Predicting Licenses for Changed Source Code

Xiaoyu Liu[1], LiGuo Huang[1], Jidong Ge[2] and Vincent Ng[3]

[1]Department of Computer Science, Southern Methodist University, Dallas, TX, USA
[2]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[3]Human Language Technology Research Institute, University of Texas at Dallas, Richardson, TX, USA

***Abstract*—Open source software licenses regulate the circumstances under which software can be redistributed, reused and modified. Ensuring license compatibility and preventing license restriction conflicts among source code during software changes are the key to protect their commercial use. However, selecting the appropriate licenses for software changes requires lots of experience and manual effort that involve examining, assimilating and comparing various licenses as well as understanding their relationships with software changes. Worse still, there is no state-of-the-art methodology to provide this capability. Motivated by this observation, we propose in this paper Automatic License Prediction (ALP), a novel learning-based method and tool for predicting licenses as software changes. An extensive evaluation of ALP on predicting licenses in 700 open source projects demonstrate its effectiveness: ALP can achieve not only a high overall prediction accuracy (92.5% in micro F1 score) but also high accuracies across all license types.**

***Index Terms*—Software License Prediction, Mining Software Repository**

I. Introduction

In recent decades, more and more Free and Open Source Software (FOSS) projects have been made available online by developers. The shift towards FOSS projects allows developers to not only contribute to the software community but also benefit themselves [1]. For example, by hosting their FOSS projects on web-based version control platforms (e.g., GitHub, SourceForge, etc.), developers can receive help from third-party testers and other developers to improve the quality of their software systems. Nevertheless, developers who are interested in releasing their projects to the open source community should be aware that the redistribution, reuse and modification of their projects must be regulated under software licenses. Software licenses are important because they are designed to protect the intellectual property of FOSS using licensing mechanisms and copyright notices that determine how an open source software can be (re)used [2].

To apply appropriate software licenses to their software projects, developers need to select from a variety of licenses: either the ones that allow redistributors to incorporate the reused software under different licenses (i.e., *permissive licenses*) or the ones that require developers to use the same license when distributing new software that incorporates the reused software (i.e., *restrictive licenses*) [1]. These licenses range from highly restrictive (e.g., the General Public License (GPL) family, which requires developers to use GPL to distribute new software that reuses GPL software) to less restrictive (e.g., the MIT license, which permits a third party to freely modify, reuse and redistribute the project by keeping term notices). Therefore, selecting the appropriate license for a given piece of software requires a great deal of experience and manual review effort on the part of developers. While there exists methods that help developers choose appropriate licenses when a software project is initially released ([3], [4], [5], [6]), an important question remains: how can licenses be *updated* when *changes* are made in software systems?

Vendome et al. [1] shed light on the aforementioned question by investigating the rationales behind license changes due to software changes from both quantitative and qualitative points of view. Their study reveals that updating the license in the presence of software changes is an even more time-consuming and labor-intensive process than determining the license for a newly released software project. Specifically, a developer has to review each changed source code file against the existing licenses to determine whether there is any license incompatibility, such as the violation of existing license terms/copyright or the presence of a license that is incompatible with the license of another piece of changed source code. Consider the example in Figure 1, which shows that a co-changed source code module,[1] *LogEntry*, is imported to the source code file *XMLPacker.java* that was originally licensed under MPL v1.1 according to its file header. To determine if a change of license is needed, the developer would begin by determining that *LogEntry* is distributed under license GPL v3+. Then, it requires a careful review and comparison of the license restrictions of MPL v1.1 and those of GPL v3+ to reveal a potential incompatibility between them. That is, a software system that imports or uses *LogEntry* is required to adopt GPL v3+, which imposes a stronger restriction than MPL v1.1. Detecting and resolving such incompatibilities thus places a lot of burden on developers.

Unfortunately, according to Vendome et al. [1], existing methods and tools for license prediction ([3], [4], [5], [6]) are insufficient for the task of predicting licenses *in the presence of software changes*. For instance, *Ninka* [4], a state-of-the-art license detection method, uses regular expressions to predict licenses by detecting the presence of license copyright and terms in the header comments of the source code files (i.e., the file header). Hence, Ninka has no problem with (independently) detecting MPL v1.1 as the license adopted

[1]A source code file f_1 is a co-changed file of another source code file f_2 if the two files both changed in a single change commit.

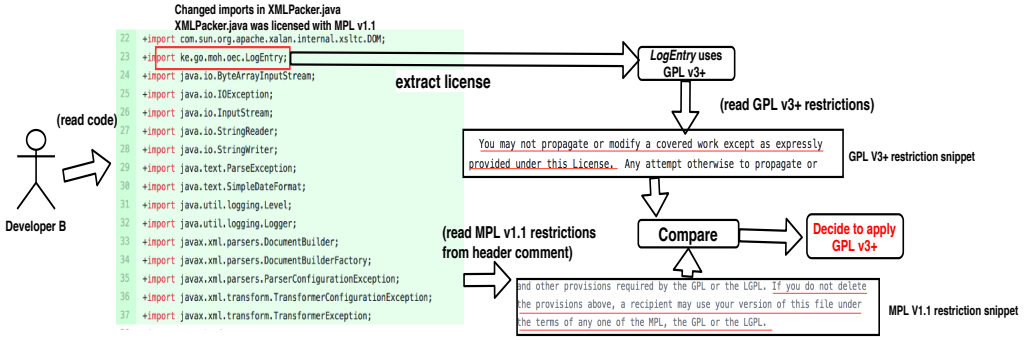

Fig. 1: Example illustrating licensing incompatibility

by *XMLPacker.java before* the code change and GPL v3+ as the license adopted by *LogEntry*. However, *Ninka cannot* detect file dependencies. So, in our example, it cannot take into account the license restriction imposed by the newly imported code module. Failure to do so deprives *Ninka* of its ability to address the license compatibility issues that arise from code changes.

Our goal in this paper is to advance the state-of-the-art in license prediction for software changes. Specifically, we propose Automatic License Prediction (ALP), a novel method and supporting tool for automatically predicting source code file-level licenses for code changes. Leveraging the recent successes of machine learning methods in empirical SE research, we propose a learning-based ALP system. At the core of ALP are four key ideas: (1) exploiting a rich set of features extracted from the inline text of the changed file under consideration; (2) modeling the license of the previous version of the file; (3) exploiting features extracted from the associated software documents and co-changed files; and (4) identifying and resolving incompatibilities, such as those illustrated in Figure 1.

Our contributions in this paper are three-fold. First, we manually annotate the licenses of 57450 changed source code files taken from 700 Java projects hosted on GitHub. To our knowledge, this is the first large-scale effort aiming to create an annotated corpus for license prediction. Second, we propose ALP, the first machine learning approach to license prediction, which considers the dependencies among the licenses of the source code modules. Note that the development of ALP is made possible by the availability of the large amount of annotated training data provided by our corpus. Finally, extensive experiments demonstrate the effectiveness of our approach. In an evaluation on 700 Java projects involving the prediction of 25 software licenses, ALP achieves a micro F1 score of 92.5%, highly significantly surpassing the performance of three baseline systems, including *Ninka*, which only achieves a micro F1 score of 73.5 on the same corpus.

