Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

YANNIK SCHMIDT
BACHELOR THESIS

LICENSE CONFUSION ON GITHUB

Submitted on 22 July 2018

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 22 July 2018

License

Erlangen, 22 July 2018

Abstract

Today software ecosystems heavily rely on the re-usage of and object level linking to existing code. Allowance for this is given by a wide variety of software licenses. Due to conflicting restrictions imposed by those licenses, some of them cannot be used together in the same project.

This work approximates the amount of such license conflicts in the four big programming language ecosystems, Python, C++, Java and JavaScript, within the code sharing platform GitHub, using information from previous studies about code duplication on the platform.

In order to accomplish this, we offer an analysis of the compatibility of open source software licenses, as well as different methods of obtaining the necessary data set. We furthermore adapt, extend and evaluate existing license and text recognition methods to automatically recognize licenses in a highly scaleable fashion.

We find the amount of projects to be affected by license conflicts to be between 0.1 and 2.8 percentage. However the difference in numbers between the language ecosystem and the nature of the analysis method which was used in the previous study that produced the data we base our study on, indicates that the number of conflicts is actually much higher.

Contents

1 Introduction

1.1 Open source software licenses

In most countries, if you create source code or any sufficiently complex work, such work belongs to you and others may only use it given certain exceptions like, for example, fair use. In order to then allow other people to use your code more freely, you may license it to somebody. Most commonly in the case of open source software a project will contain a license file, effectively licensing it to everybody that acquires a copy. After that the code can be used, as long as all requirements of such a license are met. A common example of such a requirement is the inclusion of the license in any distributions containing the original work.

There is a wide variety of open source licenses in existence, however they roughly fall into two categories or rather a spectrum between two categories: permissive and copyleft. Permissive licenses aim to give users and developers the most freedom possible, having almost no restrictions. Copyleft licenses are designed to protect the source code from propertization, requiring that all derivative work of a given source must be licensed under the same license.

In addition to this, open source licenses may impose additional restrictions. Most notably they may forbid the usage of the projects name for advertisement[1], have certain patent regulations[2], or enforce the availability of the source code, even if you only interact with an application via a network in case the of the GNU Affero General Public License (AGPL)

Many licenses do not allow re-licensing source code under another license or the removal of the license or copyright notice. If one copies a work and then distributes it without thus license he is infringing copyright. Because of those restrictions, not all licenses can be used within one project.

[1]i.e. BSD 4-clause
[2]i.e. Mozilla Public License (MPL)

1.2 Compatibility of licenses

License A is called compatible to license B, if a project that uses an A-licensed work may be licensed under license B. The converse can be true, but is not implied. For example a GNU General Public License (GPL) licensed work may contain MIT[3] licensed work but not vice versa. Compatibility is not the same as relicenseability, meaning one still has to follow restrictions like including copyright notices and license text on certain files. Also "contain" and "use" can have different meanings depending on the license. For example the GNU Lesser General Public License (LGPL) will allow you to link to code licensed under its terms at object level as a library, without its copyleft aspect applying, while the GPL will not. This means that as long as a MIT licensed project only links to a LGPL library at object level, it may indeed still be licensed under the MIT license, despite the LGPL technically being a copyleft license.

Up to now, there does not exist a single big database of licenses compatibility. The Linux Foundation has a project called the "Software Package Data Exchange"[4] (SPDX) which lists a wide variety of licenses and the Free Software Foundation (FSF) lists many commonly used open source licenses and comments on their compatibility towards the GPL.[5,6]

[3] a very permissive license

[4] https://spdx.org/

[5] https://www.gnu.org/licenses/license-list.en.html

[6] all websites listed in this work were last accessed/rechecked on 07.22.2018

1.3 Git and GitHub

1.3.1 Git

Git is a distributed version control software (VCS) introduced in 2005[7] that is commonly used in the open source community.

