスチョン	
T1-1	ライセンス名ごとに ライセンス記述を分類できるか	RQ5	GPL/BSD family ライセンスを何%の精度で分類できるか
		RQ6	作成したクラスタのうち単一のライセンスからなる クラスタは全体の何%か
		RQ7	編集距離を用いて単一のライセンスからなるクラスタのみに フィルタリングすることができるか
T1-2	ライセンスルール生成数の最小化	RQ8	正規表現をどれくらい生成するか
		RQ9	生成した正規表現はどれくらいの未知ライセンス文 とマッチするか

表 4.7: データセットの概要

プロジェクト	未知ライセンス数[1]	ライセンス種類数[2]	抽出方法
FreeBSD-10.3.0	1,821	69	FreeBSD-10.3.0 のソースファイル[3] のライセンスを Ninka で特定，未知ライセンスが検出されたファイルを抽出
Linux-4.4.6	3,561	33	Linux-4.4.6 のソースファイル[4] のライセンスを Ninka で特定，未知ライセンスが検出されたファイルを抽出
Debian-7.8.0	2,838	194	Debian-7.8.0 が管理している各 OSS パッケージ[5] から 1 ファイルずつサンプリングし，そのライセンスを Ninka で 特定，未知ライセンスが検出されたファイルを抽出

4.2.1 データセット

データセットの概要を表 4.7 に示す．本ケーススタディでは，3.1 章での予備調査 1(未知ライセンスの発見的調査) で作成された未知ライセンスのデータセット，FreeBSD-10.3.0，Linux-4.4.6 と，Debian-7.8.0 のソフトウェアパッケージから検出された未知ライセンスを計 3 つのデータセットとして用いる．これらの未知ライセンスは筆者の目視によりライセンス特定済みである．本ケーススタディでは，3 つの OSS プロジェクトから検出された未知ライセンスに対し，提案手法を適用し，ライセンスルールを生成する．

4.2.2 閾値

編集距離を用いたフィルタリング時の類似度

各クラスタに含まれている外れ値とみなせるライセンス記述をフィルタリングを編集距離を用いてフィルタリングする．外れ値とみなせるライセンス記述の除去において，クラスタの中心となるライセンス記述を判定するための類似度の閾値を定めている．本ケーススタディで用いる閾値を決定するために，予備調査を行った．Debian-7.8.0 のソフトウェアパッケージからライセンスの種類が異なるライセンス記述のペア 1,000 組の類似度を予備調査した．1000 組のライセンス記述のペアは，Debian-7.8.0 のソフトウェアパッケージのソースファイルのライセンスを Ninka で特定し，特定に成功したソースファイルからランダムサンプリングしたものである．

[1]データセットとして用いる未知ライセンスが検出されたファイル数
[2]未知ライセンスのライセンス名を目視で特定した結果，各データセットに含まれていたライセンスの種類数
[3]https://github.com/freebsd/freebsd/tree/release/10.3.0
[4]https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.6.tar.xz
[5]http://ftp.riken.jp/pub/Linux/debian/debian-cd/7.8.0/source/iso-dvd/

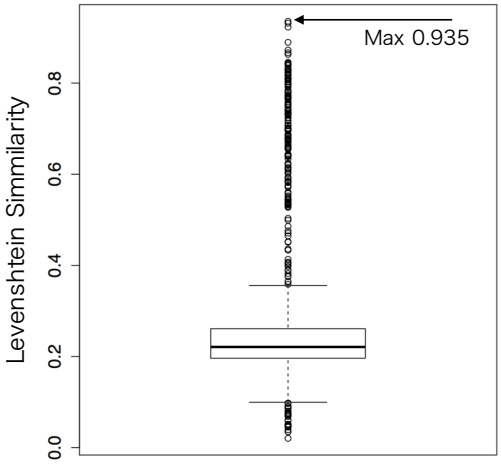


図 4.10: 1000 組の異なるライセンスのライセンス記述の類似度

その結果を図 4.10 に示す．縦軸は，編集距離によって定義される類似度を表している．多くのライセンス記述のペアの類似度は 20%–30%の範囲内に収まっているが，93.5%と非常に高い類似度を示すライセンス記述ペアが存在した．これは，LGPLv3 と LGPLv2.1 のライセンス記述のペアであり，2 つのライセンス記述で異なっている箇所は，バージョンと著作者名のみであった．このため，ライセンス記述が酷似していても区別できるように，本ケーススタディでは閾値を予備実験における類似度の最大値を考慮した 94%に設定する．

BIDE 適用時の支持度

単一のライセンスからなるライセンス記述の各クラスタに対して，BIDE を用いて，各クラスタのライセンス記述に頻出する語句の出現順序を記述パターンとして抽出する．BIDE を含む系列パターンマイニングアルゴリズムは，与えられた支持度（出現回数）以上の系列パターンを出力する．支持度を高く設定すれば，ライセンス記述に頻出する記述パターンのみを高速に抽出することができるが，与えた支持度以下の記述パターンを抽出できなくなる．一方，支持度を低く設定すれば，計算コストは高くなってしまうが，記述パターンをより多く抽出することができる．

本ケーススタディでは，支持度を 2 に設定した．これは，抽出した記述パターンが，少数のライセンス記述にしか存在しない表記揺れに対応できるようにするためである．

ライセンスルールに変換する記述パターンの要素数

要素数が少ない記述パターンをライセンスルールに変換しないようにするため，要素数が閾値以下の記述パターンを取り除いた後，ライセンスルールに変換する．本ケーススタディで用いる要素数の閾値を決定するため，Ninka のライセンスルールの単語数の予備調査を行った．GPL family，BSD family ライセンスのライセンスルールは，多数の内部ルールが存在したため，調査対象から除外した．その結果，最も単語数が少ないライセンスルールは，“This program is free software” であった．このとき，単に要素数の閾値を単語数とするのではなく，英文に頻出する英単語（“I”,

“my”, “of” など文法的ワードを含む）174 語の stopword を除いた単語数とする．“This”, “is” が stopword に該当するため，記述パターンの要素数は 3 と設定した．stopwords となる単語数を要素としてカウントすると，“(This),(is),(of),(the),(the)” などのどのライセンスにも見られるようなパターンからライセンスルールを抽出してしまう可能性がある．

4.3 実験結果

4.3.1 RQ5: GPL/BSD family ライセンスを何%の精度で分類できるか

動機： 4.1.2 章で述べたように，GPL/BSD family ライセンスは，バージョン間でライセンス記述が酷似している．GPL/BSD family ライセンスはクラスタリングによる分類が難しいと考えられるため，キーフレーズを用いて事前に除外するためのフィルター処理を設定した．そのため，本 RQ にてキーフレーズによる分類精度について調査する．

アプローチ： 4.1.2 章で述べたキーフレーズを用いて GPL/BSD family ライセンスの分類を行い，適合率，再現率，F 値を算出する．適合率（式 4.3）は，GPL/BSD family ライセンスであると判定したライセンス記述のうち，GPL/BSD family ライセンスであったライセンス記述の割合である．再現率（式 4.4）は，GPL/BSD family ライセンスであるライセンス記述のうち GPL/BSD family ライセンスであると判定した割合である．F 値（式 4.5）は，本来トレードオフの関係にある適合率と再現率の調和平均をその分類器の性能とする．適合率が高いほど，GPL/BSD family ライセンスのみを分類できていることを表しており．また再現率が高いほど，取りこぼしが少ないことを表している．本処理では，適合率を重要視する．適合率が低い場合，分類された GPL/BSD family ライセンスを目視で特定し，適切なライセンスルール作成する必要がある．一方，適合率が高い場合，ライセンスルールを作成しなくてよいため，ライセンスルール作成にかかるコストを大きく削減することができる．また，デュアルライセンスなど，1 つのライセンス記述に複数の GPL/BSD family ライセンスが検出されたライセンス（multiple）は，GPL/BSD family ライセンスとは区別して精度を評価する．本処理では，multiple を判定する仕組みを持たないため，multiple のライセンスを GPL/BSD family ライセンスとして判定した場合，間違い（False-Positive）として扱う．

$$\text{適合率} = \frac{\text{分類に成功したライセンスの数 } (TP)}{\text{分類に成功したライセンスの数 } (TP) + \text{分類に失敗したライセンスの数 } (FP)} \tag{4.3}$$

$$\text{再現率} = \frac{\text{分類に成功したライセンスの数 } (TP)}{\text{分類に成功したライセンスの数 } (TP) + \text{本来分類すべきであったライセンスの数 } (FN)} \tag{4.4}$$

$$F \text{ 値} = \frac{2 * \text{適合率} * \text{再現率}}{\text{適合率} + \text{再現率}} \tag{4.5}$$

結果: 表 4.8, 4.9, 4.10 に FreeBSD-10.3.0，Linux-4.4.6, Debian-7.8.0 における分類の評価結果をそれぞれ示す．“multiple” は，1 つのライセンス記述に複数の GPL/BSD family ライセンスが検出されたものを示している．また “全体” 欄は，GPL/BSD family ライセンスとそれ以外で評価した時の評価結果を示している．

FreeBSD-10.3.0 では，FreeBSD-10.3.0 の主なライセンスである BSD2 のライセンスが多く分類されている．一方，BSD4 の再現率が低く，BSD4 のライセンスを多くを取りこぼしていることが

表 4.8: FreeBSD-10.3.0 における GPL/BSD family ライセンス分類の評価

ライセンス	TP	FP	FN	適合率	再現率	F 値
AGPLv3+	-	-	-	-	-	-
AGPLv3	-	-	-	-	-	-
GPLv1+	2	0	0	1	1	1
GPLv1	-	-	-	-	-	-
GPLv2+	17	0	8	1	0.68	0.81
GPLv2	-	-	-	-	-	-
GPLv3+	-	-	-	-	-	-
GPLv3	-	-	-	-	-	-
LGPLv2.1+	2	0	0	1	1	1
LGPLv2.1	1	0	0	1	1	1
LGPLv3+	29	0	0	1	1	1
LGPLv3	-	-	-	-	-	-
LibraryGPLv2+	-	-	-	-	-	-
LibraryGPLv2	-	-	-	-	-	-
BSD4	11	0	163	1	0.06	0.11
BSD3	124	0	44	1	0.74	0.85
BSD2	384	20	33	0.95	0.92	0.93
multiple	0	1	0	NaN	NaN	-
全体	570	21	224	**0.96**	**0.72**	0.82

分かる．一方，各ライセンスの適合率は高く，全てが 0.95 以上である．FreeBSD-10.3.0 では，また，全体での再現率は 0.72，適合率は 96%であり，高い精度で分類できていることが分かった．

Linux-4.4.6 では，Linux-4.4.6 の主なライセンスである GPLv2 が，最も多く分類されている．一方で，GPLv2 の再現率は低く最も多く取りこぼしているのも GPLv2 であることが分かる．また，各ライセンスの適合率は，GPLv2(+) のライセンスは 0.89 以上である．一方で，BSD2 の適合率は 0.36 と低く，14 個検出した BSD3 のうち，実際に BSD3 であったのは 5 つしかない．また，全体では，再現率は 0.58 であったが，適合率は 0.92 であり，高い精度で分類できていると言える．

Debian-7.8.0 から作成したデータセットでは，最も多く分類されたのは GPLv2+であり，最も多く分類されなかったのは GPLv2 である．また，ほとんどの GPL family ライセンスが高い適合率で分類できていたものの，AGPLv3 と LibraryGPLv2+の適合率は 0.75，0.64 であった．BSD family ライセンスでは，BSD4 と BSD3 は適合率 1.00，0.94 の精度で分類できた．ただし，BSD2 の適合率は 0.39 と低く，44 個検出した BSD2 のうち，実際に BSD2 であったのは 17 個しかない．また，全体での再現率は 0.62，適合率は 0.86 であり，高い精度で分類できていると言える．以上から，RQ5 の結果は以下のように回答する．

RQ5 への回答

ライセンス記述が酷似しているライセンスを，FreeBSD-10.3.0 では 96%，Linux-4.4.6 では 92%，Debian-7.8.0 では 86%の適合率で分類できた．

表 4.9: Linux-4.4.6 における GPL/BSD family ライセンス分類の評価

ライセンス	TP	FP	FN	適合率	再現率	F 値
AGPLv3+	-	-	-	-	-	-
AGPLv3	-	-	-	-	-	-
GPLv1+	-	-	-	-	-	-
GPLv1	0	1	0	NaN	NaN	NaN
GPLv2+	79	10	42	0.89	0.65	0.75
GPLv2	613	14	530	0.98	0.54	0.7
GPLv3+	-	-	-	-	-	-
GPLv3	0	2	0	NaN	NaN	NaN
LGPLv2.1+	-	-	-	-	-	-
LGPLv2.1	-	-	-	-	-	-
LGPLv3+	-	-	-	-	-	-
LGPLv3	-	-	-	-	-	-
LibraryGPLv2+	-	-	-	-	-	-
LibraryGPLv2	-	-	-	-	-	-
BSD4	-	-	-	-	-	-
BSD3	2	0	39	1.00	0.05	0.10
BSD2	5	9	0	0.36	1.00	0.53
multiple	0	25	0	NaN	NaN	-
全体	699	61	511	**0.92**	**0.58**	0.71

4.3.2 RQ6: 単一のライセンスからなるクラスタは全体の何%か

動機： 4.1.3 章で述べたように，ライセンスルールは，単一のライセンスからなるクラスタから作成されるべきである．複数の異なるライセンスからなるクラスタからは，複数の異なるライセンスを特定するライセンスルールを作成してしまう．提案手法は，単一のライセンスからなるクラスタを多く作成するよう，4.1.3 章の処理 3 で示した，クラスタ間の距離に一定の条件を設けてクラスタの枝刈りを行う．本 RQ では，このクラスタリングにより，複数の異なるライセンスからなるクラスタの作成をどれだけ抑制できているかを確認する．

アプローチ： 4.1.2 章の処理 2 での GPL/BSD family ライセンスのフィルタリングの後に残った未知ライセンスに対しクラスタリングを適用する．クラスタリングによって出力されたクラスタを表 4.11 に示すメトリクスにより評価する．SLC は，単一のライセンスからなるクラスタを表し，RSLC は，作成されたクラスタにおける単一のライセンスからなるクラスタの割合を表すものである．RSLC の値が高いほど，単一のライセンスからなるクラスタを多く作成できていると言える．反対に，NSLC は，複数のライセンスからなるクラスタを表し，RNSLC は，作成された全クラスタ数における複数のライセンスからなるクラスタの割合を表すものである．RNSLC は低い方が望ましい．また，クラスタにならなかった（類似するライセンス記述ない）ライセンス記述を SF として示す．SF はできるだけ作成しないほうが望ましい．1 つのライセンス記述からは記述パターンが抽出できないため，ライセンスルールを生成することができない．作成されたクラスタ数に占める SF の割合を RSF と定義する．

また，評価結果を比較するため，クラスタリングの条件を追加する．提案手法（以下，**条件** α とする）は，2 つ以上のデータ，あるいはクラスタがマージされることによってしかクラスタが作成されない．図 4.7 の右側にあるような 1 つのデータと 1 つのクラスタとのマージによるクラスタ

表 4.10: Debian-7.8.0 における GPL/BSD family ライセンス分類の評価

License	TP	FP	FN	適合率	再現率	F 値
AGPLv3+	11	0	0	1.00	1.00	1.00
AGPLv3	3	1	1	0.75	0.75	0.75
GPLv1+	6	0	0	1.00	1.00	1.00
GPLv1	-	-	-	-	-	-
GPLv2+	94	3	67	0.97	0.58	0.73
GPLv2	72	5	75	0.94	0.49	0.64
GPLv3+	21	2	13	0.91	0.62	0.74
GPLv3	23	4	24	0.85	0.49	0.62
GPLv2.1+	18	1	13	0.95	0.58	0.72
LGPLv2.1	13	1	24	0.93	0.35	0.51
LGPLv3+	7	1	1	0.88	0.88	0.88
LGPLv3	13	3	8	0.81	0.62	0.70
LibraryGPLv2+	9	5	0	0.64	1.00	0.78
LibraryGPLv2	6	0	0	1.00	1.00	1.00
BSD4	2	0	4	1.00	0.33	0.50
BSD3	16	1	42	0.94	0.28	0.43
BSD2	17	27	6	0.39	0.74	0.51
multiple	0	7	0	-	-	-
全体	331	61	199	**0.86**	**0.62**	0.72

表 4.11: クラスタリング結果を評価するためのメトリクス

メトリクス	概要
C	提案手法のクラスタリングにより作成されたクラスタ数.
SLC (Single License Clusters)	単一のライセンスからなるクラスタ．本研究では SLC がより多く作られるのが望ましい
NSLC (Non-Single License Clusters)	複数のライセンスからなるクラスタ．本研究では NSLC の作成は望まれない
RSLC (Ratio of SLC)	作成されたクラスタ数に占める SLC の割合（SLC/C） 本研究では高い方が望ましい
RNSLC (Ratio of NSLC)	作成されたクラスタ数に占める NSLC の割合（NSLC/C）本研究では低い方が望ましい
SF (Single Files)	クラスタにならなかったファイル．提案手法では，単一ファイルからライセンスルールは生成できないため，本研究では SF が作成されるのは望ましくない
RSF (Ratio of SF)	作成されたクラスタ数に占める SF の割合 (SF/C)． 低い方が望ましい