We believe our results have another important ramification. Vendome et al.'s [1] study of the rationales behind license changes due to software changes were based on the automatic annotations provided by *Ninka*. However, *Ninka*'s rather mediocre performance on our corpus casts doubts on the degree to which the conclusions drawn by Vendome et al. are valid. We believe that it is worthwhile to re-examine their conclusions by re-conducting their study using ALP's output, which is considerably more accurate than *Ninka*'s.

II. Data Preparation

We collect a large set of historical change repositories from 700 Java projects hosted on GitHub. These projects and their historical change repositories are previously used by Vendome et al. [1] in the aforementioned empirical study. We determine the ground truth license name (e.g., GPL, MIT, etc.) and version (e.g., v1, v2, etc.) of each changed file in each change commit via an open coding procedure. All changed files are coded by two coders, both of whom are senior software engineering Ph.D. students who have extensive experience in industry as developers. Initially, one of the coders conducted a pilot study on a subset of the changed files and their associated software documents. This subset was chosen in the following manner. First, 250 projects and their change commit histories were randomly chosen from the dataset. Then, one file was selected randomly from each of the change commit histories. The purpose was to obtain as many different types of licenses and relevant text statements as possible. The pilot study resulted in a list of preliminary coding criteria. Each criterion either describes the conditions under which a license is applicable and/or enumerates the license(s) for which a given term is a possible indicator. For example, one criterion says that if the term "AS IS" appears in a license's text, then either LGPL v3+ (a highly restrictive license) or BSD (a fairly restrictive license) should be the license. Moreover, the choice depends on whether an incompatibility between the licenses exists: if there is no incompatibility, then BSD suffices. Then this coder trained the other coder on the coding criteria. After training, both coders simultaneously coded all the changed files in the dataset. As for inter-coder agreement, the coders achieved an agreement ratio (i.e., the percentage of changed files that are assigned the same license by the two coders) of 73.7% and a Cohen's Kappa [7] of 0.597, which indicates moderate agreement [8]. Disagreements in their annotations were resolved by open discussion. Disagreement primarily stems from the coders' differing interpretations of the terms of the licenses. For example, one coder mistakenly assigned

TABLE I: Annotation examples

	Changed file	Software doc	Co-changed file
Example 1	N/A	...you can redistribute it and/or modify it under the terms of the **GNU General Public License version 2**...	can redistribute it and/or modify it under the terms of the **GNU Lesser General Public License** as published by the Free Software Foundation; either **version 2.1 of the License or (at your option) any later version**...
Example 2	...Licensed under the Apache **License Version 2.0**...	program is made available under the terms of the **Eclipse Public License v1.0** which accompanies this distribution...	**The MIT License** Original work sponsored and donated by...

TABLE II: Dataset statistics

# of systems	700
# of commits	8128
# of changed files	57450

(a) Overall statistics

Licenses	# of changed files
Apache v2	18770 (32.7%)
GPL v2	9458 (16.5%)
GPL v3+	5943 (10.3%)
MIT	3125 (5.4%)
LGPL v3+	2609 (4.5%)
LGPL v2.1+	1542 (2.7%)
BSD	1404 (2.4%)
EPL v1	1276 (2.2%)
Other	4960 (8.6%)
Non-licensed	8363 (14.6%)

(b) Per-license frequencies

% of license changed files	6.7%
% of license unchanged files	93.3%

(c) License change statistics

LGPL v3+ as the license after seeing the term "AS IS" because he determined that a license incompatibility exists due to his interpretation of the terms. Every case of disagreement was resolved when the coders reach a common interpretation of the terms.

To enable the reader to get a better idea of how the files are annotated, Table I shows two examples. As can be seen, each example is composed of the changed file under consideration, the associated software document, and its co-changed file. Owing to space limitations, only the snippet of each file/document that is relevant to license prediction is shown. In Example 1, the software document suggests that GPL v2 should be adopted while the co-changed file suggests that LGPL v2.1 should be adopted. There is a license conflict between GPL v2 and LGPL v2.1+. Since LGPL v2.1+ has a stronger copyleft than GPL v2, in order to accommodate the strong copyleft imposed by the co-changed file, this changed file should be labeled as LGPL v2.1+. In Example 2, the three different resources suggest three different licenses: the changed file suggests Apache v2, the document suggests EPL v1, and the co-changed file suggests MIT. Since Apache v2, EPL v1 and MIT are all permissive licenses with no incompatible clauses declared, there is no need to alter the changed file's license. In other words, the changed file should be labeled as Apache v2.

Statistics of the resulting dataset, which contains 57540 changed files annotated with their licenses, are shown in Table II. From Table II-a, we can see that there are totally 24 licenses[2] that have appeared at least once in the 700 projects. Table II-b shows the distribution of the licenses over the 57450 changed files ("Non-licensed" is used when a license is absent in a changed file, while "Other" shows the statistics aggregated over the remaining 14 (lowest-frequency) licenses.[3].) As we can see from Table II-b, the most frequently used license is Apache v2 (32.7%). This is perhaps not surprising: Apache v2 extends software users enough freedom to use it for any purpose. Table II-c shows the percentages of changed files that involved a license change in the collected projects. As we can see, 6.7% of them have their license changed.

Given this dataset, we create a multi-class prediction task, where we seek to predict each changed file as having either one of the 24 licenses or *non-licensed* (i.e., the associated file does not have a license). For the sake of brevity, we will refer to the class *non-licensed* simply as one of the "licenses" to be predicted in the rest of the paper.

III. Baseline Approaches

This section introduces three baseline approaches that we implement for controlled experiments with our ALP system.

A. Ninka

As our first baseline, we employ a state-of-the-art license prediction system, *Ninka* [4]. *Ninka* is inspired by the observation that the information about a source code file license is typically found in the inline textual comment at the beginning of a source code file (i.e., the file header). In other words, *Ninka* detects the license of a changed source code file by relying on its file header. Specifically, given a changed source code file, *Ninka* first extracts the file header and segments it into a sequence of sentences, each of which is normalized by replacing each of its phrases with its equivalent common version without changing its meaning. Then it leverages a set of pre-defined regular expressions built upon these common terms to detect the presence of the license copyrights and terms. Finally, it outputs a *list* of licenses that are matched by their corresponding copyrights or terms in the file header. In our experiments, we use a publicly available implementation

[2]According to German et al. [4], all 24 licenses are frequent FOSS licenses and are detectable by *Ninka*.

[3]These 14 licenses (and their percentages) are: MPL v1.1 (2.1%), ECL v2 (0.7%), LGPL v2.1 (0.7%), LGPL v3 (0.5%), ShareAlike v3 (0.4%), OSL v3 (0.2%), ECL v1 (0.2%), LGPL v1 (0.06%), CPL v1 (0.05%), Apache v1.1 (0.02%), Microsoft (0.01%), CDDL v1 (0.007%), public-Domain (0.003%), GPL v1 (0.002%)

of *Ninka*,[4] considering its prediction for a changed file correct as long as *one* of the licenses in the list of predictions it returns is correct. Note that since we allow *Ninka* to return more than one prediction for a given file, we are effectively giving it an unfair advantage over other systems that return only one prediction per file.