1.3.2 GitHub

GitHub is a large code sharing platform. It enables developers, among other things, to version control and distribute their software. Code on GitHub is organized in repositories. A repository is a single git version controlled collection of any number of, usually thematically connected, files. With 31 million public repositories[8], it is likely the biggest data set of publicly accessible source code in existence. There is a payed service on GitHub which allows developers, to keep their work private. In this work we will only look at the the public code.

GitHub offers an API via to query certain meta-information about available projects. However, as of July 2018, this API imposes a rate limit of 5000 request per hour for non-paying users[9], making it tedious to query large amounts of data.

Files themselves can be obtained through the git protocol itself, by downloading it as separate files or by downloading an entire project as a compressed archive. GitHub states on its site that it allows using the site for research purposes under certain conditions that are described in section 5, "Scraping", in the terms of service. There appears to be some sort of rate limit concerning HTTP-request. Request will routinely fail with code 429, "Too many requests". The exact value or the inner working of this rate limit remains unclear. While sometimes requests were failing every few seconds, there were also entire days without a single failure even though requests/second were very constant.

1.3.3 Licensing on GitHub

It's important to understand, that GitHub doesn't force developers to license their Code under an open source license. Uploading your Code to GitHub, requires you to accept the GitHub terms of service which in turn means, that you effectively licensed your Code under a license that allows people to a) fork it b) read it.

[7]https://git-scm.com/
[8]July 2018, according to https://github.com/search?q=is:public
[9]https://help.github.com/articles/github-terms-of-service/

You are encouraged by a message in the web interface to choose an open source license and if such a license is chosen, GitHub will display it above the project and also make this information available through the API.

1.4 GHTorrent

The “GHTorrent data set and tool suite“ (Gousios, 2013), in essence, is an effort to circumvent the rate limit of the GitHub-API by mirroring all information into a querieable database. Normally every user can do 5000 requests per hour, based on tokens that can be created within the web interface. This means researchers can, while technically against the GitHub terms of service[10], share their authentification tokens with GHTorrent. This enables them to pull significantly more existing information from the GitHub API than a single researcher could. Not all API information are mirrored yet and unfortunately the license information field is among those missing.

The GHTorrent website itself states that their data set may contain errors.[11] Most notable are duplicated projects, but given the changing nature of GitHub in general we have to expect outdated data and consider, that, due to the size of the data set, erroneous data might go unnoticed.

The GHTorrent project offers its data through a MongoDB, a no-SQL database, by means of an SSH based connection, for which they have to accept your key. According to their own statement, their acceptance process should take one week, however in our case it took over one month. After the key was accepted, their database wasn't accessible for another month.

While the project does offer a complete dump of the MongoDB by means of an incremental backup, running a database of such size would require excessive hardware which is not at our disposal. Given this dependency on external infrastructure we have sought to minimize our reliance on the GHTorrent project, as it appeared hugely unreliable in it's availability during our work.

The MongoDB is only indexed by certain fields, meaning it can only be selectively queried by those fields. Luckily, we will only have to query based on the project name, id and owner, which all happen to be indexed fields, or query all projects or entries in a specific table which we can achieve by simply using an iterator over all existing projects.

Making small queries will make the overall querying slower, but GHTorrent is inconsistent and quite unforeseeable in its response times (likely dependent on the amount of people using their server at a given time) and they impose a 1000ms limit for any query to finish which does not allow for complex or big queries. There exists a MySQL-Database with additional index structures, but due to the described nature of our queries we will not need to use it.

[10]https://help.github.com/articles/github-terms-of-service/
[11]http://ghtorrent.org/faq.html

1.5 Dejavu

The Dejavu project (Lopes et al., 2017) analyzed source-code on GitHub to find non-forked[12] duplicates. It found that between 9 and 31 percentage[13] of projects are more than 80% similar. It was further discovered that most of these duplicates are either automatically generated code, examples, or, especially in the case of JavaScript, libraries copied into projects. The study utilized the source code focused clone detector tool “SourcerCC“ (Sajnani, Saini, Svajlenko, Roy & Lopes, 2015).