を作成することができない．そこで，比較手法として条件 α を拡張した**条件** β を設定する．条件 β では，1 つのデータをクラスタと扱い，1 つのデータと 1 つのクラスタでマージされたクラスタを切り出すことを許可する．以下（式 4.6）に示す条件は，図 4.11 中の $Cm-1, Cm-2, Cn-1$ と Cn (例：X7) をクラスタとして切り出したい場合に有効である．

$$H(C_m) > H(C_{m-i}) \ \ or \ \ H(C_n) > H(C_{n-j}) \tag{4.6}$$

結果： 表 4.12 に 2 つの条件でのクラスタリング結果を 7 つのメトリクスで評価した結果を示す．表 4.12 から，OSS ライセンスの種類よりも多く単一のライセンスからなるクラスタ（SLC）が作成されていることがわかる．例えば，Linux-4.4.6 の条件 α では，332 の SLC が作成されているが，ライセンスは 18 種類しか存在しない．つまり，平均して約 18 クラスタ（=332/18）が同じライセンスに属していることを表す．

また，NSLC は多くのライセンスを持っていることが分かった．例えば，Linux-4.4.6 の条件 α では，33 の NSLC が作成されていて，その中に 17 種類のライセンスが存在している．つまり，1

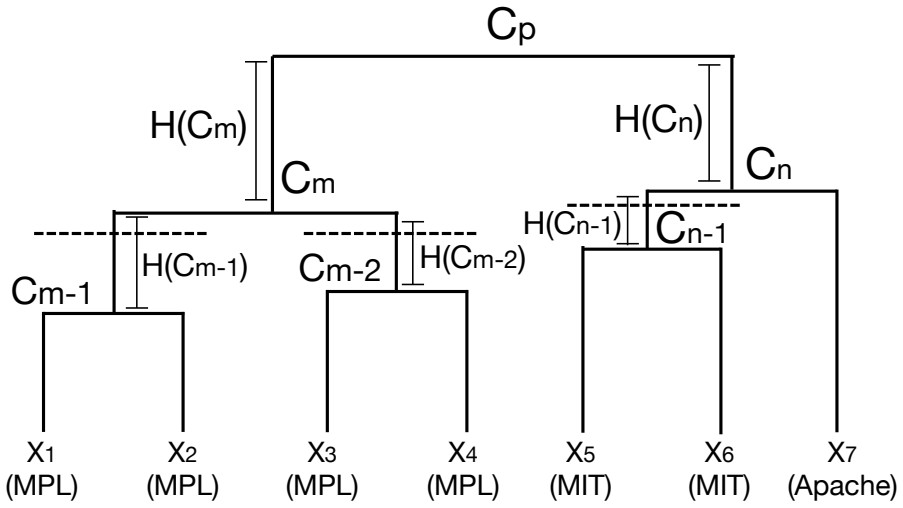


図 4.11: 条件 β によるクラスタ作成

つの未知ライセンスに対して平均 18.4（332/18）個のクラスタが属していることを表す．

また，本ケーススタディの中で，SLC、NSLC ともに、OSS ライセンスとは直接関係のない記述も多く含まれていることが分かった．例えば，ライセンス記述として誤検知されたソースコードなどが SLC や NSLC に含まれていた．これらの混入はクラスタリングの精度を曖昧にするため，今後，NSLC の作成をより低減するためにも，ライセンス特定ツールによるライセンス記述の抽出精度の向上が求められる．

今回データセットとして利用した 3 プロジェクト全てにおいて，条件 α よりも条件 β の方が，クラスタ数，SLC 共に多くの SLC を作成した．条件 β はまた，RSLC も僅かであるが条件 α より高くなっている．Linux-4.4.6 や，FreeBSD-10.3.0 の RSLC は両条件とも高くなっている．例えば，Linux-4.4.6 では 0.907（条件 α），0.932（条件 β），FreeBSD-10.3.0 では，0.917（条件 α），0.939（条件 β）となっている．一方，Debian-7.8.0 では， 0.691 (条件 α），0.605（条件 β）となっており，他のプロジェクトより低くなっている．そのため，特に Linux-4.4.6 と FreeBSD-10.3.0 では，条件 β で作成されたクラスタでは，複数のライセンス記述を含まない傾向にあるといえる．しかしながら，条件 β は非常に多くのクラスタを作成する．Linux-4.4.6 で作成されたクラスタは，条件 α では 322，条件 β では 745 となっている．他のプロジェクトも同様で，FreeBSD-10.3.0 では，121（条件 α），247 (条件 β），Debian-7.8.0 では，170（条件 α），433（条件 β）となっている．提案手法では，作成されるクラスタが多いと，正規表現も同時に多く作成される．SF や RSF も同様に条件 α より，条件 β の方が高いことから，最終的な正規表現の名前付け作業のコストは条件 β の方が高くなると想定する．

以上から，RQ6 の結果は以下のように回答する．

表 4.12: メトリクスによるクラスタリングの評価
() 内はライセンス種類数を示す

プロジェクト		Linux-4.4.6	FreeBSD-10.3.0	Debian -7.8.0
条件 α	**C** (# of created clusters)	355	132	246
	SLC (# of single license clusters)	**322** (18)	**121** (31)	**170** (41)
	NSLC (# of non-single license clusters)	33 (17)	11 (40)	76 (144)
	RSLC (Ratio of SLC)	**0.907**	**0.917**	**0.691**
	RNSLC (Ratio of NSLC)	0.093	0.083	0.309
	SF (# of single files)	38 (5)	16 (10)	48 (25)
	RSF (Ratio of SF)	**0.107**	**0.121**	**0.195**
条件 β	**C** (# of created clusters)	799	263	716
	SLC (# of single license clusters)	**745** (22)	**247** (45)	**433** (52)
	NSLC (# of non-single license clusters)	54 (12)	16 (31)	283 (135)
	RSLC (Ratio of SLC)	**0.932**	**0.939**	**0.605**
	RNSLC (Ratio of NSLC)	0.068	0.061	0.395
	SF (# of single files)	117 (7)	47 (15)	203 (39)
	RSF (Ratio of SF)	**0.146**	**0.179**	**0.284**

RQ6 の回答

2つの条件でクラスタリングを適用した結果，両方で高い割合で単一の単一のライセンスからなるクラスタを生成することができた．提案手法の条件 α では，Linux-4.4.6 では約 91%，FreeBSD-10.3.0 では約 92%，Debian-7.8.0 では約 69%が単一のライセンスからなるクラスタであった．比較手法である条件 β では，Linux-4.4.6 では約 93%，FreeBSD-10.3.0 では約 94%，Debian-7.8.0 では約 61%が単一のライセンスからなるクラスタであり，2 プロジェクトでは，条件 β の方がわずかに高い傾向となっている．ただし，作成されたクラスタ数は，条件 β の方が多いため，最終的な正規表現の名前付け作業のコストは条件 β の方が高くなると想定される．

4.3.3 RQ7: 編集距離を用いて単一のライセンスからなるクラスタにフィルタリングすることができるか

動機： 正規表現は，単一のライセンスからなるクラスタから抽出されるべきである．複数の異なるライセンスからなるクラスタからは，複数の異なるライセンスにマッチする正規表現を抽出してしまう．提案手法は，複数の異なるライセンスから正規表現を作成しないよう，GPL/BSD family のライセンスの分類とそれ以外のライセンスについてはライセンス記述をクラスタリングしている．ただし，クラスタリングは完全ではない．複数の異なるライセンスが混入しているクラスタを作成してしまった場合に備え，編集距離によって定義される類似度により，クラスタで大きく異なるライセンス記述をフィルタリングしている．本 RQ では，編集距離により複数の異なるライセンスからなるクラスタの作成をどれだけ抑制できているかを確認する．

アプローチ： 編集距離を用いたライセンス記述のフィルタリング後のクラスタを表 4.11 に示すメトリクスにより評価する．SLC は，単一のライセンスからなるクラスタを表し，RSLC は，作成されたクラスタにおける単一のライセンスからなるクラスタの割合を表すものである．RSLC

表 4.13: フィルタリング後のメトリクス（表 4.11）による評価

プロジェクト	評価するクラスタ	C	SLC	Non-SLC	RSLC	RNonSLC	SE
FreeBSD-10.3.0 (1,230 ファイル)	クラスタリング後	132	121	11	0.92	0.08	16
	外れ値除去後	148	148	0	**1.00**	0.00	221
Linux-4.4.6 (2,801 ファイル)	クラスタリング後	355	322	33	0.91	0.09	38
	外れ値除去後	372	370	2	**0.99**	0.01	988
Debian-7.8.0 (2,446 ファイル)	クラスタリング後	246	170	76	0.69	0.30	48
	外れ値除去後	166	158	8	**0.95**	0.05	1917

の値が高いほど，単一のライセンスからなるクラスタを多く作成できていると言える．本 RQ では，RSLC が 100%となることを理想とする．また，フィルタリングされてしまったライセンス記述は SE として扱う．

結果： 表 4.13 に，編集距離を持ちたフィルタリング後の評価結果を示す．FreeBSD-10.3.0 のクラスタリング結果は，132 個のクラスタのうち 121 個が SLC であり，RSLC は 0.92%であった．一方，フィルタリング後は，クラスタが 148 個に増加し，その 100%にあたる 148，つまり全てのクラスタを SLC にできていることが分かった．Linux-4.4.6 のクラスタリング結果は，355 個のクラスタのうち 322 個が SLC であり，RSLC は 0.91%であった．一方，フィルタリング後は，クラスタが 372 個に増加し，その 99%にあたる 370 が SLC となることが分かった．Debian-7.8.0 のクラスタリング結果は，246 個のクラスタのうち 170 個が SLC であり，RSLC は 69%であった．一方，フィルタリング後はその 95%にあたる 158 が SLC となることが分かった．

以上から，RQ7 の結果を以下のように回答する．

RQ7 の結果

クラスタリング後に編集距離を用いたフィルタリングを適用することで，FreeBSD-10.3.0 では 92%から 100%，Linux-4.4.6 では 91%から 99%に，単一のクラスタの割合を 100%近くにすることができた．また，Debian-7.8.0 では 69%から 95%に向上させることができた．

4.3.4 RQ8: どれくらいの正規表現を生成できるか

動機： 3.3.2 章にて，正規表現を作成しすぎないことの重要性について述べた．多くの正規表現が生成されると，後続の作業である名付け作業にかかる労力が大きくなる．本手法では，頻述するライセンス文の単語出現順序（系列パターン）を元に正規表現を生成することで，ルールの大量生成を抑止している．本 RQ では，提案手法が出力した正規表現の数を確認し評価する．

アプローチ： フィルタリング後の各クラスタに対し，BIDE による系列パターンマイニング（処理 5），記述パターンからのライセンスルールへの変換（処理 6）を行い，正規表現を生成しその数をカウントする．また．比較対象として，ライセンス記述をそのまま正規表現として追加する手法を AS–IS を設定する．AS–IS は，最も単純に正規表現を作成する方法であり，ライセンス記述を文単位に分解し，重複を排除したものである．

結果： 表 4.14 に提案手法が生成した正規表現の例を示す．提出手法ではメタ文字を利用した正規表現を生成できた．FreeBSD-10.3.0 では，RSA-MD の正規表現を生成することができた．また，

表 4.14: メタ文字を使用して生成された正規表現の例

	ライセンス名	正規表現
FB	RSA-MD	License(is also granted)? to(copy| make) and use(this software is granted| derivative work) provided[...] RSA Data Security, Inc
LI	GPL	the(terms of the)? GNU General Public License
DE	CeCILL-C	software(is governed by| under) the (terms of the)? CeCILLC license (under French law and abiding| as circulated)[...]

表 4.15: 生成された正規表現の数

プロジェクト	ライセンス文数	生成手法	正規表現数
FreeBSD-10.3.0	9,435	Proposal Method	**457**
		AS–IS	1,422
Linux-4.4.6	8,661	Proposal Method	**269**
		AS–IS	2,916
Debian-7.8.0	9,807	Proposal Method	**419**
		AS–IS	5,932

Linux-4.4.6 では，GPL ライセンスの正規表現が，Debian-7.8.0 では CeCILL-C ライセンスの正規表現を生成した．その他の正規表現については，表 4.17 に付録として掲載する．

表 4.15 に提案手法と AS–IS で生成された正規表現数を示す．FreeBSD-10.3.0 で生成された正規表現は，提案手法は 457，AS–IS では 1,422 であった．Linux-4.4.6 では，提案手法は 269，AS–IS では 2,916 であった．FreeBSD-10.3.0 と Linux-4.4.6 は全体のライセンス文に対し，AS–IS で作成された正規表現数はそれぞれ 1,422(15%)，2,916(33%) となるため，重複するライセンス文が多く存在していたことが分かる．また，Debian-7.8.0 では，AS–IS では 5,932 もの正規表現が生成されたが，提案手法は 419 であり，提案手法の方が生成された正規表現数が少なかった．AS–IS で生成された正規表現は 5,932 で全体の 60%程度と高くなっており，これは，Debian-7.8.0 が，他のプロジェクトに比べて重複するライセンス文が少ないことを示す．一方，提案手法が生成した正規表現数は 419 と他のプロジェクトと大差がない．そのため，ライセンス文の重複の少ないデータセットでも正規表現を生成できる手法といえる．以上の結果から，RQ8 の回答は以下とする．

RQ8 への回答:

提案手法で生成された正規表現は，FreeBSD-10.3.0 では 457，Linux-4.4.6 では 269，Debian-7.8.0 では 419 であり，提案手法の方が生成される正規表現数が少ないことが分かった．また，提案手法は，ライセンス文の重複が少ないデータセットの場合でも正規表現を生成できる手法であることが分かった．

表 4.16: 各メトリクスによる生成した正規表現のパフォーマンス評価

プロジェクト	生成手法	正規表現数	マッチしたライセンス文	ER	MR	PR
FreeBSD-10.3.0	Default	551	6,376	**0.91**	0.68	0.78
	Proposal Method	1,008	8,281	0.88	0.88	**0.88**
	AS–IS	1,973	9,435	0.79	**1.00**	**0.88**
Linux-4.4.6	Default	551	5,309	**0.90**	0.61	0.73
	Proposal Method	820	5,796	0.86	0.67	**0.75**
	AS–IS	3,467	8,661	0.60	**1.00**	**0.75**
Debian-7.8.0	Default	551	4,357	**0.87**	0.44	0.59
	Proposal Method	970	5,191	0.81	0.53	**0.64**
	AS–IS	6,483	9,807	0.34	**1.00**	0.51

4.3.5 RQ9: 生成した正規表現はどれくらいの未知ライセンス文とマッチするか

動機： ライセンス特定を支援する観点においては，より多くのライセンス文を特定できることも同じくらい重要である．本 RQ は，トレードオフ関係にある 2 つの要因から提案手法が生成する正規表現の性能について説明する．

アプローチ: 提案手法と AS–IS 手法で生成した正規表現をそれぞれ Ninka の標準搭載の正規表現と合わせて，ルール生成に利用した各データセットのライセンス記述とマッチングを行う．生成した正規表現を追加することによって，マッチしたライセンス文の増加量を評価する．このとき AS–IS は全てのライセンス文にマッチすると仮定する．次に，以下のメトリクスを定義し，各手法について 2 つの要因の調和平均を評価する．式 4.7 で示す Efficiency of Rules (ER) は正規表現がマッチするライセンス文する割合を正規化したものである．ER 値が高いほど，最小限の正規表現を生成することを示す．式 4.8 で示す Matching Rate (MR) は，全体におけるマッチしたライセンス文の割合である．MR 値が高いほど，多くのライセンス文とマッチすることを示す．式 4.9 で示す Performance of rules (PR) は，ER と MR の調和平均を示す．PR 値を算出し，提案手法が他の手法より高いかどうかを評価する．また，参考として，初期状態の Ninka の正規表現性能を評価する．Ninka には初期状態で 551 の正規表現が搭載されている．

$$ER = 1 - \frac{\#\ rules\ for\ sentence\ matching}{\#\ matched\ sentences} \tag{4.7}$$

$$MR = \frac{\#\ matched\ sentences}{\#license\ sentences} \tag{4.8}$$

$$PR = \frac{2 * ER * MR}{ER + MR} \tag{4.9}$$

結果： 図 4.12 に，特定に用いた正規表現の数とマッチしたライセンス文の数を示す．Default は，Ninka の標準搭載の 551 の正規表現で，FreeBSD-10.3.0 では 6,376（全体の 67.6%），Linux-4.4.6 では 5,309（全体の 61.2%），Debian-7.8.0 では 4,357（44.4%）のライセンス文とマッチした．

提案手法の特定できたライセンス文に着目する．FreeBSD-10.3.0 では，8,281（87.8%）のライセンス文に，Linux-4.4.6 では，5,796（66.9%）のライセンス文に，Debian-7.8.0 では，5,191（52.9%）

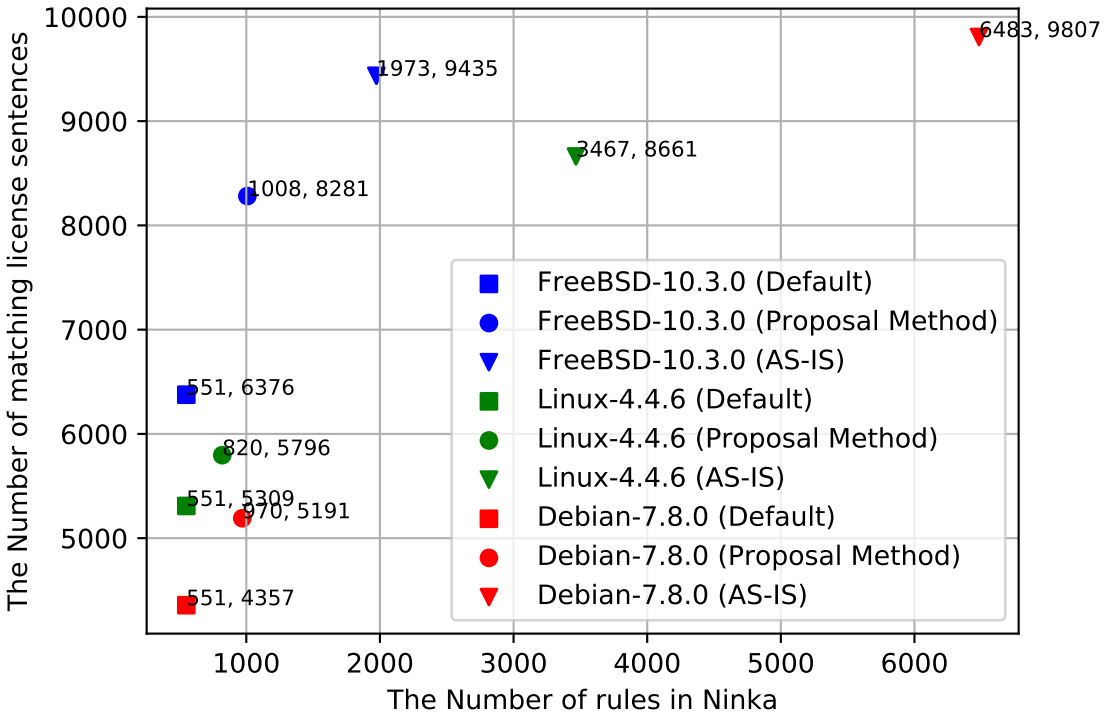


図 4.12: ライセンス特定に使用した正規表現 (rules) の数とマッチしたライセンス文

のライセンス文に，それぞれマッチしたことがわかった．一方，AS–IS では，各プロジェクトの全てのライセンス文にマッチするように前提を置いているが，非常に多くの正規表現を必要としている．