To give the reader a better sense of *Ninka*'s weakness, we apply *Ninka* to the example shown in Figure 1. *Ninka* starts by extracting and normalizing the file header of *XMLPacker.java*. Then all the pre-defined regular expressions are applied to the file header. Among them, the regular expression built for detecting the license MPL v1.1 matches the term "MPL" in the file header sentence *"If you do not delete the provisions above, a recipient may use your version of this file under the terms of any one of the MPL"*. Hence, *Ninka* incorrectly predicts that *XMLPacker.java* adopts MPL v1.1. It fails to make the correct prediction (GPL v2) because it does not consider the potential conflict with the license restrictions imposed by the changed import code module *LogEntry* licensed under GPL v2, which specify that *"You may not propagate or modify a covered work except as expressly provided under this license"*.

B. Caller-Callee (CC)

As mentioned in the introduction, developers often reason about license changes based on code imports. German et al. [9] conduct an empirical study showing that software package dependency needs to be combined with license information to identify potential cases of redistribution with license incompatibilities. In other words, any change in the licenses of the imports (imports are often referred to as the *callee*) may affect the license of the changed file under consideration (changed files are often referred to as the *caller*) [9]. For example, if a *caller A.java* licensed under Apache v2 imports *callee B.class*, while *B.class*'s license is updated to GPL v3+, then both *B.class* and *A.java* should adopt GPL v3+ since GPL v3+ is more restrictive than Apache v2. Consequently, developers examine the imported code modules and their licenses, typically assigning to the changed file the license that is associated with the largest number of imported code modules. Our second baseline, *Caller-Callee (CC)*, attempts to mimic this human decision process. Note that it is applicable to both imported third-party external libraries as well as project-internal classes.

We implement *CC* as follows. Given a changed file whose license is to be predicted, *CC* first extracts all imports using an off-the-shelf tool called *QDox* [10]. Next, it extracts the licenses associated with the imported code modules. We leverage two tools to do this: *Ninka* [4], which extracts the licenses of imported local classes and *LicenseFinder* [11], which extracts the licenses of imported third-party libraries. Using the extracted licenses, *CC* assigns a license to the changed file under consideration based on the *majority rule*. Specifically, the extracted licenses are ranked by the number of imported code modules that adopt them (a.k.a. the *import vote*), and the license with the largest number of votes is returned. Our decision to employ the import vote is inspired by (1) German et al.'s [9] finding that software package dependency needs to be combined with license information to identify potential cases of redistribution with license incompatibilities; and (2) the intuition that developers simply license a piece of software under the one adopted by a majority of its imports. If none of the imports are licensed, *CC* will classify the changed file as *non-licensed*.

[4]http://ninka.turingmachine.org

imports	license
ch.lambdaj.Lambda	Apache v2
ch.lambdaj.group.GroupCondition	Apache v2
ch.lambdaj.collection.LambdaCollections	Apache v2
ch.lambdaj.function.convert.Converter	Apache v2
ch.lambdaj.collection.LambdaSet	Apache v2
com.google.common.collect.ForwardingList	Apache v2
src.main.java.com.lexicalscope.fluent.map.FluentMap	Apache v2
ch.lambdaj.collection.LambdaMap	Apache v2
ch.lambdaj.function.aggregate.Aggregator	Apache v2
src.main.java.com.lexicalscope.fluent.collection.FluentCollection	non-licensed
ch.lambdaj.collection.LambdaList	Apache v2
ch.lambdaj.collection.LambdaGroup	Apache v2
com.google.common.collect.Lists	Apache v2
org.hamcrest.Matcher	Apache v2
static com.lexicalscope.fluent.FluentDollar.*	GPL v2
src.main.java.com.lexicalscope.fluent.FluentDollar	Apache v2
src.main.java.com.lexicalscope.fluent.adapters.ConverterFunction	non-licensed

Fig. 2: An example of imported code modules and licenses of *FluentList.java*

To give the reader a better sense of *CC*'s weakness, we apply it to the example shown in Figure 2. *CC* starts by extracting all imported classes and libraries of *FluentList.java* and then determines which license each of them adopts. All imports and their licenses are listed in Figure 2. To decide which license to choose for *FluentList.java*, *CC* determines that 14 imported code modules adopt Apache v2, one adopts GPL v2, and two are *non-licensed*. Using the import vote, it incorrectly predicts that the license of *FluentList.java* is Apache v2. In particular, it fails to predict the correct license (GPL v2) because it does not take into account the license compatibility of the different license restrictions from the imported code modules.

C. Previous Version (Prev)

Our third baseline, *Previous Version*, is motivated by the observation that only 6.7% of the license of a code file changes from one version to another (see Table II-c). To exploit this observation, *Prev* first predicts the license of the first version of each file, and for each subsequent change to the file, it simply predicts its license to be the same as the one that was used in its previous version (which essentially is the one predicted for the first version). In our experiments, *Prev* uses the Basic ALP system (see the next subsection) to predict the license of the first version of each file.

IV. Our Approach

In this section, we present ALP, our learning-based system for predicting licenses in changed source code files. ALP trains a classifier to classify the file as belonging to one of the 25 licenses in our corpus. For ease of exposition, we will decompose the description of ALP into four steps, starting

with the basic system and then incrementally augmenting it in subsequent steps.

A. Step 1: Building the Basic ALP System

The basic ALP system trains a 25-class classifier for classifying a file as belonging to one of 25 licenses. Below we present the details on how this classifier is trained and applied.

a) Training the classifier: To train the classifier, we create one training instance for each changed file in the training set. The label of an instance is the license of the corresponding file (or *non-licensed* if the file does not have a license). Each training instance is represented by two types of features.

The first type of features, **code-inline text features**, are extracted from each line of the source code inline text. The motivation behind these features should be fairly obvious: the inline text of a source code file may reflect the settings of this file, including its license adoption. We generate three kinds of code-inline-text features: unigrams (i.e., word tokens), bigrams (each of which is a pair of consecutive word tokens), and skipgrams (each of which is a pair of word tokens that are separated by exactly one other word token). We obtain word tokens from each line of the source code inline text using the tokenizer in the Stanford CoreNLP toolkit [12]. All code-inline-text features are binary features encoding the presence (value=1) or absence (value=0) of the corresponding unigram/bigram/skipgram in any line of the code-inline text. For example, given a line in the header comment "OpenEMRConnect is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY", the following skipgram features will have their values set to 1: *OpenEMRConnect-distributed*; *is-in*; *distributed-the*; *in-hope*; *the-that*; *hope-it*; *that-will*; *it-be*; *will-useful*; *but-ANY*; *WITHOUT-WARRANTY*.

Intuitively, any change in the source code inline text of a changed file from its previous version may reflect the intents behind source code changes, especially those that are directly relevant to the adoption and update of software licenses. We exploit changes in the source code inline text for license prediction by encoding them as **diff features**. Specifically, diff features are extracted from each line of the source code inline text that differs from its immediately previous version according to the Linux DIFF command. Like the code-inline text features, we generate three kinds of diff features: unigrams, bigrams, and skipgrams. We extract these features from each changed line of the source code and encode them as binary features, each of which indicates the presence (value=1) or absence (value=0) of the corresponding unigram/bigram/skipgram in any changed line of the code-inline text.