[12]for an explanation of “forking“ see section 1.6.3

[13]9% of all Java projects, 16% for C++, 11% for Python and 31% for JavaScript

1.6 Clarification of terms used

1.6.1 Cloning

To “clone“ a project, means to copy it with it’s complete history of changes. This is the primary mechanism by which one may copy projects maintained with git in general. The Dejavu project uses the word “cloning“ describing what we will call “copying“. Cloning has a very specific meaning in the context of git and we believe their usage of the word is misleading.

1.6.2 Copying

We will use the word “copying“ from now on to describe code that was duplicated by any other means, for example by downloading single files or even cloning the projects to another device and then re-uploading it. Both lead to GitHub tracking such files or projects as individual and unrelated. These kinds of copies, are the ones Dejavu analyzed.

1.6.3 Forking

Forking, though sometimes used synonymously with cloning, in this work, will refer specifically to the process of cloning a project within GitHub using the GitHub API or webinterface. In such a case, GitHub has a direct way of knowing if a give project originated from another one.

1.6.4 Detection certainty

With classification in general it is common, to not only let an algorithm return a result, but also a certainty factor on that result. A higher abstraction level can then decide if it shall propagate such information or simply discard information under a certain threshold. This is not to be confused with the “relation“ field in the Dejavu result data. This field describes the similarity between two files, or project repositories and is therefore an absolute statement about this similarity and not a certainty value, though we will use it in a similar fashion to discard projects that aren’t similar enough.

2 Research

2.1 Research goal

We want to find out how prevalent license conflicts are in the open source ecosystems of the four widely used languages Python, C++, Java and JavaScript and of what kind they are. For this we must create a highly efficient pipeline that can parse huge amounts of data in an acceptable amount of time.

This problem consist of three smaller problems, which all have to be solved:

- acquirement of the data
- recognition of the licenses
- comparison of the licenses of cloned or copied projects to find conflicts

2.2 Outline of the pipeline

The analysis pipeline we use has six stages in total. The main stages are:

Stage 1	-	data collection
Stage 2	-	license recognition
Stage 5	-	conflict analysis

with three additional stages called:

Stage 3	-	alias generation
Stage 4	-	alias application
Stage 6	-	plotting

The generation and usage of so called “aliases“ is designed to evaluate differently (possibly already existing) detection implementations that produce different formatted outputs or map similar licenses onto one (for example aliasing “GPL version 3 or any later version“ and “GPL version 2 or any later version“ to “GPLv3“). This step is not strictly necessary, but it detaches the output of stage two from the workings of stage five. Plotting can automatically generate graphs based on the output of various stages using pandas[1] and gnuplot[2]. None of these second set of “convenience“ stages will be discussed any further.

Figure 2.1 illustrates the pipeline and the information and values certain stages are able to consume and output.