AS–IS では，Ninka の正規表現と合算すると FreeBSD-10.3.0 では 1,973 もの正規表現を，Linux-4.4.6 では 3,467 もの正規表現を，Debian-7.8.0 では 6,483 もの正規表現にもなる．一方，提案手法では，Ninka の正規表現と合算すると FreeBSD-10.3.0 では 1,008 の正規表現，Linux-4.4.6 では 820 の正規表現を，Debian-7.8.0 では 970 の正規表現となり，AS–IS より少ない正規表現数でライセンス特定を行う．

表 4.16 にライセンス文のマッチングに使用した正規表現と，マッチングしたライセンス文から算出した ER 値，MR 値，PR 値を示す．Default の ER 値は，Free-BSD-10.3.0 では 0.91，Linux-4.4.6 では 0.90，Debian-7.8.0 では 0.87 で全てのプロジェクトにおいて，他の手法より最も高くなっている．これは手作業で作成された Ninka に標準搭載されている正規表現の方が多くのライセンス文にマッチする正規表現であることを示す．

AS–IS の MR 値は，全てのプロジェクトで 1.00 であり全手法より高い．これらは，提案手法と AS–IS 両方で MR 値は向上しているため，正規表現を生成・追加すると特定できるライセンス文が増加することを示している．Default の PR 値は，Free-BSD-10.3.0 では 0.78，Linux-4.4.6 では 0.73，Debian-7.8.0 では 0.59 であった．AS–IS の PR 値は，Free-BSD-10.3.0 では 0.88，Linux-4.4.6 では 0.75，Debian-7.8.0 では 0.64 であった．AS–IS は，Debian-7.8.0 の PR 値が Default より低くなっているため，Debian では，正規表現のパフォーマンスを低下させる手法であることがわかった．

提案手法の PR 値は，FreeBSD-10.3.0，Linux-4.4.6 では，AS–IS と同等，Debian-7.8.0 では，

0.64 であった．そのため，正規表現のパフォーマンスは提案手法が最も優位な手法であるといえる．以上の結果から，RQ9 の回答は以下とする．

RQ9 への回答:

提案手法が生成した正規表現の PR 値は，FreeBSD-10.3.0 は 0.88，Linux-4.4.6 では 0.75，Debian-7.8.0 では 0.64 であり，AS–IS や Ninka の初期搭載の正規表現よりも高い値を示した．

4.4 考察

4.4.1 GPL/BSD family の分類精度向上

RQ1 の結果から，全体的には FreeBSD-10.3.0，Linux-4.4.6 では高い適合率で分類することができた．しかしながら，各ライセンスごとの適合率に着目すると，BSD2 が最も低いことが分かる．BSD2 の適合率が低い原因として，ライセンス記述のバリエーションに対応できない正規表現であったことや，正規表現が曖昧であったため，対応すべきではないライセンス記述に対応してしまっていることが考えられる．まず．類似関係にあるライセンスの判別に失敗した全ての未知ライセンス記述を目視し．その失敗原因を特定した．BSD2 の失敗事例では，まず，Apache License v1.1 の誤検知が多く見られた．Apache License v1.1 のライセンス記述は，BSD2 にさらに条項を追加，改変して作成されている．そのため，BSD2 の判別条件に，Apache License v1.1 の一部のライセンス記述に対応してしまうことが分かった．対策として，“Apache” という単語が含まれる場合，除外する処理を追加することが考えられる．

一方で，再現率は，適合率より低くなっている．これは，ライセンス記述内の表記ゆれにより BSD ライセンスの検出に失敗する事例が多く見られた．中でも，“The name of ”，“Neither name of” の表記ゆれが多く存在した．そのため，それらのパターンを特定する正規表現に追加する対策が考えられる．また，GPL ライセンスの “or Later” オプションを検出できない事例が多く見られた．例えば，“or above”，“or any newer version” と表記されているものが存在した．そのため，それらのパターンを，or Later を判別する条件に加えることが考えられる．

4.4.2 ライセンスルール生成に適した階層クラスタリングの条件

RQ2 では，3 つの OS ドメインのデータセットを用いて階層クラスタリングの実験を行い評価を行った．表 4.12 から， FreeBSD-10.3.0 と Linux-4.4.6 においては，条件 β のほうが条件 α より，単一のライセンスからなるクラスタの割合（RSLC）が高く，複数のライセンスからなるクラスタ（RNSLC）の割合が低い．ライセンスルール自動生成の観点では，クラスタの中には，複数種類のライセンスは存在すべきではない．本研究の最終的なゴールは Ninka に追加する正規表現を自動生成することである．そのため，条件 β の方が，RNSLC の値が条件 α より低くく，正規表現の自動生成に適しているといえる．しかしながら，条件 α と条件 β の間での RSLC や RNSLC の値の差異はわずかとなっている．RSLC の値は，Linux-4.4.6 では 0.907（条件 α）と 0.932（条件 β），FreeBSD-10.3.0 では 0.917（条件 α）と 0.939（条件 β），Debian-7.8.0 では 0.691（条件 α）と 0.605（条件 β）であった．生成されるライセンスルールの保守という観点では，条件 α の方が，条件 β よりも適してる．生成されるライセンスルールの数は，作成されるクラスタ数に依存する

ため，ライセンスルールに名前を付けるコストは条件 α の方が小さいといえる．Debian-7.8.0 の RSLC は条件 α (0.691)，条件 β (0.605) の両方で Linux-4.4.6 と FreeBSD-10.3.0 よりも低くなっている．これはソースファイルのサンプリング方法が他の 2 プロジェクトとは異なることに起因する．Debian-7.8.0 は，12,725 のソースファイルを各パッケージから 1 ファイルずつサンプリングしており，データセットには 194 種類もの未知ライセンスが存在する．そのため，1 ライセンスあたりのライセンス記述数が他のプロジェクトと比較して少なく，クラスタになりにくい傾向があったと考えられる．しかしながら，RQ7 での，クラスタリングの後続の処理である編集距離を用いたフィルタリング後の結果から，RSLC の値が 69%から 95%に上昇していた．そのため，たとえデータセットに多くのライセンスの種類が含まれていても，後続の編集距離によるフィルタリング処理で補完されるため，生成される正規表現の品質は十分に高いといえる．

4.4.3 ライセンスルール生成数の最小化に向けて

RQ8 では，提案手法と AS–IS 手法で生成した正規表現の数の評価を行った．生成される正規表現の数が少ないほど，名前付けに必要な労力は少なくなる．表 4.15 に示すよう，FreeBSD-10.3.0 では 457（提案手法）と 1,422（AS–IS），Linux-4.4.6 では 269（提案手法）と 2,916（AS–IS）Debian-7.8.0 では 419（提案手法）と 5,932（AS–IS）であった．これらから AS–IS より提案手法の方が生成する正規表現の数が少ないことが分かった．

特に，ライセンスの種類が多い Debian-7.8.0 では，AS–IS 手法は，非常に多くの正規表現を生成したのに対し，提案手法は他プロジェクトと同程度の生成数であった．提案手法は，頻出する単語列をベースに正規表現を生成するため，同様のライセンス記述が複数存在しなければ正規表現として生成されない仕組みとなっていることが挙げられる．ライセンス種類数が多く重複するライセンス記述が少ないデータセットでも，提案手法は，安定した数の正規表現を出力することができる．そのため，提案手法は，ライセンスルールの保守を安定して行えるメリットがあると考えられる．

さらに正規表現が生成される数を少なくする方法として，よりメタ文字を使用した正規表現を生成することが挙げられる．メタ文字使ったルールの作成を増加させるには，フィルタリングの精度の向上が重要である．追加調査で，メタ文字を利用したライセンスルールは，処理 4（4.1.4 章）でのクラスタに属するライセンス記述のフィルタリングにより，表記揺れを含むライセンス記述が多く排除されていたことがわかった．その原因調査の結果，Ninka がライセンス記述だけではなく，アプリケーションの説明までも抽出していることがわかった．このような記述がライセンス記述と同時に抽出された場合，他のライセンス記述とライセンスが同一であっても類似度が低く算出されてしまいフィルタリングされてしまっていたことが分かった．Ninka は，ライセンスに関連する単語を用いてライセンス記述を抽出する．ライセンス記述の抽出に利用される単語を調査したところ，“copies” や，“modified’ などのソースコードの説明にも頻出する単語も含まれていた．そのため，正規表現の集約率を向上するには，ライセンス記述に関連する単語の見直しを行い，ライセンス記述をより正確に抽出することが重要だと考えられる．ただし，基本的にはこのフィルター処理 (4.1.4) は，一部の酷似した異なるライセンス記述を除外できるよう高めに（94%）に設定している．類似度の差異について，法的な意味が異なるかどうかの見極めができない以上，多少の改善が期待できる程度であり，Ninka に標準搭載されているほどの複雑なメタ文字

の生成は難しいと考えている．

4.4.4 手作業によるライセンスルール作成の効率化

RQ9 では，前述の RQ8 の正規表現数とマッチしたライセンス文の数の調和平均をライセンスルールのパフォーマンスとして評価した．ライセンス特定ツールに生成した正規表現を追加することで，Default のライセンスルールの PR 値を改善することが分かった．また，提案手法と AS–IS 手法と比較した場合，その PR 値は同等以上であった．追加されるライセンス文は少ない方が労力が少ないため，PR 値が AS–IS と同等であったとしても，ER 値が優位な提案手法のほうが，ライセンスルール作成の効率化という観点では優位であるといえる．

また，Ninka に標準搭載されている正規表現（Default）との比較を行なっている．提案手法は，Default の ER 値と比較すると提案手法の方が劣位である．これは，手作業で作成された Ninka の正規表現の方が，効率的なルールであることを示す．手作業でルールを作成する場合，ライセンス記述に関する表記揺れなどの知識を持っている開発者が作成すると考えられる．今後の展望として，提案手法の ER 値を改善するには，既に判明している表記揺れを一般化するなどの対応を加えることが有効であると考えられる．

実際にライセンス文とどれくらいマッチしたかを示す MR 値は FreeBSD-10.3.0 は 0.88，Linux-4.6.0 は 0.67，Debian では 0.53 であった．これらは最低でも 53%のライセンス文にマッチしたということであり，手作業で正規表現を作成した場合と比較すると十分支援できていると考えている．特に FreeBSD-10.3.0 は全体の 88%と大部分のライセンス文とマッチできておりライセンスルール作成の効率という観点では十分に支援できているといえる．

4.4.5 制約

内的妥当性への脅威

本研究では，ライセンスルール自動生成手法を提案した．また，実際にケーススタディを行い評価している．提案手法の中で，3 つの閾値を設定している．4.2.2 章に記載の編集距離によるクラスタ内のフィルタリング閾値 (1)，BIDE サポート数 (2), ライセンスルールに適したサポート数 (3) がある．これらの閾値を変更した場合，本研究結果と異なる結果になる可能性がある．今回のデータセットに関しては，最適な閾値を設定できていると考える．

外的妥当性への脅威

本研究でのケーススタディは， FreeBSD-10.3.0，Linux-4.4.6，Debian-7.8.0 の 3 つの OSS プロジェクトのソースファイルで行っている．これらは著名な OSS の代表例であるが、他の OSS プロジェクトでは実験を行っていない．今後，ライセンスルール自動生成手法を他の OSS に対しても適用し、本アプローチの汎用性を向上させる予定である．

本研究では，提案手法を評価するため，FreeBSD-10.3.0，Linux-4.4.6，Debian-7.8.0 から検出された未知ライセンスのライセンス名を筆者が手作業で特定している．ただし，筆者は法律の専門家として従事していないため，ライセンス特定の際のライセンス記述は筆者の解釈に依存する．

今後、本提案手法をより厳格に評価するため，法律の専門家と協力しデータセットの未知ライセンスのエビデンスを作成し，評価結果の一般性をより一層高める必要がある．

4.5 関連研究

4.5.1 ライセンス特定ツール

現在多くのライセンス特定ツールが提案されている．ライセンス特定ツールは大きく分けて，文書間の類似度を用いる手法 [15] と，正規表現による手法 [16,17,42] がある [14] がある．文書間の類似度を用いる手法として，Fossology [15] がある．Fossology は，タンパク質の塩基配列のパターンマッチングに用いられる技術に基づいた bSAM アルゴリズムにより，入力となるライセンス記述とライセンスルールのライセンス記述との類似度を算出している．類似度を用いている場合，正規表現を用いた場合と比べて，ライセンスルールを多く搭載する必要がないため，高い再現率が期待できるが，ライセンスのバージョンなど法的に意味が異なる文字列の差異を区別するのは難しいため，適合率が低くなる傾向にある．また，類似度の算出に時間がかかるというデメリットもある．ASLA や OSLC は，1 分間に 4000 ファイルのライセンスを特定できるのに対し，Fossology は，1 分間に 1 ファイルしか特定できないことが分かっている [18]．一方で，正規表現を用いているものには，Ohcount [17]，ASLA [18]，OSLC [16] がある．正規表現を用いた場合，完全一致によりライセンスを特定するため高い適合率が期待できる．ただし，Ohcount [17]，OSLC [16] は，単純な正規表現を用いているため，ライセンサの意図と反対のライセンス特定結果を出力することがあることが German らにより報告されている [14]．例えば，“This file is not licensed under the GPL” に対し，GPL と判定されているものが発見されている．また，対応するライセンスルールが作成されていないライセンスについては特定することが出来ないため，再現率がライセンスルールの量や種類に依存する．本研究では，速度や適合率の面から，正規表現による手法である Ninka に着目し，そのデメリットであるライセンスルールの作成を支援することが本研究の目的である．ソースファイルだけでなく，jar アーカイブ [43] やバイナリ形式 [44] のファイルのライセンスを特定するライセンス特定ツールも提案されている．これらの研究は，ソースファイル以外の形式のファイルに対してライセンス特定を行うものであるが，手法の一部に Ninka が使用されているため，本研究が適用可能であると考えている．

4.5.2 ライセンス違反検出

ライセンス違反検出に関する研究が行われている．Mlouki ら [22] は，Android Ecosystem を対象としたライセンス違反を検出する手法を提案している．Lokhamn ら [45] は，ソフトウェアアーキテクチャレベルで，ライセンス衝突を検出するツールを提案している．また，Alspaugh ら [46] は，ライセンスの権利と義務をタプルで表現し，ライセンスの衝突を計算する手法を提案している．German ら [47] は，ソフトウェアコンポーネントにおいてライセンスの衝突が発生した場合の解決方法をまとめた．German ら [19] は，ソフトウェアパッケージのメタデータのライセンスと，ソフトウェアパッケージに含まれるソースファイルのライセンスの incompatibilities を分析する手法を提案している．Di Penta ら [8] は，ソフトウェア開発履歴からライセンスの違法な変更がないかどうかを自動で追跡する手法を提案している．これらの研究は，最終的に，ライセン

ス違反のリスクを減らすことができる点において本研究と関連している．ただし，本研究は，ライセンス特定ツールの精度を上げることでライセンス違反リスクを減らすため，これらの研究のアプローチとは異なっている．

4.5.3 OSS のライセンスとソフトウェア開発

OSS のライセンスとソフトウェア開発に関する調査も行われている．Kashima [24] らは，開発者がソフトウェアの再利用性を考慮した上でライセンスを決定できるよう，ソースファイルのライセンスがコピーアンドペーストによる再利用にどのように影響を与えるのか調査している．Colazo ら [48] は，コピーレフト性のある OSS ライセンスを用いる OSS プロジェクトはコピーレフト性のない OSS ライセンスの OSS プロジェクトと比べて，開発者の地位やコーディング量，活動の永続性が高く，開発期間が短くなることを示している．Sojer ら [13] は，ソフトウェア開発者にソフトウェアの再利用に関するインタビューとライセンスに関するクイズを実施し，ソフトウェア開発者のライセンスに関する知識が不十分であることを明らかにした．ソフトウェア開発者が容易にライセンスルールの作成を行えるようにするための支援を最終目標にしている．Almeida ら [49] は，ソフトウェア開発者のライセンスの理解度を分析している．開発者は，人気なライセンスについて理解しているが，複数のオープンソースライセンスの組み合わせて使用されている場合，それらの技術的な相互関係について理解が十分でないことを明らかにしている．Meloca ら [29] は，OSI に認証されていないライセンスがソフトウェアプロジェクに与える影響について 657,000 の open-source project の開発データを対象に調査している．Vendome ら [50] は，開発者が OSS ライセンスをいつどのように決定しているのかを分析するためアンケートを行なっている．また，Vendome らは，バグトラッキングシステム上での 1200 件のライセンスに関する報告を分析している．これらの研究は，最終的に，開発者が OSS のライセンスを正しく認識した上でライセンスを決定できるようにする点において本研究と関連している．ただし，本研究は，再利用するソフトウェアのライセンスを正しく理解できるようにするため，これらの研究のアプローチとは異なっている．