To train the 25-class classifier, we employ the linear-chain conditional random field (CRF) learning algorithm as implemented in the Wapiti software package [13]. The motivation behind our choice of CRF as the underlying learner will become obvious when we describe the next step.

b) Applying the classifier: After training, the resulting classifier can be used to label each test instance. Test instances are created in the same way as the training instances. As described before, we use the trained CRF to classify a test instance as having one of the 25 licenses.

B. Step 2: Modeling the Previous License

License adoption depends not only on the current state of the changed file, but also on its past states. For example, if a changed file *A.java* was licensed with GPL v3+, there is no need to update its license when a new import with MPL v1.1 is added because GPL v3+ has more "copyleft" than MPL v1.1. However, since Basic ALP predicts the license of each changed file independently of the other files, it does not exploit a changed file's *previous* license(s). ALP2 is an extension of Basic ALP that attempts to *implicitly* exploit the license of a changed file's immediately previous version. More specifically, rather than predicting each changed file's license independently, we cast our license prediction task as a *sequence prediction* task.

Recall that given a sequence $x_1 x_2 \ldots x_n$ as input, the goal of sequence prediction is to output a sequence $y_1 y_2 \ldots y_n$ of the same length. In other words, output element y_i is assumed to be the predicted class for input element x_i. In the context of license prediction, each input sequence $x_1 x_2$ is a sequence of length 2, where x_2 corresponds to the changed file and x_1 corresponds to the previous version of the changed file.[5] Hence, $y_1 y_2$, the output sequence produced for $x_1 x_2$, will also be of length 2, where y_2 is the predicted license for x_2 and y_1 is the predicted license for x_1.

ALP2 uses CRFs (as implemented in the Wapiti software package) to learn how to label sequences. Recall that CRFs are inherently sequence labelers. In fact, its sequence labeling capability distinguishes itself from the majority of the widely used machine learning algorithms. During training, a CRF is trained to maximize the probability of seeing the correct output sequence given an input training sequence. During testing, the Viterbi algorithm [14] is used to decode the most probable output sequence given an input test sequence. An important aspect of Viterbi is that it captures the *relationship* between consecutive elements in an output sequence. Since we only have sequences of length 2, CRF helps us capture the relationship between license y_1 and license y_2 in the prediction process. For instance, if license y_1 is rarely followed by license y_2 in the training data, the CRF learning algorithm will learn a model that assigns a low probability to this and other unlikely license sequences. In contrast, if y_1 is frequently followed by y_2 in the training data, the CRF learner will learn a model that assigns a high probability to this and other likely license sequences. Hence, a CRF can potentially allow us to improve the prediction of y_2 (the license of the changed file) by exploiting y_1 (the license of the previous version of the changed file). We represent x_1 and x_2 using the same features

[5] We could employ longer sequences to capture a longer history of a given changed file's previous licenses, but preliminary experiments indicate that employing sequences of length more than 2 does not provide additional gains, presumably because a file's license is primarily dependent on that of its immediately preceding version.

that we used in Basic ALP for encoding a changed file and its previous version.

Two points deserve mention. First, for each output sequence y_1y_2, we use y_2 to be the predicted license for changed file x_2. However, we do *not* use y_1 to be the predicted license for x_1. The reason is that in practice, when predicting the license for x_1, x_2 may not even exist. Hence, it does not make sense to use information from x_2 when predicting x_1.

Second, we mentioned above that each input/output sequence for our license prediction task is of length 2. This is not true in a small number of cases, however. Recall that a changed file may not have a previous version (e.g., it is a newly created file). In that case, we will create a length one training/test sequence for each of these changed files that appears in the training/test set.

C. Step 3: Adding New Knowledge Sources

Next, we augment ALP2's feature set with additional features extracted from two sources, the software documents associated with the changed file under consideration and its co-changed files.

1) Extracting features from software documents: The software documents associated with a changed file can sometimes provide important clues as to which license the file should adopt. As an example, consider the software document shown in Figure 3, which is a LICENSE file.[6] The phrases highlighted in yellow, including "retain the above copyright notice", "reproduce the above copyright notice", and "name of the author may not be used to endorse or promote", are relevant to determining that the license that the file should adopt is BSD, as they are consistent with the terms of the BSD license. While a software document may contain useful information as far as license prediction is concerned, this example illustrates why *automatically* extracting such information is not always straightforward. First, only a small portion of the document may be relevant, so one challenge involves *locating* where the useful information is. Second, the useful information may not be expressed as explicitly as the name of the license that the file should adopt. Third, the license declared in the document may not be applicable to all the changed files while the source code changes. Nevertheless, being learning-based, our ALP system should be able to learn the association between phrases and licenses.

Motivated by these observations, we propose to extract *document-text features* from all the software documents (i.e., the readme, the POM file, and the license file) that are related to the changed file under consideration. Specifically, given a changed source code file, all the related software documents are first retrieved based on the relevance to the change. Then, a set of document-text features are extracted from each line of the retrieved software documents to represent their textual contents. Like code-inline text features, we also have three types of document-text features, namely unigrams, bigrams, and skipgrams. For each document-text feature, its value is 1 if the corresponding feature is present in the software document file. Otherwise, its value is 0.

[6]Only a snippet of the document is shown owing to space limitations.

```
Copyright (c) 2011 Peter McQuillan
Based on the ALAC decoder - Copyright (c) 2005 David Hammerton

All rights reserved.

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

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * 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.
    * The name of the author may not be used to endorse or promote products
      derived from this software without specific prior written permission.
```

Fig. 3: A snippet of a software document

2) Extracting features from co-changed files: Like software documents, co-changed files can similarly be useful for predicting the license of a given changed file. Recall that a file f_1 co-changed with another file f_2 if they both changed in a single change commit. Our example in the introduction, which was shown in Figure 1, illustrated why co-changed files are potentially useful for license prediction. In that example, the file *XMLPacker.java* was originally licensed under MPL v1.1 according to its file header, but when a co-changed source code module, *LogEntry*, was imported, it should adopt the stricter license that *LogEntry* adopts, GPL v3+. As mentioned before, without analyzing the dependencies among different files, it would not be possible to identify GPL v3+ as the correct license to use after the code change.

However, neither Basic ALP nor ALP2, the two systems we introduced in Steps 1 and 2, exploits the potentially useful information from co-changed files. In light of this weakness, we seek to extract *co-change features* from a source code file *co-changed* with the changed file whose license is to be determined. Motivated in part by the features used in Basic ALP and ALP2, we similarly extract from a co-changed file *code-inline text features* (i.e., unigrams, bigrams, and skipgrams) as well as *diff* features, which encode the difference in content between the co-changed file and its previous version. A natural question is: how many co-changed files should we use to extract co-change features from? Using all of them could pose a computational efficiency problem, so we could use a subset of them. However, if only a subset of them were to be used, which ones should be chosen? Recall from the example in Figure 1 that a co-changed file will be most useful for license prediction if it suggests a license that is *different* from the one suggested by the changed file under consideration. Hence, it makes sense for us to use those co-changed files that suggest a different license. Of course, it is possible that none of the co-changed file suggests a different license than the one that the changed file suggests. If this happens, the co-changed files will be randomly chosen.