[1]a Python library
[2]a standard Linux plotting tool

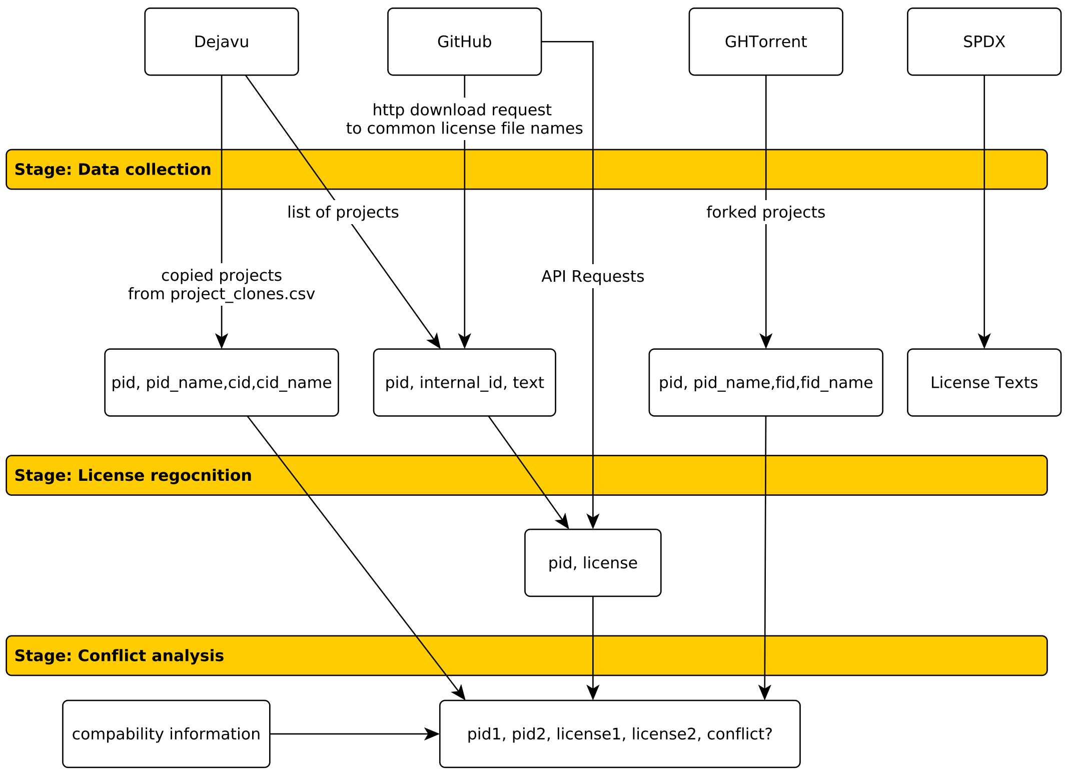


Figure 2.1: Overview of the pipeline.

2.3 Data acquisition

2.3.1 Dejavu data

The Dejavu project offers its refined data set on its website. Documentation of the exact format (meaning the values of the columns of their CVS-formatting), unfortunately, is poorly outlined and information is scattered throughout their paper and artifacts. The names of the columns are often misleading, so we choose to change the naming scheme to make the meaning of certain values more clear. Another problem related to this is the creation of artificial "id" keys for use during the projects pipeline when interacting with a SQL database, instead of using a joint key over multiple fields. These id (basically enumeration) fields are described in the documentation, but are only present in the Java and JavaScript results. It is important not to confuse the 'id' field with the 'project id' field.

project_clones.csv	projects.csv
enumeration (Java only)	project id
cloned project id	name of project archive
number of copied files in cloned project	project URL
total files in cloned project	
percent of copied fields in cloned project	
origin project id	
number of affected files in origin project	
total number of files in origin project	
percent of affected files in origin project	

Figure 2.2: Columns in relevant CSV formatted files as described in an explanation of the project's analysis pipeline within a virtual machine that is provided on their website to test thus pipeline.

2.3.2 GHTorrent data

The Dejavu data only covers copied projects, however we might also want to examine forked projects in the future, so we have to query the GHTorrent database ourselves. As a result, we will get another CSV file with the the projects id, name and the forked project's id and name.

Performance of querying this data varies as is expected when dealing with a database concurrently accessed by a lot of uncoordinated users. The average request speed, is about 30.000 successful queries per hour on a 100Mbit/s symmetric port with a script, that would back off for 1-10 seconds after a failed request. Requests only fail extremely rarely (1 in 100 000) so we can chose a long backoff time just to be safe. After that we have to filter out duplicates. We found 1.2% of all projects are duplicates in the GHTorrent data set. The Dejavu project found 1.6% duplicates on their subset limited to Python, Java, JavaScript and C++. The people behind GHTorrent advise in a bug report ("issue") we created, to simply use the newest available entry and ignore all others.[3] This information does not appear to be anywhere in the projects documentation itself.