4.5.4 系列パターンマイニング

本研究では，系列パターンマイニングアルゴリズムの 1 つである BIDE を用いて，ライセンス名が同一であるライセンス記述のクラスタから記述パターンを抽出している．系列パターンマイニングは，ソードコードの静的解析のドメインにおいてよく用いられている．例えば，本研究で用いている BIDE は，ソースコードから同時に利用される API の関係を抽出するのに用いられている [51, 52]．Zhong ら [53] は，系列パターンマイニングアルゴリズムの 1 つである SPAM [54] を用いて同時に利用される API の関係を抽出する手法を提案している．これらの研究は，ソースファイルに対して，系列パターンマイニングを行っている点において本研究と関連している．ただし，本研究は，ライセンスルール作成に必要なライセンス記述パターン抽出するために，系列パターンマイニングを行っている．そのため，これらの研究とは目的が異なっている．

4.6 まとめ

本研究では，手作業によるライセンスルール作成を支援するため，先行研究である，未知のライセンス記述のクラスタから，ライセンス文を特定するルール（正規表現）を生成する手法を提案した．アプローチとしては，ライセンス記述の各クラスタにフィルタリングを適用した後に系列パターンマイニングを行い，抽出されたパターンをライセンスルールに変換する．さらに，正規表現のメタ文字を利用し，複数のライセンスルールを 1 つのライセンスルールに集約している．提案手法を評価するために FreeBSD-10.3.0，Linux-4.4.6，Debian-7.8.0 から検出された 1,821，3,561，2,838 の未知ライセンスを用いてケーススタディを行った．その結果，提案手法が最小限のライセンスルールでより多くのライセンスルールを特定する手法であること，さらに，提案手法により作成されたライセンスルールを Ninka に追加することで，ライセンスルールのパフォーマンスが 2%–10%改善することが分かった．今後の展望としては，ライセンスルール生成後に発生する手作業によるルール命名の支援を行う技術の構築が考えられる．ライセンスルール命名は必ず人手で行う必要があるが，ライセンス記述の内容と OSI や FSF に登録されているライセンス名の関係から複数ライセンス命名案を提示するなどを行えれば，より開発者にとって使いやすい技術に発展すると考えられる．

表 4.17: 【付録】提案手法で生成したライセンスルールの例（一部抜粋）

プロジェクト	ライセンスルール	ライセンス名
FB	Refer to the GNU DIFF General Public License [...] Among other things, the copyright notice [...] Everyone is granted permission to copy, modify and redistribute GNU DIFF, but only under the conditions described in the GNU DIFF General Public License. GNU DIFF is distributed in the hope that it will be useful, but WITHOUT[...]. A copy of this license is supposed to have been given to you along with GNU DIFF so you can know your rights and responsibilities.	GNU DIFF GPL
	A copy of the CDDL is also available via the Internet at [...] This file and its contents are supplied under the terms of the The Inner Net License Version 3 applies to this software. Common Development and Distribution License (“ CDDL ”), version 1.0. You may only use this file in accordance with the terms of version 1.0 of the CDDL.	CDDL v1.0
	You should have received a copy of the license with this software. If you didn ’ t get a copy, you may request one from [...]	Inner Net License v3
	I let hardware handle unaligned except on page boundaries (see below for details). The module is, however, dual licensed under OpenSSL and CRYPTOGAMS licenses[...]	Open SSL and CRYPTOGAM
	THIS SOFTWARE IS PROVIDED BY THE AUTHOR [...] IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR [...] Redistribution in binary form must reproduce [...] Redistribution of source code must retain[...] Redistribution and use in source and binary forms, [...]	BSD2
LI	This file is (c) under GNU PUBLIC LICENSE	GPL
	Released under the term of the GNU GPL v2.	GPLv2
	Use and redistribute under the terms of the GNU GPL v2.	GPLv2
	Licensed under the terms of the GNU GPL License version 2.	GPLv2
	Released according to the GNU GPL, version 2 or any later version.	GPLv2
	This program is free software; you can redistribute it and/or modify it	GPLv2
	under the terms of the GNU General Public License 2 [...]	GPLv2
	This program is distributed in the hope that it will be useful, but WITHOUT [...] See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; see the file COPYING. If not, write to the Free Software Foundation,[...]	GPLv2
DE	This software is subject to [...] Zope Public License, Version 2.1 (ZPL) THIS SOFTWARE IS PROVIDED “ AS IS ” AND ANY AND ALL [...]	ZPLv2.1
	This software/database is a ”United States Government Work ” under the [...] PUBLIC DOMAIN NOTICE National Center for Biotechnology Information The National Library of Medicine and the U.S. Government have not plac[...] The NLM and the U.S. Government disclaim all warranties, express or im[...]	Public Domain
	2 - This Source Code Form is subject to the terms of the Mozilla Publi[...] If a copy of the MPL was not distributed with this file, You can obtai[...]	MPLv2.0
	You should have received a copy of the GNU Lesser General Public Lic[...] This program is distributed in the hope that it will be useful, but WITH[...]	LGPL

第5章　Dockerイメージ開発時におけるライセンス非互換の予測

本章では，3.3 章で説明した技術的課題 T2，開発中の Docker イメージの最終的なライセンス互換検証結果の予測方法と，その評価のために行ったケーススタディについて説明する．

5.1　ライセンス互換性検証結果予測

Docker イメージの最終的なライセンスの互換検証結果を開発早期で予測する方法について説明する．図 5.1 に提案手法の概要を示す．提案手法は大きく分けて学習フェーズと予測フェーズの 2 つに分けられる．学習フェーズでは，3.2 章での GPL 互換性分析と同様に，まず，多数の Docker イメージのパッケージとそのライセンスを特定する．そして，表 3.4 に基づき，Docker イメージの OSS パッケージ中に GPL 非互換となるライセンスの組み合わせを検出する．次に，各イメージをパッケージの利用情報でベクトル化し（本研究では bag-of-pkgs ベクトルと呼ぶ），GPL 非互換の検証結果のアノテーションを行う．

次にこの学習データから機械学習アルゴリズムの 1 つである多層パーセプトロンを用いて予測モデルを生成する．3.3.3 章で述べた技術的課題 T2 では，パッケージの依存関係の最下位のパッケージは最初にインストールされる点について述べた．そのため，本研究では，ライセンス非互換の原因となるパッケージと依存関係のあるパッケージが開発早期でインストールされていれば，最終的なライセンス互換性検証結果を予測できると考えた．Docker イメージは開発が進むほど，インストールされるパッケージ数は増加していく．今回，Docker イメージにインストールされるパッケージ情報は，時間によって変化する特性を持つことから時系列データとして捉える．この時系列を本研究では**開発進捗度と定義する．**開発進捗度 100%の状態は，Docker Hub などにアップロードされている既存の完成したイメージを指す．しかしながら，本研究では，開発早期の Docker イメージのライセンス互換性検証結果を予測するため，完成した Docker イメージの状態から開発早期の状態に擬似的に巻き戻す必要がある．本研究ではこれを Docker イメージに含まれているパッケージの利用情報をマスキングして削減することで実現する．

学習フェーズでは開発中の Docker イメージの情報量と合わせるため，各開発進捗度ごとに予測モデルを作成する．具体的には各イメージのパッケージ総数のうち 0%から 90%まで 5%間隔で利用情報をマスキングした学習データから予測モデルを準備する．これは予測対象となるイメージに，ライセンス互換性と依存関係の最下位にあたるパッケージが開発早期にインストールされるという仮説が成り立つ場合，開発進捗度 100%の Docker イメージのパッケージ情報を用いて予測するより，予測対象と同等の開発進捗度の Docker イメージにインストールされるパッケージ情報のみを学習して予測する方が精度が高くなると考えたためである．

最後に予測フェーズにて，開発中の Docker イメージのパッケージ情報から将来的なライセンス

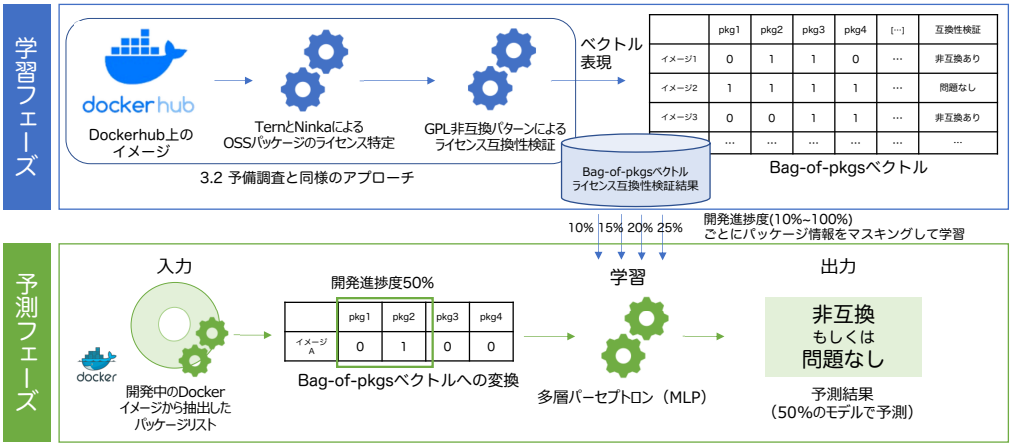


図 5.1: ライセンス互換検証結果予測の概要

互換性検証結果を予測する．このとき予測対象の Docker イメージの開発進捗度と同一の学習データから作成した予測モデルを用いる．以降，それぞれの処理の詳細について説明する．

5.1.1 パッケージ情報を用いた Docker イメージのベクトル化

まず，Docker イメージから Tern と Ninka を用いて OSS パッケージの名前とそのライセンス情報を抽出する．手順は 3.2 章での Docker イメージのライセンス互換性検証の予備調査と同様である．

次に，抽出したパッケージ情報から bag-of-pkgs ベクトルを生成する．bag-of-pkgs ベクトルは各 Docker イメージを OSS パッケージの利用情報でベクトル化したものである．bag-of-pkgs ベクトルは，縦軸 Y に抽出に用いた Docker イメージ名，横軸 X に全 Docker イメージで利用されていたパッケージ名を取る．bag-of-pkgs ベクトルの各値 [x,y] には OSS パッケージの利用情報が入力されている．OS の制約上，同じパッケージは 1 つのコンテナに 1 つしかインストールできないため，各値 [x,y] は 0 (利用なし) か 1(利用あり) となる．

最後に各ベクトルに対し，表 3.4 の GPL 非互換の組み合わせの有無の検証を行う．その検証結果を各 bag-of-pkgs ベクトルにアノテーションしたものをデータセットとする．GPL 非互換の検証結果は，組み合わせが 1 つ以上存在した場合 1 (非互換あり)，なければ 0 (互換性に問題なし) でアノテーションする．

5.1.2 開発進捗度に合わせたパッケージ利用情報のマスキング

5.1.1 章で作成された bag-of-pkgs ベクトルは，完成した Docker イメージから作成されているため，開発進捗度が 100%の状態である．しかしながら，予測対象となる Docker イメージは開発途上を想定しており，当然ながら導入されているパッケージ数は少なくなる．そのため，パッケージの利用情報は開発途上の Docker イメージと学習データの Docker イメージに含まれるパッケージ数を合わせるため，bag-of-pkgs ベクトルのデータマスキングを行う．

表 5.1 に，開発進捗度 100%の場合の学習データの例を，表 5.2 に開発進捗度 50%の場合の学習データの例を示す．開発進捗度 100%の場合は，5.1.1 章で作成された bag-of-pkgs ベクトルをその

表 5.1: 開発進捗度 100%の学習データ
※ 100 %の場合，bag-of-pkgs ベクトルから予測モデルを生成

	pkg1	pkg2	pkg3	pkg4	計
img1	0	1	1	0	2
img2	1	1	1	1	4
img3	0	0	1	1	2

表 5.2: 開発進捗度 50%の学習データ
半分のパッケージの利用情報を 1 から 0 にマスキングする．(マスキング部分を青字で記載)

	pkg1	pkg2	pkg3	pkg4	計
img1	0	1	0	0	1
img2	0	1	0	1	2
img3	0	0	1	0	1

まま学習する．一方，開発進捗度 50%の場合は，100%の場合のパッケージ情報のうち 50%を（利用あり）から 0（利用なし）にする．本研究ではこの作業をマスキング処理と呼ぶ．マスキング対象のパッケージは，Docker イメージへのインストール順を考慮して選定する．Tern の出力結果は，パッケージがインストールされた順番に昇順で出力される．その情報を利用し，各 Docker イメージのパッケージ情報を降順（最も新しくインストールされたパッケージ）から順番に利用情報を 1 から 0 にマスキングする．

5.1.3 多層パーセプトロンによる予測モデルの構築

本手法の目的は，少量の OSS パッケージ情報から，将来的に GPL ライセンスと不整合となるライセンスのパッケージが導入されるかを予測する．5.1.2 章でマスキングされた bag-of-pkgs ベクトルから，機械学習による予測モデルを構築する．機械学習アルゴリズムには，多層パーセプトロン（MLP）[55] を利用する．MLP は単純なニューラルネットワーク（単純パーセプトロン）に中間層を設定し，3 つ以上の層にすることで線形分離不可な問題にも対応できるアルゴリズムである．ニューラルネットワークは，入力データを正しく分類するための適切な重みを最適化し学習する．これは入力データから最適な確率分布を点推定するのと同様であり，ライセンス非互換の原因となるパッケージとその依存関係にあるパッケージが利用される確率を推定することで予測できると考えられる．

MLP は入力層，中間層，出力層の 3 層で構成される．その概要を図 5.2 に示す．入力層は，説明変数 X_n を受け取る層を示す．W_{nm}，K_n はそれぞれの経路に設定された重みを示す．この重みは，出力層の目的変数 y に対する各説明変数 X_n の寄与率を調整する役割を示す．出力層はノードで算出された値の合計が閾値以上か否かを判定し予測結果を出力する．MLP の中で重要な中間層は，複数の入力と重みの計算結果を 1 つに抽象化する役割を果たす．多層パーセプトロンの抽象化関数には式 5.1 に示す ReLU 関数で行う．ReLU は入力層の各ノードからの重みの総和 x が 0 以上であれば各ノードの重みの総和を中間層の各ノードの値として採用する．他の活性化関数であるシグモイド関数やステップ関数は 0 から 1 の間の値に抽象化するのに対し，ReLU は重みの総和を直接ノードの値として採用し抽象化する．そのため，ライセンス互換性に関係するパッケージの重みを予測結果に強く反映させることができる．

$$y = max(0, W^T * X) \tag{5.1}$$

多層パーセプトロンで適切な予測を行うには各説明変数に対する重み W を適切に決定する必要がある．多層パーセプトロンの学習フェーズでは，この重みを自動的に決定する．まず，重みを

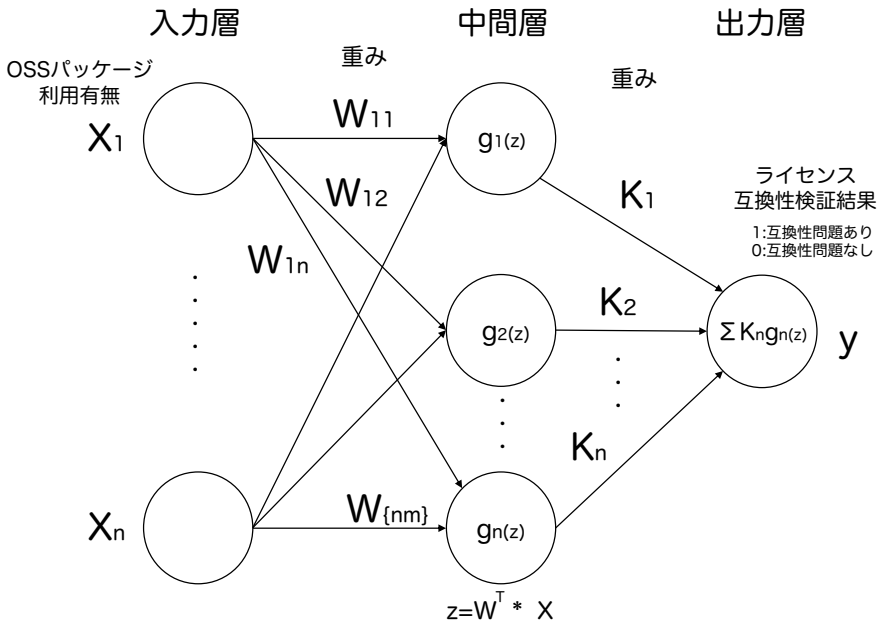


図 5.2: 多層パーセプトロン（MLP) の原理

表 5.3: 技術的課題 T2 とリサーチクエスチョンとの対応

	技術的課題	リサーチクエスチョン	
T2	開発早期におけるライセンス互換性検証結果の予測	RQ10	開発進捗度 100%のパッケージ情報を学習した場合，どれくらい早期からライセンス互換性を予測できるか
		RQ11	開発進捗度を考慮した学習により 開発早期での予測精度は向上するか

ランダムに設定し，学習データで予測を行い成否を評価する．出力層からでた出力値が正解と異なっていた場合、出力値と正解の誤差を元に重みの修正を行う．重みの修正は出力関数を各入力変数で微分し最急降下法で算出する．重みの修正は，まず，中間層と出力層との間の K_n で行われる．その結果を元に入力層と中間層の重みを修正する．この処理を重みの修正が行われなくなるまで繰り返すことで学習が終了する．

5.2 実験方法

本研究の目的は，Docker イメージ開発早期の情報で，将来的なライセンス互換性検証結果を予測することである．本章では，提案手法を評価するために行う評価実験について説明する．評価実験では表 5.3 に示す 2 つの RQ について取り組む.