Since we prefer to choose those co-changed files that suggest a different license, the question, then, is: how many of them should be used? To answer this question, we employ an empirical observation of our corpus: rarely do we see more than two different licenses suggested by a set of co-changed files. Given this observation, it is plausible that using just one co-changed file may suffice as long as it suggests a different license. To empirically determine how the number of co-changed files used to extract co-change features would impact overall performance, we will conduct experiments where we extract co-change features from five co-changed files[7] and from just one co-changed file.

An important question that we have eluded so far is: how do we know whether a co-changed file suggests a different license than the one that the changed file under consideration suggests? Our idea is to use our ALP2 system. Specifically, we will use ALP2 to predict the license of a changed file and all of its co-changed files, and select from those co-changed files whose *predicted* license is different from the original changed file's predicted license according to ALP2. Of course, ALP2 is not perfect, but it provides a viable way of identifying such co-changed files.

We retrain ALP2 by augmenting its feature set with the document-text features and the co-change features.[8] Note that these two additional types of features can be used in combination and in isolation. When they are used in combination, we name the resulting system ALP2+Doc+Co, as ALP2 is trained with three types of features: document-text features, co-change features, and the original features extracted from the changed file under consideration. When they are used in isolation, we end up with two systems, ALP2+Doc and ALP2+Co, depending on which of them is used to augment ALP2. However, if software documents or co-changed files are not available for the changed file under consideration, no document-text features and co-change features can be extracted, in which case we will simply set the values of these features to 0.

D. Step 4: Modeling Conflicts

As discussed before, software documents and co-changed files are most useful for license prediction if they suggest a different license than the one suggested by the changed file under consideration. In the previous subsection, the conflicts that resulted from the different licenses suggested by different sources of information are resolved *implicitly* by the CRF. More specifically, the CRF has access to *one* set of features extracted from the different sources and determines the license for the changed file under consideration.

We hypothesize that license prediction performance could be improved if we model the aforementioned conflicts *explicitly*. Specifically, we propose to first identify the set of instances with conflicts (i.e., the instances for which more than one license is suggested by different sources), and then *learn* to determine which source of information should be used to predict the license of a changed file when conflicts arise. Before we explain why we believe this "explicit" approach is potentially better than the "implicit" approach used in Step 3, we provide the details of our explicit approach, which is composed of two stages.

Stage 1 centers around one question: how can we identify the "conflict" instances? Given a changed file, we first train three CRFs to predict its license. The first CRF is trained only on all and only the features extracted from each changed file (i.e., the code inline text features and the diff features) in the training set. The second CRF is trained only on all and only the document-text features extracted from a changed file's associated software documents. The third CRF is trained on all and only the co-change features extracted from co-changed files. We identify an instance as a conflict instance if at least two of the three CRFs predict more than one license for the changed file under consideration. Note that if a changed file does not have any associated software documents or co-changed files, the corresponding instance will not be marked as a conflict instance.[9]

Stage 2 centers around another question: how can we *resolve* the conflicts that arise in the conflict instances identified in Stage 1? We answer this question by training a *conflict resolver*. Our idea is to cast the conflict resolution task as a *ranking* task, where we train a discriminative *ranker* to resolve conflicts using the ranker-learning algorithm implemented in the libSVM software package [15]. Specifically, we create one ranking problem for each conflict instance (i.e., each changed file determined to have a license conflict) identified in Stage 1. The instances to be ranked in a ranking problem are created as follows. The first instance is represented using all and only the features extracted from the changed file under consideration. The second instance is represented using all and only the document-text features extracted from all of the associated software documents. For each co-changed file, we will create one instance that is represented using all and only the co-change features extracted from the co-changed file. As mentioned in Step 3, we will experiment with using one co-changed file and using five co-changed file. This means that each ranking problem will be composed of $2-7$ instances: we know that it will contain at least two instances because the changed file was determined to have a conflict in Stage 1; at the same time, we know the upper bound is 7 because besides the changed file with license to be predicted, we can have at most one instance representing the software document and at most five instances corresponding to the five co-changed files.

The goal of the ranker-learning algorithm is to rank the instances in each ranking problem so that the ones that predict

[7]If fewer than five co-changed files are present, we will just use all of them.

[8]What this means is that the document-text features and the co-change features need to be computed for both the training instances and the test instances. In particular, to select which co-changed files to compute co-change features from for a given changed file in the *training* set, we use ALP2 to predict the license of each changed and co-changed file by performing five-fold cross validation on the *training* set.

[9]Note that we need to identify conflict instances from both the training set and the test set. We use the three CRFs to predict the license(s) of a changed file in the *training* set by performing 5-fold cross-validation on the training set.

the correct license are ranked higher than those that predict an incorrect license. With this goal in mind, we assign the rank value to each instance in each ranking problem as follows. If the license associated with an instance is correct, its rank value is HIGH; otherwise; its rank value is LOW. Note that the license associated with an instance is predicted by one of the three CRFs in Stage 1. For instance, the license associated with the instance corresponding to the software documents is the one predicted by the second CRF in Stage 1.

The resulting ranker can be applied to the conflict instances in the test set. For each conflict instance in the test set, a ranking problem will be created. This ranking problem is created in the same way as those in the training set. The ranker is then used to rank the instances in the ranking problem. The license associated with the highest-ranked instance according to the ranker will be our system's predicted license for the changed file under consideration. Note that the ranker will be applied to all and only those changed files that are determined to have a license conflict.

We believe our approach of explicitly modeling conflicts has at least two advantages over the implicit approach described in Step 3. First, (explicitly) identifying conflict instances enables us to learn a ranker to handle them specifically. This contrasts with the implicit approach, where all of the instances, regardless of whether they are conflict instances or not, are being classified by *one* model. In other words, the conflict instances, which are supposedly the difficult cases in license prediction, may be given less attention by the CRF model in the implicit approach because the CRF is being trained on both the easier (non-conflict) instances and the difficult (conflict) instances. Second, our ranker is not trained to directly predict licenses. Rather, it ranks instances corresponding to different sources of information. Consequently, compared to the CRFs trained in Steps $1-3$, the ranker will have less bias towards classifying a conflict instance as belonging to one of the frequently occurring licenses in our corpus.

V. Empirical Evaluation

A. Experimental Setup

The goal of our empirical evaluation is to determine how accurately ALP can predict licenses in software changes. Our evaluation dataset is composed of changed source code files collected from 700 Java projects, where each file can be classified as belonging to one of 25 licenses.

Evaluation settings. We did not apply any text preprocessing to the relevant software documents we retrieved or the changed files. Hence, all systems, including the three baselines and all variants of our ALP system, are performed on the original (un-preprocessed) software documents and/or changed files. Given that ALP is learning-based, we evaluate it by adopting a five-fold cross validation strategy, in which the subject projects are evenly distributed into five folds. In each fold experiment, we use three folds for training ALP, one fold for development (i.e., parameter tuning), and the remaining fold as our held-out test set.