2.3.3 GitHub data via HTTP

We do not have the resources to do the same as the Dejavu project and download/store the entire content of GitHub (especially to store it). However we can look for filenames in the top directory. Table 2.1 shows the filenames, most commonly containing license information. Though more could be added, each one would be another HTTP-request per project, notably adding download time due to the sheer number of projects.

LICENSE	LICENSE.rst	LICENSE.md	LICENSE.txt
COPYING	COPYING.rst	COPYING.md	COPYING.txt

Table 2.1: Common filenames of files with license information. ".md" and ".rst" are the standard extensions for the common markup languages "Markdown" and "reStructuredText".

2.3.4 GitHub data via API

GitHub has an API that contains license information for some projects which GHTorrent hasn't copied yet and has not stated intention to [4]. The main problem with this is the API rate limit which we talked about in the introduction. But there is another problem: during our own work we realized how easy it is to fail with license recognition. People on GitHub don't just paste in a license text, but rather mess them up in a variety of ways, ranging from just adding a newline or changing format, to adding Unicode characters and including information about the project on top of the license file.

[3]https://github.com/ghtorrent/ghtorrent.org/issues/463
[4]https://github.com/ghtorrent/ghtorrent.org/issues/488

GitHub's only information about their own effort of license recognition stems from a 2015 blog post[5]. It states that they use an implementation written in ruby called *"licensee"*[6]. Apart from that, the entire process is a black box. Also only the top level license is provided, so using GitHub's own data would make our approach difficult to extend in the future.

Using the data collected by GitHub would be negligent in other ways too. Neither does the API provide a confidence factor concerning the recognition, nor does GitHub state which certainty threshold they used, if any. It is most likely that they used the implementation's default threshold of 0.95. However our own tests show that most licenses even with as low 80% similarity, are in fact correctly recognized. This means that, from a statistical point of view, we would be missing a significant data set. This is especially important, because the subset of people failing to add licenses properly, might indeed be quite different.

Most unfortunately, reading the license identification requires two API requests. Only querying the 900.000 Python projects listed in the Dejavu files took over two weeks, meaning that querying the remaining 3.6 million projects could be done within a month. This is beyond the time limits of this work, but within the realms of practicality.

2.3.5 Raw license texts

The thing that comes closest to an open source licenses database is SPDX. It does not offer a single download for all the licenses, so the texts have to be scrapped from their page. We will thin out the amount of licenses manually to those listed in section 2.5.3 for reasons we describe in the upcoming sections.

[5]https://blog.github.com/2015-03-09-open-source-license-usage-on-github-com/
[6]https://github.com/benbalter/licensee

2.4 License recognition

There are existing solutions to identify licenses, notably the ruby gem “licensee“ GitHub claims to use, the “licensecheck“ package found in Debian that performs extraordinarily badly and a not commonly known Golang/Python implementation from Ben Boyter [7]. The best performing identification in the pipeline uses Mr. Boyter’s original Python code, rather than his later go implementation, mostly because it was more well structured, documented and easily adaptable.

If we analyze the problem, keeping in mind our extreme performance needs, we can make the following observations:

- We only have to compare our files to a limited set of reference files, namely the licenses we got from SPDX. That means we are looking for very specific patters rather than for “all“ patterns as the Dejavu projects did. This can be translated into a performance gain.
- Hashing with a fast collision avoiding algorithm like SHA-1 would be great, but as stated above, there is a huge number of slightly different “versions“ of each license. We would need to hash each version once first, in order to identify them based on the hash again later.
- We could alternatively hash each file with a locality sensitive hashing algorithm. The performance of this will have to be evaluated, as many licenses have similar parts, which may confuse a locality sensitive hashing algorithm.
- We could search for unique substrings in a sample set of licenses and then check each file for this string.
- We could search the head of files for keywords like GPL or MIT.
- Since we are doing a statistical analysis about license conflicts, we can be slightly more relaxed towards falsely identifying licenses, as long as those misidentified licenses are similar in nature. It would not be a big problem, if we identify one permissive license as another.