5.2.1 データセット

本ケーススタディでは，3.2 章の予備調査で用いた Docker イメージを利用する．評価実験に用いるデータセットを表 5.4 に示す．データセットの前処理として，ベース OS のみのイメージに対

表 5.4: 予測実験で用いるデータセット

	GPL 非互換が含まれている	GPL 非互換を含まない	合計
イメージ数	457（77%）	141（23%）	598

してソフトウェアパッケージが追加でインストールされていない Docker イメージ（環境設定変更のみ等）は，非互換の組み合わせが含まれていないため評価実験のデータセットからレイヤー数が 2 以下の Docker イメージを除外している．最終的に評価実験で扱うのは 598 イメージとする．

598 イメージのうち，GPL 非互換が含まれていたイメージは 457（77%），GPL 非互換が含まれていないイメージは 141（23%）であった．

また，データセットを作成する過程で，パッケージ名が同一でも，インストールされている Docker イメージやバージョンによってライセンスが異なるケースが存在した．誤りの場合の可能性もあるが，OSS ライセンスは著作者によってしか変更できないため，それぞれ別の独立したパッケージとして扱うこととした．その結果，2,376 種類のパッケージ名を取得することができた．また，1 コンテナにインストールされていた平均パッケージ数は約 92，中央値は約 78 となっていた．

5.2.2 機械学習アルゴリズムの評価方法

本評価実験では，提案手法の比較評価を行うため，MLP に加え次の機械学習アルゴリズムを設定する．Random Forest [56]，Support Vector Machine (SVM) [57]，Naive Bayes [58]，Gradient Boosting(GBR) [59] を用いる提案手法である MLP とこれらの機械学習アルゴリズムを用いて，各イメージを 0 もしくは 1（1：GPL 非互換，0：GPL 互換）の 2 値分類する．同時にデータセットとなる Docker イメージは全て GPL の検証結果をラベリングしている．

Docker イメージの互換性検証結果の予測は交差検証で実施する．図 5.3 に実験の概要を示す．データセットとなる 598 の Docker イメージのパッケージ情報のうち，70%を学習データ，残りの 30%をテストデータとする．また，本実験では，開発進捗度ごとに予測精度の評価を行うため，各パッケージ利用情報を部分的にマスキングしながら行う．マスキングのレートは学習データ，テストデータと同じ値とし，0%から 90%まで，5%ずつ増分（0%, 5%, 10%, 15% … ）させながら計 19 点の間隔で予測を行う．予測精度を測定するメトリクスは，適合率（式 5.3），再現率（式 5.4），F 値（式 5.5）を用いる．また，これらの機械学習アルゴリズムは，実行の度に結果が異なるため，各マスキングレートで図 5.3 の実験を 100 回ずつ行い，その平均値をそのマスキングレートの代表値として比較する．

5.3 実験結果

5.3.1 RQ10：開発進捗度 100%のパッケージ情報を学習した場合，どれくらい開発早期からライセンス互換性を予測できるか

動機： Docker イメージ内のパッケージのライセンス互換性検証予測は，できるだけ開発早期に実施するのが望ましいが，早期ではパッケージ情報が不完全なため，高い精度で予測することは単純には難しいと想定される．しかしながら，どれくらい Docker イメージが開発されている必要

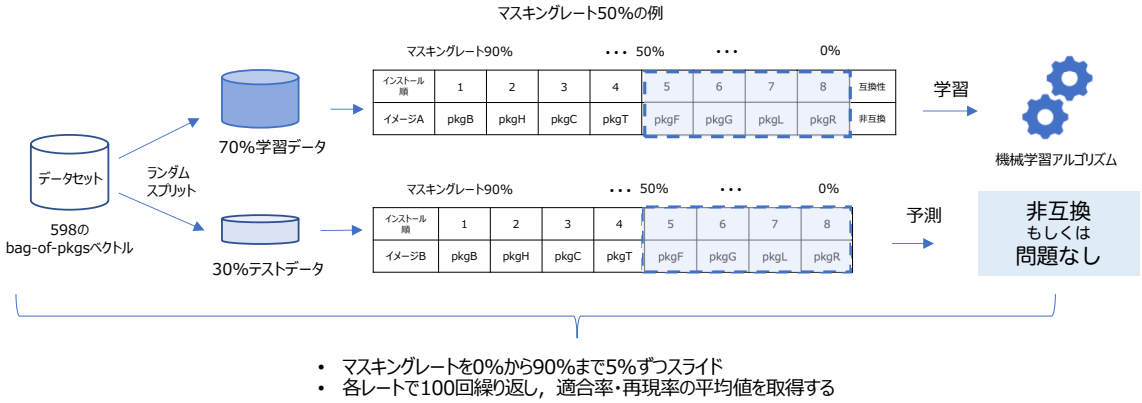


図 5.3: ライセンス互換性検証予測の実験概要

があるかは定量的には不明である．本 RQ では，まずは，開発進捗度 100%の Docker イメージのパッケージ情報を学習し，開発進捗度 10%〜100%の Docker イメージに対し，ライセンス互換性検証結果の予測を行う．その後，開発進捗度ごとの予測精度の変化を確認することで，正しく予測するのに最低 Docker イメージがどの程度開発されている必要があるか検証する．

アプローチ： Docker Hub 上の 598 のイメージを 70%を学習データ，残りの 30%をテストデータにランダムに分割し，そのうち，学習データとそのライセンス互換性検証結果を多層パーセプトロン（MLP）[55] で学習する．

本 RQ では，開発進捗度 100%の Docker イメージを学習するため，イメージに含まれている全てのパッケージ情報を学習する．多層パーセプトロンの中間層数は 1 とし，活性化関数は 5.1.3 章で説明したように ReLU を採用する．中間層のニューロン数は，データセットのパッケージに設定されていたライセンスは全 107 種類であったことを参考に 100 と設定した．ただし，最適なニューロン数を調査するため，ニューロン数を 50 と 150 に設定したパターンを追加し実験を行う．

テストデータの開発進捗度は 10%〜100%まで 5%刻みで設定する．5.1.2 章で説明したように，各開発進捗度に対応するテストデータとなるよう，パッケージ情報をマスキングする．マスキングは，開発進捗度に応じて，各パッケージの利用情報が 1（利用あり）から 0（利用なし）に置換されるが，このとき，0 に置換すると，今後パッケージが導入されないという情報を与えることになり，学習データとの整合性が取れない．そのため，本 RQ に限っては，各開発進捗度に応じてパッケージ情報を 0 にマスキングするのではなく，以下の式 5.2 で計算される期待値でマスキングする．データセットを調査したところ，1 イメージあたりに利用されているパッケージ数の中央値は 78 であった．利用パッケージ数の中央値をパッケージの種類数 2,376 で除算することで 1 パッケージあたりの期待値を算出できる．ただし，実際は 1 イメージあたりのパッケージ数の中央値は開発進捗度によって異なるため，さらに開発進捗度を乗算して調整する．

$$\text{マスキング値} = \frac{\text{1 イメージあたりの利用パッケージ数の中央値} * \text{各開発進捗度}}{\text{パッケージの全種類数}} \tag{5.2}$$

$$\text{適合率} = \frac{\text{予測して正解した } \textit{Docker} \text{ イメージ数 } (TP)}{\text{予測して正解した } \textit{Docker} \text{ イメージ数 } (TP) + \text{予測して間違った } \textit{Docker} \text{ イメージ数 } (FP)} \tag{5.3}$$

$$\text{再現率} = \frac{\text{予測して正解した } Docker \text{ イメージ数 } (TP)}{\text{予測して正解した } Docker \text{ イメージ数 } (TP) + \text{本来予測すべきであった } Docker \text{ イメージ数 } (FN)} \tag{5.4}$$

$$F \text{ 値} = \frac{2 * \text{適合率} * \text{再現率}}{\text{適合率} + \text{再現率}} \tag{5.5}$$

予測結果は適合率（式 5.3），再現率（式 5.4），F 値（式 5.5）で評価する．ライセンス互換性検証には，適合率・再現率の両方重要であると考えている．そのため，目標とする F 値は 90%以上と十分に高い値を設定する．ライセンスコンプライアンスの分野は，非常にセンシティブであり，ライセンス特定ツール Ninka [14] の F 値は約 89%となっている．テストデータの開発進捗度が高くなればなるほど，予測精度は向上すると考えられるため，F 値が 90%に到達するタイミングを予測可能な開発進捗度と定義し調査する．

結果： 開発進捗度 10%〜100%まで，MLP の各ニューロン数で予測結果を評価したものを，表 5.5 に適合率，表 5.6 に再現率，表 5.7 に F 値を示す．各表の結果は，小数点第三位を四捨五入し掲載している．また，それぞれをグラフ化したものを図 5.4 に示す．図 5.4 から，開発進捗度が上がるにつれ，再現率，適合率，F 値が上昇することがわかった．

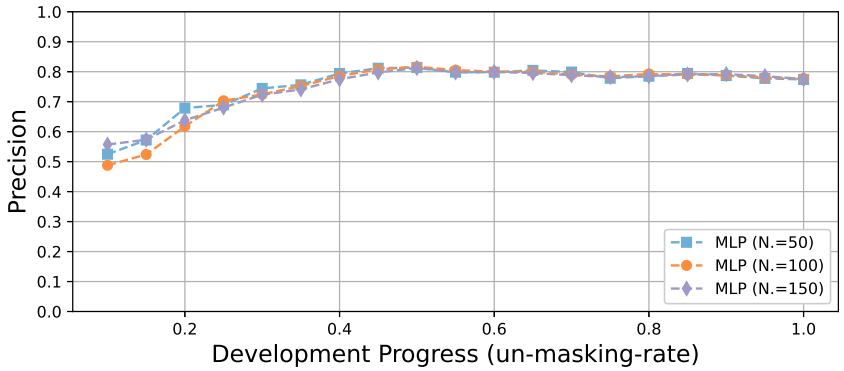
開発進捗度 10% 50%に着目すると，適合率は 50%台 80%台に上昇しているのに対して，再現率は数%の状態から〜75%へ急激に上昇している．開発進捗度 10%では，再現率が数%なため，ほとんどのイメージに対しライセンス互換性に問題がないと予測している状態である．これは，開発進捗度 10%では，1 イメージあたり約 8 個（100%時は 1 イメージあたり約 78 個）ほどのパッケージしかインストールされていないためである．しかしながら，開発進捗度が上がるにつれ，イメージ内のパッケージ数が増加し，ライセンス互換性に問題のある Docker イメージに対しても予測できるようになっていることが分かる．

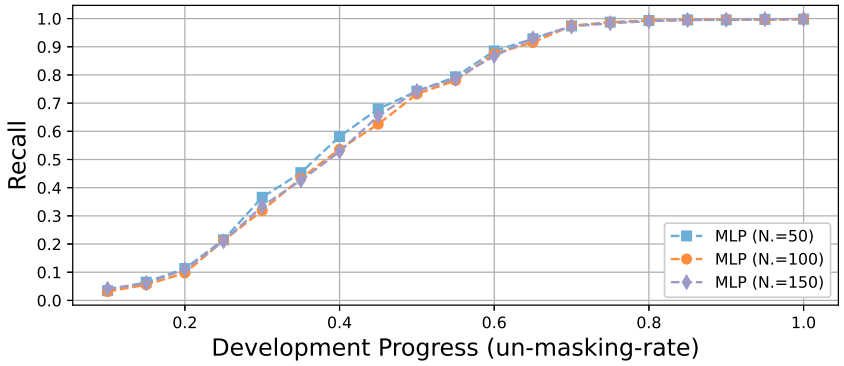
開発進捗度 50%以降に着目すると，再現率は 75%から 100%に上昇しているが，適合率は約 80%でほぼ横ばいであり，全体的に上昇率が鈍化している．これはイメージ内で使用されるパッケージ量が予測するのに十分な量に達していると考えられる．

F 値に着目すると，開発進捗率がどの割合でも目標の 90%に到達しなかった．F 値の最大値の到達地点は各ニューロン数で同等であり，開発進捗率が 80%に到達した時点で F 値が 88%となる．開発進捗率が 80%は開発後期であり，単純に開発進捗度 100%のパッケージ情報を学習するだけでは開発早期での予測は難しいことが分かった．また，ニューロン数での F 値の平均はニューロン数が 50 の場合は 0.66，100 の場合は 0.65，150 の場合は 0.65 であり，50 の場合の方が 1%ほど高くなっているが小数点第三位を四捨五入しているため，誤差の範囲内といえることが分かった．ニューロン数が 50 以上であれば予測結果に大きな影響はないと考えられる．以上の結果から，RQ10 は以下のように回答する．

RQ10 への回答:

開発進捗度 100%の Docker イメージのパッケージ情報を学習し，MLP を用いてライセンス互換性検証結果を予測した結果，開発進捗度が上昇するに従い予測精度が向上することが分かった．しかしながら，ベースラインとなる F 値 90%には到達せず，最大値は開発進捗度 80%の時点で F 値が 88%であった．





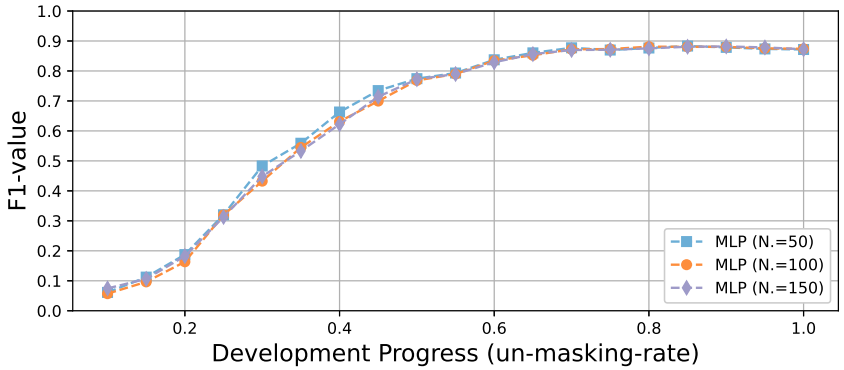


図 5.4: 開発進捗度 100%のパッケージ情報を学習した場合の予測結果

表 5.5: 開発進捗度 100%のパッケージ情報を学習した場合の適合率（N. はニューロン数を示す.）

開発進捗度	MLP(N.=50)	MLP(N.=100)	MLP(N.=150)
0.10	0.52	0.49	0.56
0.15	0.57	0.52	0.57
0.20	0.68	0.62	0.64
0.25	0.69	0.70	0.68
0.30	0.74	0.72	0.72
0.35	0.76	0.75	0.74
0.40	0.79	0.79	0.77
0.45	0.81	0.81	0.80
0.50	0.81	0.82	0.81
0.55	0.80	0.81	0.80
0.60	0.80	0.80	0.80
0.65	0.80	0.80	0.80
0.70	0.80	0.79	0.79
0.75	0.78	0.78	0.78
0.80	0.79	0.79	0.78
0.85	0.79	0.79	0.79
0.90	0.79	0.79	0.79
0.95	0.78	0.78	0.79
1.00	0.77	0.78	0.78
平均	0.75	0.74	0.75

表 5.6: 開発進捗度 100%のパッケージ情報を学習した場合の再現率（N. はニューロン数を示す.）

開発進捗度	MLP(N.=50)	MLP(N.=100)	MLP(N.=150)
0.10	0.03	0.03	0.04
0.15	0.06	0.06	0.06
0.20	0.11	0.10	0.11
0.25	0.22	0.21	0.21
0.30	0.37	0.32	0.33
0.35	0.45	0.43	0.43
0.40	0.58	0.54	0.53
0.45	0.68	0.63	0.65
0.50	0.74	0.73	0.74
0.55	0.79	0.78	0.79
0.60	0.89	0.88	0.87
0.65	0.93	0.92	0.93
0.70	0.97	0.97	0.97
0.75	0.99	0.99	0.98
0.80	0.99	0.99	0.99
0.85	1.00	1.00	0.99
0.90	1.00	1.00	1.00
0.95	1.00	1.00	1.00
1.00	1.00	1.00	1.00
平均	0.67	0.66	0.66

5.3.2 RQ11：開発進捗度を考慮したパッケージ情報の学習により開発早期での予測精度は向上するか

動機： RQ10 では，開発進捗度 100%のパッケージ情報を MLP で学習し，ライセンス互換性検証結果の予測を行った．しかしながら，開発進捗度が 80%で最大 F 値が 88%であり，開発早期の予測は行うことができなかった．これは，開発進捗度 100%では 1 コンテナあたり，約 78 個のパッケージが含まれてるのに対し，開発進捗度 10%では約 8 個ほどであり，このパッケージ差異により開発早期での予測が行えないと考えられる．そこで，本 RQ11 では，学習データをテストデータと同一のパッケージ数（同一の開発進捗度）にした場合の予測精度を評価し，機械学習アルゴリズムでも開発早期に予測が行えるようになるかを確認する．

アプローチ： RQ11 では，データセットを学習データをテストデータに分け，10%〜100%まで 5%ごとの開発進捗度に応じてマスキングする．次に，MLP で予測モデルを構築し学習データとテストデータの開発進捗度を同一しながら予測する．このとき MLP のニューロン数は 100 に設定する．RQ10 では，ニューロン数を 50,100,150 に設定し実験を行ったが，3 パターンとも結果は同様であったため，ニューロン数が 50 以上であれば予測結果に大きな影響はないと想定される．最後にテストデータと同じ開発進捗度の予測モデルでライセンス互換性検証予測を行う．例えば，テストデータを開発進捗度 10%にマスキングしていた場合，開発進捗度 10%の学習データから作成された予測モデルを用いる．