Two points deserve mention. First, using a five-fold cross-validation strategy ensures that the entire dataset is used for training, parameter tuning and testing. For parameter tuning, we tune the regularization parameter *C* associated with each CRF and each SVM ranker we train. Intuitively, the larger the *C* value is, the higher the penalty on training error is. We choose the *C* value that maximizes the overall micro F1 score (see below for details) on development data. Second, note that we divide the subject projects into five folds, meaning that all the files associated with a particular project will appear in the same fold. The reason for doing this is simple: in reality, a license prediction system will likely be used to predict licenses for the files in a totally new project. In other words, we cannot assume that there is any relationship between the training projects and the test projects. Hence, our dividing the projects into folds mimics this real-life application scenario. Not surprisingly, the learning task resulting from this particular way of creating the five folds is also harder, as it renders any project-specific knowledge that our system learns from the training data useless when the system is applied to the unseen projects in the test data.

Evaluation metrics. We use per-license precision, recall and F1 score to measure the performance of our systems. The precision (P) for license l is the percentage of changed files predicted as l that are correct with respect to the gold set (i.e., Precision = $\frac{TP}{(TP+FP)}$). The recall (R) for license l is the percentage of changed files licensed under l that are correctly predicted as l (i.e., Recall = $\frac{TP}{(TP+FN)}$). The F1 score is the harmonic mean of precision and recall (i.e., F1=$\frac{2*R*P}{(R+P)}$)

To facilitate comparisons between different systems, we also compute the *overall* performance of each system by *aggregating* the per-license results. Specifically, we employ two commonly-used metrics, macro F1 and micro F1. Macro F1 is the unweighted average of the per-license F1 scores. Micro F1 is the fraction of instances that are correctly classified. Hence, macro F1 gives equal importance to each license, whereas micro F1 puts more weights on more frequently occurring licenses.

Statistical significance and effect size. To determine whether the performance difference between two systems is statistically significant or not, we conduct the Wilcoxon rank-sum test. The type of distribution used for Wilcoxon rank-sum test is normal distribution. Following Miller [16], the result of a significance test can be interpreted as follows: The performance difference between the two systems under comparison is (1) *highly significant* if the null hypothesis (i.e., there is no performance difference between the two systems) can be rejected at the 0.01 level; (2) *significant* if it can be rejected at the 0.05 level; and (3) *moderately significant* if it can be rejected at the 0.1 level. Otherwise, the difference is statistically indistinguishable. Moreover, to evaluate the amount of performance difference between the two systems under comparison, we compute Cliffs delta [17], a non-parametric effect size measure. According to Romano et al. [18], the difference implies (1) a *large* effect size if the

TABLE III: Five-fold cross-validation results. The strongest result in each column is boldfaced.

	Systems	macro-F1	micro-F1
1	Ninka	38.2	73.5
2	CC	17.3	39.6
3	Prev	30.6	66.3
4	Basic ALP	38.9	82.2
5	ALP2	46.4	88.3
6	ALP2+Co1	45.7	87.9
7	ALP2+Co5	45.9	88.8
8	ALP2+Doc	47.9	90.3
9	ALP2+Doc+Co1	48.3	90.9
10	ALP2+Doc+Co5	47.9	90.3
11	ALP2-Ranker1	**79.2**	**92.5**
12	ALP2-Ranker5	77.4	92.5

delta value is greater than 0.474; (2) a *medium* effect size if the delta value is greater than 0.33; and (3) a *small* effect size otherwise.

B. Results and Discussion

This section empirically answers our research questions.

RQ1: *Which license prediction system performs the best?*

Five-fold cross-validation results are shown in Table III. Each row shows the macro and micro F1 scores of one system.

A few points deserve mention. First, *Ninka* is the best of the three baselines (rows 1 to 3), achieving a micro F1 of 73.5 and a macro F1 of 38.2. In particular, it highly significantly outperforms *Prev*, the second best baseline, with a large effect size in terms of macro F1 and significantly outperforms it with a large effect size in terms of micro F1. *Prev* in turn highly significantly outperforms *CC*, the weakest baseline, with a large effect size in terms of both macro and micro F1.

Second, Basic ALP (row 4), the most basic variant of ALP, performs as least as well as *Ninka* (row 1), the best baseline. Specifically, Basic ALP achieves micro and macro F1 scores of 82.2 and 38.9, which represents a highly significant improvement of 8.7 points with a large effect size in micro F1 and a moderately significant improvement of 0.7 points with a large effect size in macro F1.

Third, ALP2 (row 5), which casts license prediction as a sequence prediction problem, highly significantly outperforms Basic ALP (row 4) with a large effect size in terms of micro F1 and moderately significantly outperforms it with a large effect size in terms of macro F1. These results provide suggestive evidence that modeling the immediately previous license is useful for predicting both frequent licenses (because of the improvement in micro F1) and infrequent licenses (because of the improvement in macro F1).

Fourth, incorporating additional knowledge derived from software documents and co-changed files as features for training ALP2 is generally, though not always, helpful for license prediction. As mentioned before, the document-text features (derived from the software documents) and the co-change features (derived from the co-changed files) can be applied in isolation and in combination with the changed file's features that are originally used to train ALP2. Results of adding only co-change files to ALP2 are shown in row 6 (ALP2+Co1, where co-change features were derived from just one co-changed file) and row 7 (ALP2+Co5, where co-change features were derived from five co-changed files).[10] As we can see, adding co-change features may not always yield better performance. In contrast, adding only document-text features (ALP2+Doc, row 8) yields small, but moderately significant improvements with a medium effect size in terms of both macro and micro F1 scores. When the two types of features are applied in combination, we see small, insignificant gains in both micro and macro F1 scores when one co-changed file was used (ALP2+Doc+Co1, row 9). Overall, these results seem to suggest that document-text features are more useful than co-change features for license prediction when used in combination with the features derived from the changed file. In addition, deriving features from five co-changed files yields results that are statistically indistinguishable from those obtained using only one co-changed file.

Finally, comparing rows 11 and 12 with rows 6 to 9, we see that explicitly modeling and resolving conflicts using a ranker (ALP-Ranker1 and ALP-Ranker5) is much more effective in improving ALP2 than implicitly resolving conflicts (by incorporating features derived from different sources into just one feature set). Again, we have two sets of ranking results, one obtained by employing one co-changed file and the other five co-changed files. Note that the difference between these two sets of results is indistinguishable. Both sets of ranking results are highly significantly better than the best implicit results (row 9) with a large effect size in terms of macro F1 and significantly better than it with a large effect size in terms of micro F1. It is worth noting that in comparison to row 9 (the best implicit results), the macro F1 score improves by more than 30 points. This is very encouraging, since it is common for learning-based systems to sacrifice minority class performance for majority class performance (because of their bias towards classifying an instance as belonging to a majority class). These results suggest that our idea of training a ranker to *not* predict licenses directly can effectively mitigate the problem that skewed class distributions typically bring about.

RQ2. *How do the systems perform on the easy, difficult, and conflict instances?*

To gain additional insights into the different ALP variants, we report the performance of different systems, including the baselines, on three *subsets* of the instances in our dataset.