また，参考のため MLP 以外の機械学習アルゴリズムも用いる．MLP と合わせて検証するのは，Random Forest [56]（RF），Support Vector Machine [57]（SVM），Naive Bayes [58]（NB），Gradient Boosting [59]（GB）らの機械学習アルゴリズムで学習する．実装には，Python の機械学習ライブラリである scikit-learn[1]を，パラメータは初期値で利用した．

MLP およびその他の機械学習での予測精度を各アプローチの予測結果を適合率（式 5.3），再

[1] https://scikit-learn.org/stable/

表 5.7: 開発進捗度 100%のパッケージ情報を学習した場合の F 値
（N. はニューロン数を示す.）

開発進捗度	MLP(N.=50)	MLP(N.=100)	MLP(N.=150)
0.10	0.06	0.06	0.07
0.15	0.11	0.10	0.11
0.20	0.19	0.16	0.18
0.25	0.32	0.32	0.31
0.30	0.48	0.43	0.45
0.35	0.56	0.54	0.53
0.40	0.66	0.63	0.62
0.45	0.73	0.70	0.71
0.50	0.77	0.77	0.77
0.55	0.79	0.79	0.79
0.60	0.84	0.83	0.83
0.65	0.86	0.85	0.86
0.70	0.88	0.87	0.87
0.75	0.87	0.87	0.87
0.80	0.88	0.88	0.88
0.85	0.88	0.88	0.88
0.90	0.88	0.88	0.88
0.95	0.87	0.88	0.88
1.00	0.87	0.87	0.87
平均	0.66	0.65	0.65

表 5.8: 開発進捗度ごとにパッケージ情報を学習した場合の適合率
（N. はニューロン数を示す.）

開発進捗	MLP(N.=100)	RF	SVM	NB	GB
0.10	**0.94**	**0.94**	0.93	0.91	**0.94**
0.15	0.94	**0.95**	**0.95**	0.92	0.94
0.20	**0.96**	**0.96**	**0.96**	0.91	**0.96**
0.25	0.96	**0.97**	0.95	0.92	0.96
0.30	0.95	**0.96**	**0.96**	0.92	**0.96**
0.35	0.95	0.96	**0.97**	0.92	0.96
0.40	0.96	0.97	**0.97**	0.91	0.96
0.45	0.95	**0.97**	0.96	0.92	0.95
0.50	**0.97**	0.96	0.95	0.92	0.96
0.55	0.96	**0.97**	0.95	0.92	0.96
0.60	**0.96**	**0.96**	0.95	0.92	**0.96**
0.65	**0.96**	**0.96**	**0.96**	0.91	**0.96**
0.70	**0.97**	**0.97**	0.96	0.92	0.96
0.75	**0.97**	**0.97**	0.95	0.92	0.96
0.80	**0.98**	**0.98**	0.97	0.92	0.96
0.85	**0.97**	**0.97**	0.96	0.91	0.95
0.90	**0.98**	**0.98**	0.95	0.92	0.97
0.95	**0.98**	**0.98**	0.95	0.92	0.97
1.00	**0.99**	**0.99**	0.96	0.92	0.98
平均	0.96	**0.97**	0.96	0.92	0.96

現率（式 5.4），F 値（式 5.5）で評価し，各開発進捗度での精度の推移を確認する．

結果：

開発進捗度 10%〜100%まで，MLP とその他のアプローチの予測精度を表 5.8 に適合率，表 5.9 に再現率，表 5.10 に F 値を示す．各表の結果は，小数点第三位を四捨五入し掲載している．また，それぞれをグラフ化したものを図 5.5 に示す.

図 5.5 から，学習データとテストデータの開発進捗度を同一にして予測すると，開発進捗度が 10%の時点で，高い適合率，再現率，F 値で予測できていることがわかる．開発進捗度が 10%の場合の適合率は，表 5.8 に示すように，MLP，Random Forest，Gradient Boosting が最も高く 94%となっている．開発進捗度が 10%の場合の再現率は，表 5.9 に示すように，MLP，Random Forest，Gradient Boosting で最も高く 96%となっている．開発進捗度の場合 10%の場合の F 値は，表 5.10 に示すように，RandomForest が最も高くなっており，96%である．提案手法の MLP の F 値は最大とはならなかったが，95%と十分に高い結果となっている．また，開発進捗度全体を通して F 値の平均値が最も高かったのは MLP であった．ただし，その他の機械学習モデルも F 値で 90%を超えており，どの開発進捗度でも十分にライセンス検証結果予測ができることがわかった．以上の結果から RQ11 の回答は以下とする．

RQ11 への回答:

学習データとテストデータの開発進捗度を同一にして予測することで，開発進捗度 10%時点でも機械学習アルゴリズムで適合率，再現率，F 値で 90%を達成できており，より早期のライセンス互換性検証結果を予測できることが分かった．また，全体で最も F 値の平均が高かったのは MLP であることが分かった.

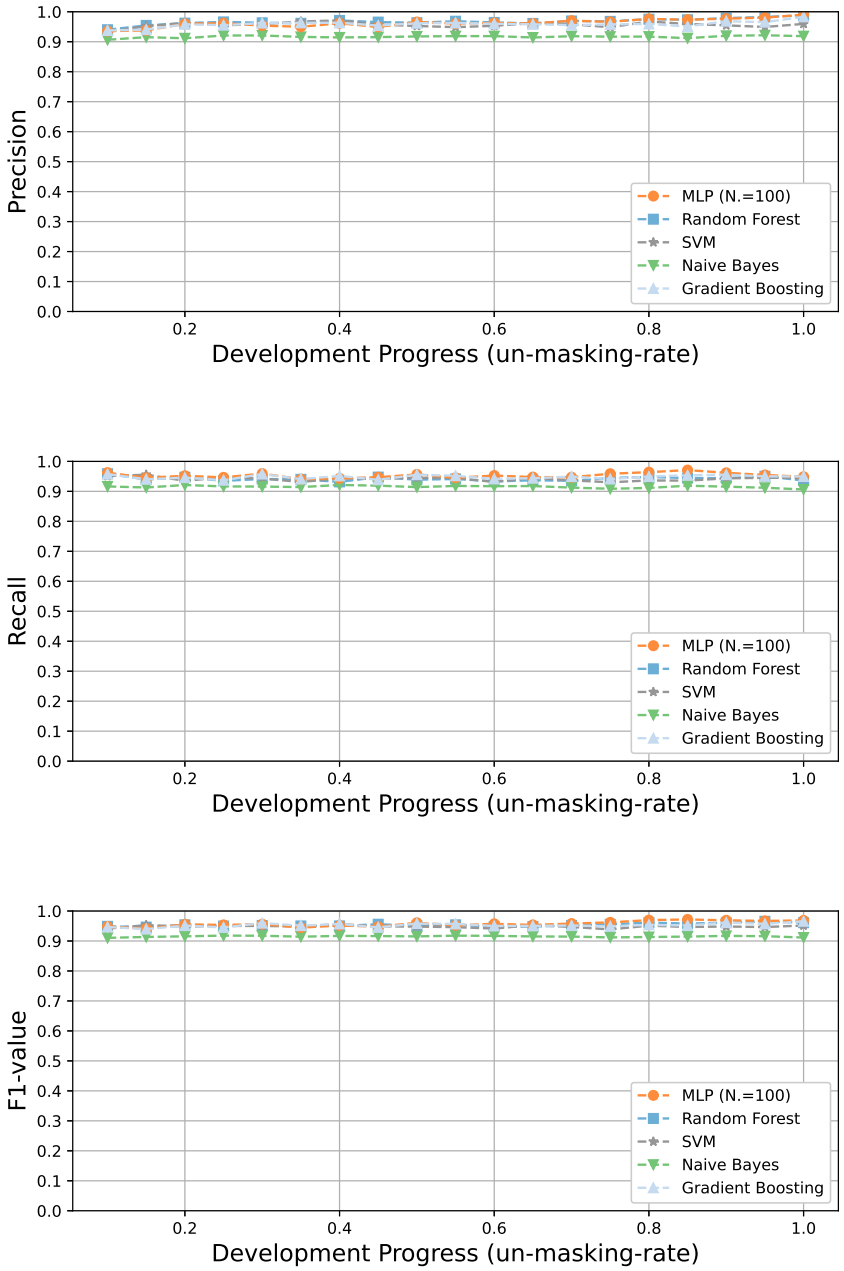


図 5.5: パッケージ情報を各開発進捗度ごとに学習した場合の予測結果

表 5.9: 開発進捗度ごとにパッケージ情報を学習した場合の再現率
（N. はニューロン数を示す.）

開発進捗度	MLP(N.=100)	RF	SVM	NB	GB
0.10	**0.96**	**0.96**	0.95	0.92	**0.96**
0.15	0.95	0.94	**0.96**	0.91	0.94
0.20	**0.95**	**0.95**	0.94	0.92	**0.95**
0.25	**0.95**	0.94	**0.95**	0.92	0.94
0.30	**0.96**	0.94	0.94	0.92	**0.96**
0.35	**0.94**	**0.94**	0.93	0.91	**0.94**
0.40	0.94	0.93	0.94	0.92	**0.95**
0.45	**0.95**	**0.95**	0.94	0.92	0.94
0.50	**0.96**	0.94	0.94	0.91	**0.96**
0.55	**0.95**	0.94	0.94	0.92	**0.95**
0.60	**0.95**	0.93	0.93	0.92	0.94
0.65	**0.95**	0.94	**0.95**	0.92	0.94
0.70	**0.95**	0.94	0.94	0.91	**0.95**
0.75	**0.96**	0.94	0.93	0.91	0.94
0.80	**0.96**	0.95	0.94	0.91	0.95
0.85	**0.97**	0.94	0.94	0.92	0.95
0.90	**0.96**	0.94	0.94	0.92	**0.96**
0.95	**0.95**	**0.95**	**0.95**	0.91	**0.95**
1.00	**0.95**	0.94	0.94	0.91	**0.95**
平均	**0.95**	0.94	0.94	0.92	**0.95**

表 5.10: 開発進捗度ごとにパッケージ情報を学習した場合の F 値
（N. はニューロン数を示す.）

開発進捗度	MLP(N.=100)	RF	SVM	NB	GB
0.10	0.95	**0.96**	0.95	0.91	0.95
0.15	0.94	0.94	**0.96**	0.91	0.94
0.20	**0.96**	0.95	0.94	0.92	0.95
0.25	**0.95**	0.94	**0.95**	0.92	**0.95**
0.30	**0.96**	0.94	0.94	0.92	**0.96**
0.35	0.94	0.94	0.93	0.91	**0.95**
0.40	0.95	0.93	0.94	0.92	**0.96**
0.45	**0.95**	**0.95**	0.94	0.92	**0.95**
0.50	**0.96**	0.94	0.94	0.92	**0.96**
0.55	**0.95**	0.94	0.94	0.92	**0.96**
0.60	**0.96**	0.93	0.93	0.92	0.95
0.65	**0.95**	0.94	**0.95**	0.92	**0.95**
0.70	**0.96**	0.94	0.94	0.91	**0.95**
0.75	**0.96**	0.94	0.93	0.91	0.95
0.80	**0.97**	0.95	0.94	0.91	0.95
0.85	**0.97**	0.94	0.94	0.91	0.95
0.90	**0.97**	0.94	0.94	0.92	0.96
0.95	**0.97**	0.95	0.95	0.92	0.96
1.00	**0.97**	0.94	0.94	0.91	**0.97**
平均	**0.96**	0.94	0.94	0.92	0.95

5.4 考察

5.4.1 GPL 非互換予測精度と開発進捗度の関係

RQ10，RQ11 では，Docker イメージに含まれる OSS パッケージ情報を学習し，開発進捗度ごとにライセンス互換性検証結果の予測を行った．RQ10 では，開発進捗度 100%のイメージに含まれるパッケージ情報を学習し，10%〜100%の開発進捗度のイメージに対して予測した結果，開発進捗度 80%で F 値が 88%に到達することを確認した．一方，RQ11 では，学習データもテストデータと開発進捗度を同一にしたところ，全ての開発進捗度で F 値が 90%以上となった．

各開発進捗度ごとに学習したほうが予測精度が向上した理由を明らかにするため，非互換が発生している Docker イメージで初めてライセンス非互換が発生したレイヤーの割合を調査した．その結果を図 5.6 に示す．非互換が発生したレイヤーの割合とは，何番目のレイヤで非互換が発生したかを調査し，その Docker イメージの総レイヤー数で正規化したものである．レイヤーの割合が低いと開発初期にインストールされ，高いと開発終盤にインストールされたことを表しており，開発進捗度に相当する指標となる．

図 5.6 から，Docker イメージの非互換の発生レイヤは中央値 0.60 を中心に満遍なく非互換が発生していることが分かる．このことから，非互換が発生しているレイヤはコンテナによってバラバラであるといえる．また，図 5.7 に各パッケージ別のライセンス非互換が発生したレイヤーの分布を gnu 系パッケージのみ抜粋して示す．図 5.7 中央の libgsasl パッケージや，gccmultilib パッケージのように全レイヤーにわたって非互換が発生しているパッケージもあれば， 5.7 上の gpg 関連パッケージのようにライセンスレイヤーの範囲に傾向があるパッケージもある．そのため，各開発進捗度に合わせてパッケージ情報を学習した方が，コンテナのパッケージの特性が予測モデルに反映され，機械学習による予測精度が向上したと考えられる．

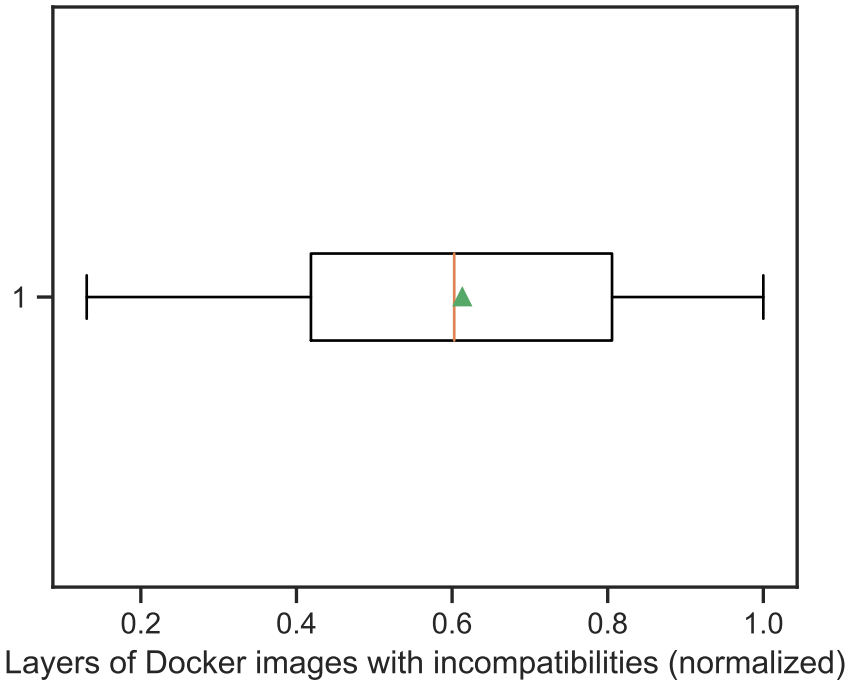


図 5.6: ライセンス非互換が発生したレイヤー
▲は平均値を表す

5.4.2 最適な予測モデル

RQ10 では，開発進捗度 100%の Docker イメージのパッケージ情報を学習し予測を行った．RQ11 では，Docker イメージのパッケージ情報を各開発進捗度に合わせて学習し，さらに比較のため，Random Forest，SVM，Naive Bayes，Gradient Boosting でも合わせて実験を行った．提案手法の MLP に着目すると，RQ10 では F 値が最大となったのは開発進捗度 80%の時であり，開発後期でしか予測が行えないが，RQ11 では，開発進捗度 10%の時点で既に F 値は 95%であり，開発早期での予測が可能であることが分かった．

ただし，RQ11 での実験結果では MLP 以外の機械学習アルゴリズムでも，高い精度で予測可能であることが分かった．MLP は他のアルゴリズムより，全体の F 値の平均値は 1%程度高くなっているが，他のアルゴリズムも F 値が全て 90%以上と実用に十分な性能がでている．

実際に本手法の運用を想定した場合，予測精度以外の点において MLP にはデメリットが存在する．MLP のデメリットとして，非互換の原因となるパッケージを出力することができないことが挙げられる．ライセンス互換性に問題があると予測された場合に，どのパッケージを変更すればいよか検討することができない．非互換の原因となるパッケージ名を合わせて出力するという観点では，Random Forest 等の決定木アルゴリズムはパッケージ情報ごとの重要度を出力することができる．Random Forest における重要度とは，各説明変数がクラスの分類にどの程度寄与しているのかを示す指標である．表 5.11 に開発進捗度 100%の場合の Random Forest の重要度上位パッケージ 10 種類とそのインストール率を示す．最も寄与したパッケージは libsasl (BSD4)，libsaslmodulesdb (BSD4) であり，全 Docker イメージのうち libsasl は 66%，libsaslmodulesdb は