First, we compare system performance on only the Ninka-detectable instances (i.e., the set of instances whose license can be predicted by *Ninka*). They account for 60.6% of the instances in our dataset. They are of interest because they are the easy-to-classify instances: their licenses can be simply extracted using one of *Ninka*'s high-precision regular expressions. Macro and micro F1 scores on these easy instances are shown under the "Ninka-det" column in Table IV. As we can see, *Ninka* achieves near-perfect performance on these

[10]Due to randomness involved in the selection of the one/five co-changed files, we repeat each of these experiments five times and report the average F1 scores in Table III.

TABLE IV: Five-fold cross-validation results of systems on Ninka-detectable, Ninka-undetectable, and conflict instances

	Systems	Ninka-det		Ninka-undet		Conflict	
		ma F1	mi F1	ma F1	mi F1	ma F1	mi F1
1	Ninka	97.2	98.1	0.0	0.0	32.3	56.9
2	CC	38.1	41.7	18.7	9.6	15.6	37.0
3	Prev	88.9	84.3	45.0	28.6	22.2	49.5
4	Basic ALP	98.0	98.8	49.5	37.6	30.8	64.2
5	ALP2	97.8	98.8	67.5	57.4	37.9	69.8
6	ALP2+Co1	97.5	98.7	67.2	56.5	37.0	69.8
7	ALP2+Co5	97.8	98.8	68.6	59.6	37.5	72.0
8	ALP2+Doc	98.0	99.0	72.2	64.6	41.3	76.5
9	ALP2+Doc+Co1	97.8	98.8	73.3	67.8	41.6	77.3
10	ALP2+Doc+Co5	97.6	98.8	71.4	65.2	41.9	78.1
11	ALP2-Ranker1	99.1	98.7	90.0	85.0	76.7	86.1
12	ALP2-Ranker5	99.1	98.3	91.5	83.8	79.4	87.7

instances, which is not surprising. Moreover, all of the ALP variants perform at least as well as *Ninka*, with the best results achieved by the rankers (rows 11 and 12).

Second, we compare system performance on only the Ninka-undetectable instances (i.e., the set of instances for which *Ninka* failed to predict any license). They account for 39.4% of the instances in our dataset. These instances are of interest for one important reason: since *Ninka* is a state-of-the-art system and these instances cannot be classified by *Ninka*, any success in predicting their licenses represents a solid advance over the current state-of-the-art. Macro and micro F1 scores on these "difficult" instances are shown under the column "Ninka-Undet" in Table IV. As we can see, more sophisticated ALP variants tend to yield better performance on these difficult instances than their simpler counterparts. In fact, comparing the results on the Ninka-detectable instances and the Ninka-undetectable instances, we can see that our extensions to the Basic ALP system have primarily helped to predict the licenses of the difficult instances. The best performance on the difficult instances is achieved by ALP2+Ranker1 (row 11): macro and micro F1 scores of 90 and 85, respectively.

Finally, we compare system performance on only the instances that are determined to be conflict instances according to ALP2-Ranker1. Results on the conflict instances will shed light on how well the ALP variants, particularly the rankers, are in resolving conflicts. Macro and micro F1 scores on these instances are shown under the "Conflict" column in Table IV. As we can see, *Ninka* achieves a micro F1 score of 56.9, meaning that not all conflict instances are difficult to classify. A closer examination of the conflict instances reveals the reason. Recall that these instances are *predicted* to be conflict instances. Specifically, some easy (Ninka-detectable) instances that do not have conflicts are mis-predicted to have conflicts. As an example, *Ninka* predicts an instance as having license A, and the associated LICENSE file simply says a license is needed (without specifying which license should be used). This is an instance that does *not* have a conflict, but the CRF classifies the software document as *non-licensed*, thus erroneously creating a conflict instance for an easy instance. Nevertheless, the substantially higher macro and micro F1 scores achieved by the rankers show that they have successfully classified many difficult conflict instances.

TABLE V: Examples of errors made by ALP2-Ranker1

	Changed file	Sofware doc	Co-changed file
Example 1	...you can redistribute it and/or modify it under the terms of the **GNU General Public License version 2**... You may not impose any **further restriction** on the recipients' exercise of the rights granted herein...	N/A	...under the conditions of the **GNU General Public License Version 3**...you may add to a covered work material governed by the terms of that license document provided that the **further restriction** does not survive such relicensing or conveying...
Example 2	...Licensed under the Apache **License Version 2.0**...	...under the terms of the GNU General Public **License version 2.0**...	N/A

RQ3. *What are some of the errors made by our best ALP variant, ALP2-Ranker1?*

To address this research question, we show in Table V two conflict instances that the ranker misclassified. As can be seen, each example is composed of the changed file under consideration, the associated sofware document, and the co-changed file. Owing to space limitations, only the snippet of each file/document that is relevant to license prediction is shown.

Example 1 shows that the changed file originally adopts GPL v2 while the co-changed file adopts GPL v3+. Both GPL v2 and GPL v3+ are weak copyleft licenses. While the correct license is GPL v2, the conflict resolver suggests a license change to GPL v3+. In cases like this, developers will typically keep GPL v2 as the changed file's license since it is a major license (16.5% of the source code files across all subject projects in our dataset have this license). To improve the accuracy of ALP2-Ranker1, one can encode the license type (e.g., weak copyleft) and the popularity of a license as features.

Example 2 shows that the changed file originally adopts Apache v2 while the software document suggests GPL v2. Both licenses are loss copyleft (either weak or no copyleft) licenses. While the correct license is GPL v2, the conflict resolver mistakenly labels the changed file as Apache v2. To correctly classify this instance, however, the resolver may need to understand that GPL v2 does not permit incorporating one program into another proprietary program, such as linking proprietary applications with the library created by the author. In other words, given its stricter license clauses, GPL v2 should be used as the license when a conflict instance involves both Apache v2 and GPL v2.

VI. Threats to Validity

Threats to *internal validity* can occur in our training and test sets. To address this concern, we used five-fold cross validation, in which we trained, tested and tuned our ALP systems in different random splits of subject projects. In addition, threats to *external validity* can occur during data collection. To avoid such threats, we collected data for our experiments (i.e., changed source code files from different commits) across 700 open source projects of different types and domains from GitHub. The code-level licenses applied in changed files were distributed into 25 different kinds of licenses in different categories (i.e., permissive and restrictive).

VII. Related Works

A. Software License Identification and Classification

Techniques have been introduced to automatically identify and classify software licenses. Tuunanen et al. [3] proposed ASLA, a tool aimed at identifying licenses in FOSS systems. German et al. [4] proposed a tool called *Ninka* to identify license statements that takes as input text files and outputs license names and versions using a pattern matching approach. Di Penta et al. ([5], [19]) proposed approaches to automatically identify licenses of jar files via combined code search and textual analysis. Vendome et al. [6] applied a machine learning approach to detect exceptions of software licenses. Hoffmann et al. [20] analyzed actual license choices and correlated project growth from ten years of open source projects and discovered closed analytical models. Stewart et al. [21] found that business-friendly open source licenses had a correlation with project success. Alspaugh et al. [22] developed a meta-model to analyze the interaction of licenses from the viewpoint of software architecture. Mlouki et al. [23] investigated license violations and the evolution of these violations over time in the Android ecosystem. German et al. [24] detected license inconsistencies in code clones between Linux and other OpenBSD and FreeBSD. While all the existing works help developers to tackle the license identification problem, none of them proposed an approach or developed a tool to predict software licenses at file-level changes.