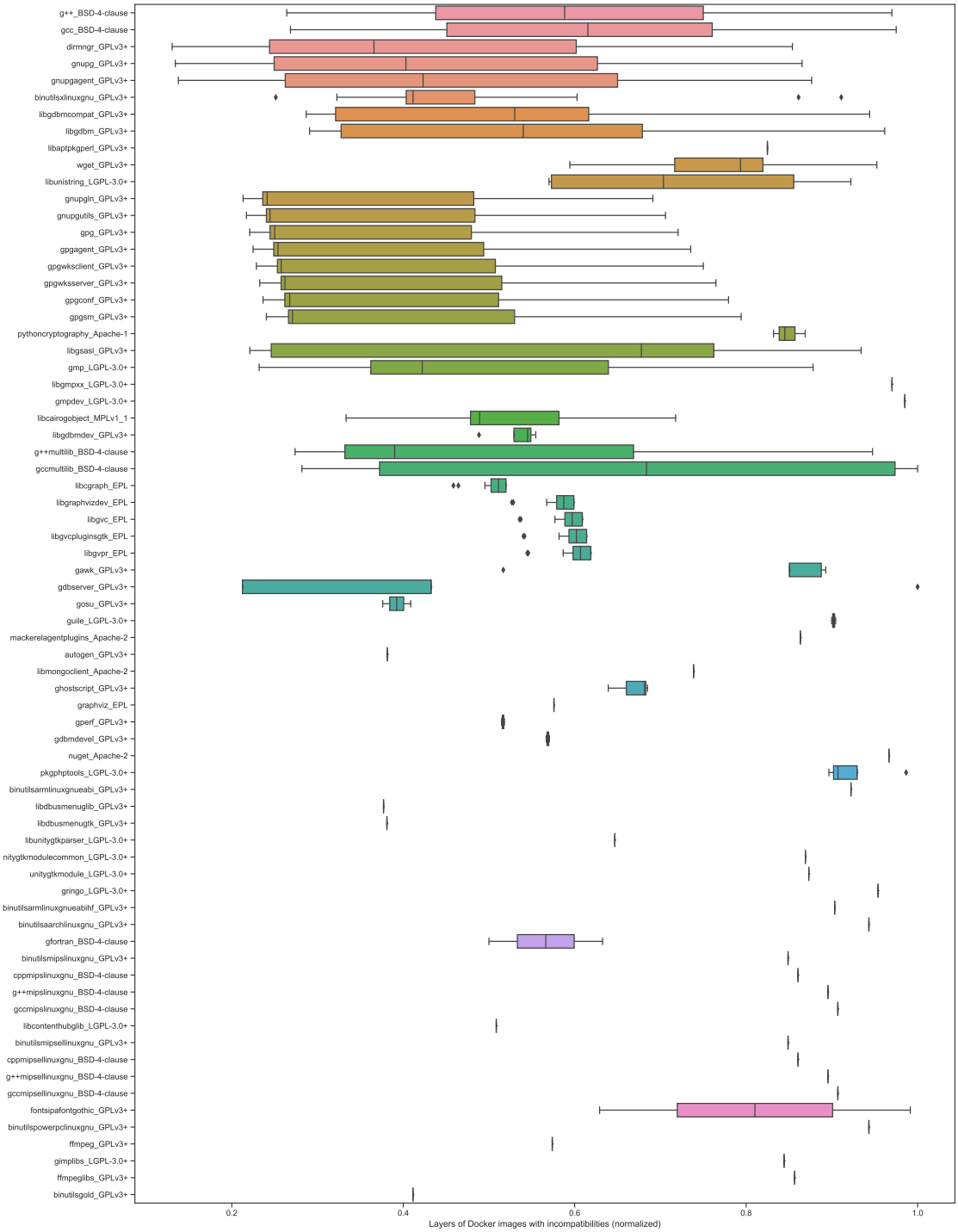


図 5.7: パッケージ別ライセンス非互換が発生したレイヤーの分布（GNU 系パッケージを中心に抜粋）

表 5.11: Random Forest 重要度上位 10 位のパッケージとイメージ数

	パッケージ名 (ライセンス名)	イメージ数	インストール率	重要度
1	libsasl (BSD4)	393	0.66	0.04
2	libsaslmodulesdb (BSD4)	387	0.65	0.038
3	libidn (GPLv3+)	423	0.71	0.027
4	libpkit (BSD3)	430	0.72	0.023
5	libnettle (LGPL-2+)	373	0.62	0.014
6	lsbbase (GPLv2)	450	0.75	0.013
7	libhogweed (LGPL-2+)	367	0.61	0.012
8	cacertificates (GPLv2+)	402	0.67	0.012
9	libpcre (BSD3)	451	0.75	0.012
10	adduser (GPLv2+)	450	0.75	0.011

65%のインストール率と比較的多く利用されているパッケージであった．この結果は，ライセンスの互換性に問題があると予測された場合，どのパッケージについて変更を検討すればよいか開発者に示すことができる．そのため，予測精度という観点では MLP が最も望ましいが，実際の運用を想定した場合，要因となるパッケージの分析も同時に行える Random Forest が有用な場面も出てくるといえる．

5.4.3 GPL 非互換となるパッケージのインストール経路

3.2 章での Docker Hub のイメージに関する予備調査では，ライセンス非互換の最も多い組み合わせは GPL と BSD4（表 3.8）であることを明らかにした．また，RQ11 では，実際に機械学習アルゴリズムで開発早期でのライセンス互換性検証予測が行えることを示した．さらに，表 5.11 から，Random Forest の分類に最も寄与したパッケージは，libsasl (BSD4)，libsaslmodulesdb (BSD4) であり，全 Docker イメージのうち libsasl は 66%，libsaslmodulesdb は 65%のインストール率であることが分かった．そこで，このパッケージのインストール経路について調査したところ，Docker のオフィシャルイメージに共通したインストールパターンを発見した．その例を Dockerfile のコードベースで説明する．

Listing 5.1: buildpack-deps:buster を継承しているイメージ

```
1 FROM buildpack-deps:buster
2
3 ENV PATH /usr/local/bin:$PATH...
```

この Dockerfile では，buildpack-deps:buster を継承しており，その後，ENV でベース OS 内の環境設定を行なっている．この Dockerfile だけでは，Docker イメージにライセンス非互換の問題が発生するパッケージがインストールされているかどうかは不明である．そのため，buildpack-deps:buster イメージにどのようなソフトウェアパッケージが存在するかを調べる必要がある．buildpack-deps:buster イメージをビルドするために作成された Dockerfile は次のようになっている．

Listing 5.2: buildpack-deps:buster イメージの Dockerfile

```
1 FROM buildpack-deps:buster-scm
2
3 RUN set -ex; \
4       apt-get update; \
5       apt-get install -y --no-install-recommends \...
```

この Dockerfile では，buildpack-deps:buster-scm を継承しており，その後，RUN ではビルドツールなどをインストールしている．ここでも，基本的には GPL でライセンスされたモジュールが導入されているだけである．buildpack-deps:buster-scm イメージをビルドするために作成された Dockerfile は以下のようになっている．

Listing 5.3: buildpack-deps:buster-scm イメージの Dockerfile

```
1 FROM buildpack-deps:buster-curl
2
3 RUN apt-get update && apt-get install -y
                --no-install-recommends \
4       git \
5       mercurial \...
```

buildpack-deps:buster-scm では，ソースコントロールシステムを導入し，バージョン管理機能を提供している．buildpack-deps:buster-curl 継承しており，その後，RUN コマンドではビルドツールなどをインストールしている．buildpack-deps:buster-curl イメージをビルドするために作成された Dockerfile は以下のようになっている．

Listing 5.4: buildpack-deps:buster-curl イメージの Dockerfile

```
1 FROM debian:buster
2
3 RUN apt-get update && apt-get install -y
                --no-install-recommends \
4       ca-certificates \
5       curl \
6       netbase \
7       wget \
...
```

buildpack-deps:buster-curl では，ベース OS イメージである debian:buster を FROM コマンドで呼び出している．ここで，apt-get install により，curl に依存関係のあるモジュール（libsasl や libsaslmodulesdb など）を自動的にインストールされることが確認できる．ここで，ca-certificates, netbase, wget パッケージは GPL であり，curl と依存関係のある libsasl や libsaslmodulesdb は BSD4 のため，GPL 非互換が発生することが分かる．

buildpacks[2]は，Java などのプログラム言語環境が動作するのに必要なパッケージがインストールされており，Docker イメージの共通部品としての利用を想定している．Docker イメージの開発者は，このイメージを FROM コマンドで継承しておくことで，これらのパッケージのバージョンアップ等の保守をカプセル化することができる．しかしながら，実際はライセンス非互換が発生する原因の 1 つとなっており，かつイメージの中のパッケージも確認しないことから，この GPL

[2] https://registry.hub.docker.com/_/buildpack-deps/

非互換に開発者の意識下のみで気づくのは難しいといえる．このようなライセンス非互換を回避するためにも本手法は役立てられると考えている．

5.4.4 制約

内的妥当性への脅威

内部妥当性の脅威は、研究の内部要因に起因した結果に影響を与える可能性のある脅威を指す．本研究では、結果をシンプルで分かりやすくするために，複数のライセンスを持つパッケージを除外している．そのため，本研究結果は複数のライセンスを持つパッケージを含むデータセットでは，結果が異なる可能性がある．また，本研究結果は，Tern と Ninka に大きく依存している．両ツールは高性能であるため、誤検出は少ないと考えられる．

外的妥当性への脅威

外部妥当性の脅威は、実験結果の普遍性に関わる脅威を指す．本研究では、GitHub の上の Dockerfile をビルドし，from 文による継承等により Docker Hub から Docker イメージを入手している．ただし，このプロセスはほとんどの Docker イメージを有する環境で行われており、その違いが結果に与える影響は軽微であると想定している．また，今回は，最も普及している Docker イメージを調査したが、その数は限定的である．Docker Hub には 800 万以上の Docker イメージが登録されているため、今回の分析結果は Docker Hub を代表するものではない可能性がある．

5.5 関連研究

近年急速にコンテナ技術を取り入れたシステム開発が広まりつつある．そのため，Docker イメージの開発で発生する課題・問題を明らかにする研究が多く行われている．本章では，Docker イメージに関する研究について説明し，本研究で取り組んでいるコンテナ開発で発生する OSS 法的リスクへの問題がどのような位置づけとなるかを明らかにする．

5.5.1 Docker イメージの開発

ソフトウェア工学の分野では，Docker イメージはどのように開発しているのかを明らかにする研究が行われている．Haque ら [60] は，開発者が抱えているコンテナシステムの開発や運用での課題を把握するため，Docker に関する StackOverflow の QA を LDA で分類し，13 のカテゴリにまとめた．中でも最も多くのカテゴリを占めていたのは Web アプリケーションのデプロイに関する質問であることを明らかにした．コンテナのパッケージの脆弱性に関する分析・対応方法に関する研究も行われている．Zerouali ら [61] は，コンテナのバージョンに着目してコンテナ内のパッケージがどの程度古くなっているか，また、それがバグや重大な脆弱性の存在とどのように関連しているか，Debian のディストリビューションを元に作成された 7,380 件のコンテナを対象として分析した．その結果，Stable な古いパッケージであっても，ほとんどのコンテナパッケージが Debian には最新の修正版が確認できた．また，脆弱性の数とバグの数には相関があり，Debian は

脆弱性を優先して対応するため，Docker イメージには未解決のバグが多く存在することが分かった．安定性とセキュリティを重視するコンテナ配備者は，より優れた更新手順を確立する必要があることを示している．Zerouali ら [62] は，Docker イメージの npm パッケージ（Javascript のパッケージ管理システム）の脆弱性について依存関係の観点から分析をおこなった．古い npm パッケージが存在すると、潜在的なセキュリティリスクが高まることが明らかになり、Docker イメージのメンテナは、インストールした JavaScript パッケージを適切にバージョンアップする必要があることが分かった．Docker イメージに関する研究も行われている．Wu ら [63] は，Docker イメージのビルドについて調査した．エラーが出てビルドに失敗した Dockerfile の修正されるまでの時間は，そのほかの修正にかかる時間よりも長いということを示した．Zhang ら [64] は，コンテナのベースイメージを推薦する手法 DCCimagerec を提案している．DCCimagerec は抽象構文木解析とニューラルネットワークを用いて，Dockerfile の関数コードの構文構造と意味的特徴から，適切な Docker ベースイメージを自動選定する．

コンテナの開発で発生するライセンスリスクに関する研究は現時点では行われていないが，コンテナのビルドにより自動的にインストールされるアプリやパッケージに関する分析は行われている．コンテナ内のパッケージにはライセンス以外にもバグやセキュリティといったリスクも存在する．これらのリスクは，動作環境そのものが共通部品化し，コンテナ開発者による直接的なパッケージ管理が行われないことにより顕在化する．そのため，今後のコンテナ開発では，これらのリスクをいかに可視化するかが重要になると考えられる．

5.5.2 コンテナ品質向上のための Dockerfile 分析

Docker イメージは Dockerfile をビルドすることにより生成される．Docker イメージのビルド不具合やコードの保守性などの課題を解決するため，Dockerfile に関する分析が行われている．

Horton ら [65] は，StackOverflow や GitHub 上で共有されている Python のコードのうち，約 50%が使用ライブラリの依存関係の問題で実行できない問題を指摘している．そこで，インポートエラーなしに Python コードスニペットを実行するために必要な依存関係を分析し，その環境を Dockerfile で提供する DockerizeMe を提案した．パッケージの依存関係の分析には相関ルールマイニングを用いている．Hassan ら [66] は，Dockerfile とアプリケーションのソースコードの関係を分析し，ソースコードの変更に応じて，ファイルパスなどの修正を自動で推奨する RUDSEA を提案している．実際の 1,199 の更新を対象とした評価により、RUDSEA は正しい更新箇所の指摘が全体の 78.5%、具体的な変更コードの推奨が全体の 44.1%に対して行えることを示した．

また，Docker イメージのバージョンの欠落に対応するための研究も行われている．Cito ら [1] は， GitHub と Docker Hub から 7 万件の Dockerfile を収集し，Docker のエコシステムを様々な観点から分析した．その結果，Docker イメージの品質問題の大部分（28.6%）は、ベースイメージの具体的なバージョンの指定が誤っているもしくは欠落していることに起因することを明らかにした．Yin ら [67] は，Docker Hub では，Docker イメージのバージョンなどを保持するタグ情報を自動推薦する D-Tagger を提案した．タグ情報はイメージ継承時の検索などで利用される重要な情報であるが，現在は手動入力となっている．D-Tagger は，学習用の Dockerfile とタグ情報を収集し，Labeled Latent Dirichlet Allocation (L-LDA) を用いて Dockerfile に対する適切なタグを出力する．

また，Dockerfile のコードの保守性に関する分析も行われている．Henkel ら [68] は，Docker の専門家が開発した Dockerfile から抽出したゴールドセット（ベストプラクティスルール）を用いて，Dockerfile の問題を特定するツール binnacle を提案した．binnacle は IDE（東郷魁夷発環境）に組み込んで利用することができ，Github 上の Dockerfile に対して分析したところ，専門家が作成した Dockerfile と比べて平均して 5 倍近くゴールドセットに違反していたことを明らかにしている．Wu ら [69] は，Dockerfile のコードスメルに関する分析を行い，異なる Dockerfile でのコードスメル（保守性の低いコード）の共起関係を明らかにしている．Lin ら [70] は，2015 年 2020 年の 5 年間の Docker イメージと Dockerfile の品質を進化の観点で調査した．すぐに使い始められるように設定されている言語環境やアプリケーションのイメージへの依存関係がより大きくなっていること，イメージを小さく保つというベストプラクティスの採用が増え，Docker イメージサイズが全体的に減少していること、Dockerfile 内のコードスメルの数が減少傾向にあることが分かった．また，マイナス面では、セキュリティリスクとなる過去のベースイメージの使用が増加傾向にあることを明らかにしている．

これらの研究により，人手で作成された Dockerfile の課題に対し，機械的な解決策を提案している．本研究でも Docker イメージ開発において発生するライセンスリスクを前もって予測する手法を提案している．全てのコンテナのパッケージは Dockerfile の各命令を起因としてインストールされるため，今後の展望として，IDE に組み込むなど具体的にツールとして実装し，Horton [65] らのパッケージ依存関係推定機能を組み合わせるなどにより開発者をより支援する手法に発展させることが望まれる．

5.6 まとめ

本章では，Docker イメージ開発におけるライセンス互換性検証を支援するため，機械学習を用いたコンテナ開発早期におけるライセンス互換性検証予測手法を提案した．アプローチとしては，まず，Docker イメージに含まれるパッケージ情報を抽出する．そして，学習データとテストデータの開発進捗度を同一となるようパッケージ利用情報をマスキングしながら，最終的なライセンス互換性検証結果を MLP で予測する．提出手法の評価実験では，開発進捗度に合わせて学習した場合，開発進捗度 10%の時点でも適合率 94%，再現率 96%，F 値 95%の精度でライセンス検証結果を予測できることが分かった．また，開発進捗度全体では，機械学習アルゴリズムに MLP を用いた場合，最も予測精度が高くなる事が分かった．

今後の展望としては，ライセンス互換性違反の原因となるパッケージを特定し，依存関係から代替パッケージを推薦したり，テスト自動化などの CI（継続的インテグレーション）プロセスへのの組み込み，ライセンス互換性違反に対するコンプライアンス遵守を強制化を検証するなど，開発プロセスへの統合技術としての発展を期待する．

第6章　おわりに

本研究では，コンテナ開発における法的リスク特定自動化への 2 つのアプローチを提案した．1 つは，ソースファイルのライセンス特定ツールにおけるライセンスルールの自動生成である．本研究では，未知のライセンスとして検出されたソースファイルのライセンスを特定するための正規表現を自動生成する手法を提案した．本手法は，未知ライセンスと判定されたソースファイルをクラスタリングでライセンス名によって分類する．次に編集距離によって定義される類似度を利用して各クラスタのライセンス記述をフィルタリングする．次に各クラスタごとに系列パターンマイニングを適用し記述パターンを抽出する．最後に記述パターンを正規表現に変換しライセンスルールとして出力する．提案手法を評価するために，FreeBSD-10.3.0, Linux-4.4.6, Debian-7.8.0 から検出した 1,821, 3,561, 2,838 件の未知ライセンス状態を用いたケーススタディを行った．その結果、提案手法は最小限のライセンスルールでより多くのライセンスを識別できること，提案手法で作成したライセンスルールを Ninka に追加することで，ライセンスルールのパフォーマンスが 2%–10%向上することを示した．