B. License Adoption and Evolution

Our research is also related to software license adoption and evolution. Di Penta et al. [19] studied licensing evolution on six open sources systems and found that license version and type changed during software evolution. Manabe et al. [25] studied licenses on FreeBSD, OpenBSD, Eclipse, and ArgoUML evolution and discussed characteristics of license evolution. German and Hassan [2] built a model to investigate specific licenses about applicability, advantages and disadvantages. German et al. [9] conducted an empirical study on binary packages of the Fedora-12 Linux distribution to understand and audit licensing consistency between packages and source files and claimed that it was challenging to audit licensing issues. German et al. [24] also studied the cloned code fragments between the Linux Kernel and two distributions of BSD and concluded that code migration was caused by additional restrictions of software licenses. Wu et al. [26] found that the license could be inconsistent among cloned files. Vendome et al. [27] conducted a survey with developers and found that facilitating commercial reuse was a common reason for license changes. They further investigated open source projects to gain insights of causes of license migration [1]. They found that licensing adoption and changes could be triggered by various factors. They also pointed out that there was a lack of traceability of when and why licensing changes were made. Almeida et al. [25] conducted a survey that posed development scenarios involving three popular open source licenses and found that developers struggled when multiple licenses were involved and developed a tool to recommend when adoption and evolution of license would be needed. Sen et al. [28] explored factors that affected the choice of a license for a project through analysis of open source project artifacts. These studies discussed the challenges and importance of predicting software licenses in software changes. However, none of the studies proposed a method or developed tools to predict licenses for software changes. To the best of our knowledge, we are the first to propose a coherent method and supporting tool to predict software licenses for software changes and provide a quantitative measure.

VIII. Conclusions and Future Work

Making appropriate selection of software licenses to adopt or update after software changes usually requires a great deal of experience and manual effort. To address this challenge, we annotated a large corpus of changed files with their licenses and developed ALP, a novel method and tool for automatic code-level license prediction for software changes. In an evaluation on 700 open source projects with a rich code change history, ALP2-Ranker1, the best variant of ALP, achieves an accuracy of 92.5% micro F1 score and 79.2% macro F1 score on licensed code changes, significantly surpassing the performance of three baselines, including a state-of-the-art license prediction system, *Ninka*. Future work will investigate additional impact factors in predicting licenses for software changes such as software code dependencies.

Acknowledgments

We thank the three anonymous reviewers for their helpful comments on an earlier draft of this paper. This work was partially supported by NSF grants IIS-1528037 and CCF-1848608. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views or official policies, either expressed or implied, of NSF. Xiaoyu Liu, LiGuo Huang, and Jidong Ge are the corresponding authors.

References

[1] C. Vendome, G. Bavota, M. Di Penta, M. Linares-Vásquez, D. German, and D. Poshyvanyk, "License usage and changes: A large-scale study on GitHub," in *Empirical Software Engineering*, vol. 22, no. 3. Springer, 2017, pp. 1537–1577.

[2] D. M. German and A. E. Hassan, "License integration patterns: Addressing license mismatches in component-based development," in *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 188–198.

[3] T. Tuunanen, J. Koskinen, and T. Kärkkäinen, "Automated software license analysis," in *Automated Software Engineering*, vol. 16, no. 3-4. Springer, 2009, pp. 455–490.

[4] D. M. German, Y. Manabe, and K. Inoue, "A sentence-matching method for automatic license identification of source code files," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2010, pp. 437–446.

[5] M. Di Penta, D. M. German, and G. Antoniol, "Identifying licensing of jar archives using a code-search approach," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 151–160.

[6] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk, "Machine learning-based detection of open source license exceptions," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 2017, pp. 118–129.

[7] J. Cohen, "A coefficient of agreement for nominal scales," in *Educational and Psychological Measurement*, vol. 20, no. 1. Sage Publications, 1960, pp. 37–46.

[8] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: The Kappa statistic," in *Fam Med*, vol. 37, no. 5, 2005, pp. 360–363.

[9] D. M. German, M. Di Penta, and J. Davies, "Understanding and auditing the licensing of open source software distributions," in *Proceedings of the 18th IEEE International Conference on Program Comprehension*. IEEE, 2010, pp. 84–93.

[10] "Qdox." [Online]. Available: https://github.com/paul-hammant/qdox

[11] "Licensefinder." [Online]. Available: https://github.com/pivotal/LicenseFinder

[12] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014, pp. 55–60.

[13] T. Lavergne, O. Cappé, and F. Yvon, "Practical very large scale CRFs," in *Proceedings the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, July 2010, pp. 504–513.

[14] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," in *IEEE Transactions on Information Theory*, vol. 13, no. 2, 1967, pp. 260–269.

[15] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," in *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, May 2011, pp. 27:1–27:27.

[16] D. A. Miller, "Significant and highly significant," in *Nature*, vol. 210, no. 5041. Nature Publishing Group, 1966, p. 1190.

[17] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," in *Psychological bulletin*, vol. 114, no. 3. American Psychological Association, 1993, p. 494.

[18] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohens d for evaluating group differences on the NSSE and other surveys," in *Annual Meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[19] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the evolution of software licensing," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 145–154.

[20] G. Hofmann, D. Riehle, C. Kolassa, and W. Mauerer, "A dual model of open source license growth," in *Proceedings of the IFIP International Conference on Open Source Systems*. Springer, 2013, pp. 245–256.

[21] K. J. Stewart, A. P. Ammeter, and L. M. Maruping, "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects," in *Information Systems Research*, vol. 17, no. 2. INFORMS, 2006, pp. 126–144.

[22] T. A. Alspaugh, W. Scacchi, and H. U. Asuncion, "Software licenses in context: The challenge of heterogeneously-licensed systems," in *Journal of the Association for Information Systems*, vol. 11, no. 11. Association for Information Systems, 2010, p. 730.

[23] O. Mlouki, F. Khomh, and G. Antoniol, "On the detection of licenses violations in the Android ecosystem," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1. IEEE, 2016, pp. 382–392.

[24] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 81–90.

[25] Y. Manabe, Y. Hayase, and K. Inoue, "Evolutional analysis of licenses in FOSS," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2010, pp. 83–87.

[26] Y. Wu, Y. Manabe, T. Kanda, D. M. German, and K. Inoue, "A method to detect license inconsistencies in large-scale open source projects," in *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 324–333.

[27] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. M. German, and D. Poshyvanyk, "When and why developers adopt and change software licenses," in *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 31–40.

[28] R. Sen, C. Subramaniam, and M. L. Nelson, "Determinants of the choice of open source software license," vol. 25, no. 3. Taylor & Francis, 2008, pp. 207–240.