もう 1 つは，Docker イメージのライセンス互換性検証結果を開発早期に予測することである．本研究では，最終的なライセンス互換性検証結果を開発早期のパッケージ情報から予測する手法を提案した．まず，Docker イメージをそのパッケージ情報によりベクトル化する．学習データとテストデータの開発進捗度が同一となるよう，パッケージ情報を各開発進捗度ごとにマスキングする．最後に MLP による予測を実施する．Github から抽出した Dockerfile をビルドして生成した 598 件の Docker イメージをもとに評価実験を行った. その結果、開発進捗度 10%の時点でも適合率 94%，再現率 96%，F 値 95%の精度でライセンス互換性検証結果を予測できることが分かった．

今後の展望としては，法的リスク特定後の開発者への支援機能の搭載が考えられる．ライセンスルール生成後の既存ルールとの統合機能や，手作業によるルール命名の支援ライセンス互換性検証後のパッケージ推薦技術を構築できれば，より開発者にとって扱いやすい技術に発展すると期待される．

謝 辞

本研究を進めるにあたり，多くの方々から御指導，御協力を賜りました．この場を借りて，お世話になった方々に，感謝の意を表したいと思います．

指導教員である，和歌山大学 システム工学部 大平 雅雄 准教授には，学部生の頃から，研究の方向性についてはアドバイスや，研究を行う姿勢，論文の書き方など研究の基礎的な部分について熱心なご指導をいただきました．そのおかげで，主体的に研究に取り組むことができ，論文誌へ論文投稿できるまでに成長することができ，とても有意義な大学院生活を送ることができましたことを心から御礼申し上げます．

本論文を審査してくださった，和歌山大学 システム工学部 和田 俊和 教授，風間 一洋 教授に心から御礼申し上げます．特に和田 俊和 教授には，学部生の頃の自身の体調について大変気遣っていただき，多大なるご配慮いただきましたことを心から御礼申し上げます．

共同研究者である福知山公立大学 情報学部 眞鍋 雄貴 講師には，本研究を行うにあたり，OSS ライセンスに関する基礎的な知識，研究の方向性についてご指導いただきました．研究の範囲に限らず，毎週ミーティング，論文執筆，学会発表において熱心なご指導をいただきましたことを心から御礼申し上げます．

共同研究者である奈良先端科学技術大学院大学 先端科学技術研究科 柏 祐太郎 助教には，筆者が学部生の頃から，研究のアドバイスや技術的な問題をサポートしていただき，最終的には研究成果を国際会議に投稿するまで発展させることができましたことを心から御礼申し上げます．

共同研究者である同オープンソースソフトウェア工学研究室メンバーの福井 克法氏には，在学当時より実験のサポートをいただきましたことを心から御礼申し上げます．本研究に限らず，多くの時間を共に過ごした和歌山大学 オープンソースソフトウェア工学研究室の皆様に心から御礼申し上げます．

筆者の勤務先である株式会社日本総合研究所の皆様にも多大なるご協力をいただきました．本研究を進めるにあたり，社会人の身でありながら，大学への入学を認めてくださった高野 宏治朗 副社長執行役員，グループ情報系タスクフォース 石井 政仁 本部長，共通基盤システム本部 杭田 真和 本部長補佐，HR マネジメント部 岩堀 仁 部長，渡辺 久貢 氏に心から御礼申し上げます．また，業務上のご配慮いただきました，システム企画部 加藤 研也 部長，金光 淳一郎 部付部長に心から御礼申し上げます．また，HR マネジメント部 大竹 秀喜 部長，篠崎 宏州 次長，額宮 志織 氏には，多大なるご支援をいただきましたことを心から御礼申し上げます．

最後に，本研究は学部 4 年間での勉学及び修士 2 年間の研究内容が礎となっています．計 6 年の大学に通うことを支援してくださった父・母に対し心より御礼申し上げます．

研究業績

論文誌 Yunosuke Higashi, Masao Ohira, and Yuki Manabe, “Automating License Rule Generation to Help Maintain Rule-based OSS License Identification Tools” Journal of Information Processing (JIP), volume 31, pages 2-12, January 2023.（本紙 4 章に関連）

論文誌 宮崎智己, 東裕之輔, 大平雅雄, “単語分散表現による類義語統一と単語 N-gram によるフレーズ抽出に基づくセキュリティ要件分類手法”, 情報処理学会論文誌, volume 63, number 1 pages 94–103,2022 年 1 月.

論文誌 Yunosuke Higashi, Masao Ohira, Yutaro Kashiwa, and Yuki Manabe, “Hierarchical Clustering of OSS License Statements Toward Automatic Generation of License Rules” Journal of Information Processing (JIP), volume 27, pages 42-50, January 2019.（本紙 3.1 章，4 章に関連）

論文誌 宮崎 智己, 伊原 彰紀, 大平 雅雄, 東 裕之輔, 山谷 陽亮, ”OSS コミュニティにおける開発者の活動継続性を理解するための Politeness 分析,” 情報処理学会論文誌, volume 59, number 1, pages 2-11, 2018 年 1 月.

国際会議（査読あり） Yunosuke Higashi, Katsunori Fukui, Yutaro Kashiwa and Masao Ohira, “A Preliminary Analysis of GPL-Related License Violations in Docker Images,” In Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pages 444-448, March 2022. （本紙 3.2 章に関連）

国際会議（査読あり） Yunosuke Higashi, Yuki Manabe, and Masao Ohira, “Clustering OSS License Statements Toward Automatic Generation of License Rules,” In Proceedings of the 7th IEEE International Workshop on Empirical Software Engineering in Practice (IWESEP), pages 30–35, March 2016. （本紙 3.1 章，4 章に関連）

国内会議（査読あり） 坂本 廉也, 東 裕之輔, 大平 雅雄, ”Dockerfile の依存関係とビルドエラーの関係分析” 第 29 回ソフトウェア工学の基礎ワークショップ (FOSE2022) 2022 年 11 月 採録決定

国内会議（査読あり） 宮崎 智己, 伊原 彰紀, 大平 雅雄, 東 裕之輔, 山谷 陽亮, ”Politeness 分析に基づく OSS 開発者の活動継続性の理解,” ソフトウェアエンジニアリングシンポジウム 2018 論文集, pages 261-262, 2018 年 8 月.

国内会議（査読あり） 東 裕之輔, 眞鍋 雄貴, ”ライセンス特定のためのルール自動生成を目的としたライセンス記述パターン抽出手法,” ソフトウェア工学の基礎 XXIII, 日本ソフトウェア科学会 (FOSE), 2016, volume 23, pages 13-22, 2016 年 12 月.（本紙 4 章に関連）

国内会議（査読あり） 東 裕之輔, 眞鍋 雄貴, 大平 雅雄, ”機械学習を用いたテキスト分類によるライセンス特定のためのルール作成プロセス支援,” ソフトウェア・シンポジウム 2015 in 和歌山 論文集, pages 70-79, 2015 年 6 月.（本紙 3.1 章に関連）

国内会議（査読あり） 眞鍋 雄貴, 東 裕之輔, 大平 雅雄 , ”オープンソースライセンス変更によるプロジェクトへの影響評価に向けて,” 第 21 回ソフトウェア工学の基礎ワークショップ (FOSE2014), pages 273-274, 2014 年 12 月.

参 考 文 献

[1] Jürgen Cito, Gerald Schermann, John E. Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*, pp. 323–333, 2017.

[2] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly, 2016.

[3] 上田 理 (著), 岩井久美子 (監修). OSS ライセンスの教科書. 技術評論社, 2018.

[4] Armijn Hemel. Docker containers for legal professionals. The Linux Foundation. `https://www.linuxfoundation.jp/wp-content/uploads/2020/04/Docker-Containers-for-Legal-Professionals-Whitepaper_v4.ac_-3.pdf` Accessed on February 27, 2023.

[5] John J. Marciniak(著), 片山 卓也 (訳), 鳥居 宏次 (訳), 土居 範久 (訳). ソフトウェア工学大事典. 朝倉書店, 1998.

[6] 河知豊. ソフトウェアライセンスの基礎知識. ソフトバンククリエイティブ, 2008.

[7] Thomas Wolter, Ann Barcomb, Dirk Riehle, and Nikolay Harutyunyan. Open source license inconsistencies on github. *ACM Transactions on Software Engineering and Methodology*, December 2022.

[8] Massimiliano D. Penta, Daniel M. German, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, pp. 145–154, 2010.

[9] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the 7th Conference on Data and Application Security and Privacy (CODASPY'17)*, pp. 269–280, 2017.

[10] Ahmed Zerouali, Valerio Cosentino, Gregorio Robles, Jesus M. Gonzalez-Barahona, and Tom Mens. Conpan : A tool to analyze packages in software containers. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR'19)*, pp. 592–596, 2019.

[11] Antony Martin, Simone Raponi, Théo Combe, and Robert. D. Pietro. Docker ecosystem - vulnerability analysis. *Computer Communications*, Vol. 122, pp. 30–43, 2018.

[12] Cesar D.L.Torre, Bill Wangner, and Mike Rousos. .net microservices: Architecture for containerized .net applications. Microsoft Corporation. `https://learn.microsoft.com/en-us/dotnet/architecture/microservices/` Accessed on February 27, 2023.

[13] Manuel Sojer and Joachim Henkel. License risks from ad hoc reuse of code from the internet. *Communications of ACM*, Vol. 54, No. 12, pp. 74–81, 2011.

[14] Daniel M. German, Yuki Manabe, and Katsuro Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the 10th International Conference on Automated Software Engineering (ASE '10)*, pp. 437–446, 2010.

[15] Robert Gobeille. The fossology project. In *Proceedings of 5th International Working Conference on Mining Software Repositories (MSR '08)*, pp. 47–50, 2008.

[16] OSLC. Open source license checker. `ttps://sourceforge.net/projects/oslc/` Accessed on July 10, 2020.

[17] Ohcount. Ohloh's source code line counter. `https://github.com/blackducksw/ohcount` Accessed on July 10, 2020.

[18] Timo Tuunanen, Jussi Koskinen, and Tommi Kärkkäinen. Automated software license analysis. *Automated Software Engineering*, Vol. 16, No. 3-4, pp. 455–490, 2009.

[19] Daniel M. German, Massimiliano Di Penta, and Julius Davies. Understanding and Auditing the Licensing of Open Source Software Distributions. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC '10')*, pp. 84–93, 2010.

[20] Yuki Manabe, Daniel M. German, and Katsuro Inoue. Analyzing the relationship between the license of packages and their files in free and open source software. In *Proceedings of the 10th IFIP WG 2.13 International Conference on Open Source Systems (OSS '14)*, pp. 51–60, 2014.

[21] Maria Kechagia, Diomidis Spinellis, and Stephanos Androutsellis-Theotokis. Open source licensing across package dependencies. In *Proceedings of the 14th Panhellenic Conference on Informatics (PCI '10)*, pp. 27–32, 2010.

[22] Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. On the detection of licenses violations in the android ecosystem. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, pp. 382–392, 2016.

[23] Yaroslav Golubev, Maria Eliseeva, Nikita Povarov, and Timofey Bryksin. A study of potential code borrowing and license violations in java projects on github. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, pp. 54–64, 2020.

[24] Yu Kashima, Yasuhiro Hayase, Norihiro Yoshida, Yuki Manabe, and Katsuro Inoue. An investigation into the impact of software licenses on copy-and-paste reuse among oss projects. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*, pp. 28–32, 2011.

[25] Xiaoyu Liu, LiGuo Huang, Jidong Ge, and Vincent Ng. Predicting licenses for changed source code. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE '19)*, pp. 686–697, 2019.

[26] Sihan Xu, Ya Gao, Lingling Fan, Zheli Liu, Yang Liu, and Hua Ji. Lidetector: License incompatibility detection for open source software. *ACM Transactions on Software Engineering and Methodology*, Vol. 32, No. 1, pp. 1–28, 2022.

[27] Shi Qiu, Daniel M. German, and Katsuro Inoue. Empirical study on dependency-related license violation in the javascript package ecosystem. *Journal of Information Processing*, Vol. 29, pp. 296–304, 2021.

[28] Georgia M. Kapitsaki, Frederik Kramer, and Nikolaos D. Tselikas. Automating the license compatibility process in open source software with spdx. *Journal of Systems and Software*, Vol. 131, pp. 386–401, 2017.

[29] Rômulo Meloca, Gustavo Pinto, Leonardo Baiser, Marco Mattos, Ivanilton Polato, Igor Scaliante Wiese, and Daniel M German. Understanding the usage, impact, and adoption of non-osi approved licenses. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*, pp. 270–280, 2018.

[30] Massimiliano D. Penta and Daniel M. German. Who are source code contributors and how do they change? In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE '09)*, pp. 11–20, 2009.

[31] Free Software Foundation. Various licenses and comment them. `https://www.gnu.org/licenses/license-list.html.en` Accessed on February 27,2023.

[32] Daniel M. German and Jesus M. Gonzalez-Barahona. An empirical study of the reuse of software licensed under the gnu general public license. In *Proceedings of the 5th IFIP WG 2.13 International Conference on Open Source Systems (OSS '09)*, pp. 185–198, 2009.

[33] Gerald Schermann, Sali Zumberi, and Jürgen Cito. Structured information on state and evolution of dockerfiles on github. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, pp. 26–29, 2018.

[34] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE'06)*, pp. 199–208, 2006.

[35] Larry Smith. Shift-left testing. 2001. `https://www.drdobbs.com/shift-left-testing/184404768` Accessed on February 27, 2023.

[36] Stefano Zacchiroli. A large-scale dataset of (open source) license text variants. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*, pp. 757–761, 2022.

[37] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, Vol. 58, No. 301, pp. 236–244, 1963.

[38] Vladimir Levenshtei. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, Vol. 10, pp. 707–710, 1966.

[39] Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, Vol. 42, No. 1-2, pp. 31–60, 2001.

[40] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 11, pp. 1424–1440, 2004.

[41] Jianyong Wang and Jiawei Han. BIDE: Efficient mining of frequent closed sequences. In *Proceedings of 20th International Conference on Data Engineering (ICDE '04)*, pp. 79–90, 2004.

[42] Timo Tuunanen, Jussi Koskinen, and Tommi Kärkkäinen. Retrieving open source software licenses. In *Proceedings of the International Conference on Open Source Systems (OSS '06)*, pp. 35–46, 2006.

[43] Massimiliano D. Penta, Daniel German, and Giuliano Antoniol. Identifying licensing of jar archives using a code-search approach. In *Proceedings of the 7th Working Conference on Mining Software Repositories (MSR '10)*, pp. 151 –160, 2010.

[44] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*, pp. 63–72, 2011.

[45] Alexander Lokhman, Antti Luoto, Imed Hammouda, and Tommi Mikkonen. Open source legality compliance of software architecture, a licensing profile approach. In *Proceedings of the 8th International Conference on Software Engineering Advances (ICSEA '13)*, pp. 571–578, 2013.

[46] Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi. Analyzing software licenses in open architecture software systems. In *Proceedings of ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '09)*, pp. 54–57, 2009.

[47] Daniel M. German and Ahmed E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pp. 188–198, 2009.

[48] Jorge Colazo and Yulin Fang. Impact of license choice on open source software development activity. *American Society for Information Science and Technology*, Vol. 60, No. 5, pp. 997–1011, 2009.

[49] Daniel A. Almeida, Gail C. Murphy, Greg Wilson, and Mike Hoye. Do software developers understand open source licenses? In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*, pp. 1–11, 2017.

[50] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel M. German, and Denys Poshyvanyk. When and why developers adopt and change software licenses. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME '15)*, pp. 31–40, 2015.

[51] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*, pp. 54–57, 2006.

[52] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen abd Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th International Conference on Mining Software Repositories (MSR'13)*, pp. 319–328, 2013.

[53] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*, pp. 318–343, 2009.

[54] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining (KDD '02)*, pp. 429–435, 2002.

[55] Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, Vol. 40, No. 1–3, p. 185–234, September 1989.

[56] Leo Breiman. Random forests. *Machine Learning*, Vol. 45, No. 1, p. 5–32, 2001.

[57] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, Vol. 20, No. 3, p. 273–297, 1995.

[58] Andrew McCallum and Kamal Nigam. A comparison of event models for naive bayes text classification. In *Proceedings of the workshops at the 15th National Conference on Artificial Inteligence (AAAI '98)*, pp. 41–48, 1998.

[59] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232, 2001.

[60] Mubin U. Haque, Leonardo H. Iwaya, and Muhammad A. Babar. Challenges in docker development: A large-scale study using stack overflow. In *Proceedings of the 14th International Symposium on Empirical Software Engineering and Measurement (ESEM '20)*, pp. 1–11, 2020.

[61] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER '19)*, pp. 491–501, 2019.

[62] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER '19)*, pp. 619–623, 2019.

[63] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An empirical study of build failures in the docker context. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*, pp. 76–80, 2020.

[64] Yinyuan Zhang, Yang Zhang, Xinjun Mao, Yiwen Wu, Bo Lin, and Shangwen Wang. Recommending base image for docker containers based on deep configuration comprehension. In *Proceedings of the 29th International Conference on Software Analysis, Evolution and Reengineering (SANER '22)*, pp. 449–453, 2022.

[65] Eric Horton and Chris Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *Proceedings of the ACM/IEEE 41st International Conference on Software Engineering (ICSE '19)*, pp. 328–338, 2019.

[66] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. Rudsea: Recommending updates of dockerfiles via software environment analysis. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE '18)*, pp. 796–801, 2018.

[67] Kang Yin, Jiahong Zhou, Wei Chen, Guoquan Wu, Jiaxin Zhu, and Jun Wei. D-tagger: A tag recommendation approach for docker repositories. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware (Internetware '18)*, pp. 1–10, 2018.

[68] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*, pp. 38–49, 2020.

[69] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, Vol. 8, pp. 34127–34139, 2020.

[70] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. A large-scale data set and an empirical study of docker images hosted on docker hub. In *Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME '20)*, pp. 371–381, 2020.