2.4.1 Character or word distance

In terms of pure recognition rate, this approach performs very good. It is less susceptible to the problematic disturbing factors with simple and locality sensitive hashing, even if there is a long text behind the actual license. However a distance by its very definition means, that it has to be computed between all

[7]https://github.com/boyter/python-license-checker/blob/master/parse.py

characters/words in the analyzed file and all possible licenses, making it almost a brute-force like analysis method and anticipatedly too slow to be used with a large data set. The ruby implementation used by GitHub itself works along the lines of this principal.

2.4.2 Locality sensitive hashing

Locality sensitive hashing (LSH) reduces the dimensions of a text like normal hashing, but unlike normal hashing the algorithms are designed to return similar hashes for similar input strings. One can imagine each letter in given text as a dimension. So "hello_world" would have ten dimensions and a hypothetical LSH algorithm might reduce it to "phX" (three dimensions). This works mostly well, however the impact of the disturbance factors mentioned above is still clearly visible. Especially text before or after the actual license confuses this method greatly. The LSH variant used here is context triggered piece wise hashing, (Kornblum, 2006) implemented by the ssdeep library[8], using Python function bindings.[9]

2.4.3 Unique subsequence

This method parses all license texts first and tries to find a unique sequence of words within each of the licenses. Once created, all unique sequences are in essence nothing more than a regular expression, which the text can be searched for. This is not as fast as hashing, but much faster than the character or word distance based approach and yields similar results as the latter.

2.4.4 Unique subsequences with normal hashing

Figures 2.3 and 2.4 show, that our unique sub sequence search performs well in terms of recognition but simply isn't fast enough to scan a huge data set with it. To fix this, we revisit the hashing idea. If we simply hash each file as we encounter it, we can check for an existing hash first and only if we can't find one, fall back to looking for subsequences. This is essentially an optimizing with dynamic programming.

[8]https://ssdeep-project.github.io/ssdeep/index.html

[9]Python bindings are a way of interacting with code written in different programming languages using normal Python code

2.4.5 Unique subsequences with LSH and normal hashing

One of the problems of LSH as a recognition method, formerly, was the problem of efficiently classifying a large data set of hashes and re-finding similar ones. Now we already have a mechanism to create a large number of hashes of identified licenses. This means we can be much more lazy with the recovery of those hashes. In fact our implementation only hashes texts one and two times, respectively and puts them into sorted lists. It should be noted at this point that the Python ‘sorted‘ list is heavily optimized for inserting elements into an already sorted list.

Because looking up and comparing hashes in this index structure is cheap, we can even compare multiple neighbors and still have an overall performance boost because we have to do less, expensive, unique subsequence searches. Using this simple memorization structure, we are able to find more than 95% of files within our hashed list, which is a 5 points increase from normal hashing. This approach will not scale indefinitely, above a certain number of entries it will be outperformed by a more complex memorization structure. However we did never reach this point during this work.

2.4.6 Keywords instead of unique subsequences

Unique subsequences are great for longer texts, but in future we, not only, want to analyze license files but normal source code files as well. Those files may only contain a small license header, with a long tail of source code behind it. While the license headers suggested for use on SPDX would be parse able by the above method, there seems to be a huge number of files that only contain a single line like *“This software is licensed under the terms of the GPL version 3“*. This would break the subsequence approach.

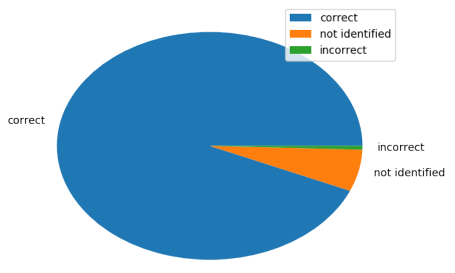
Keyword search is error prone when used with full files, so in the implementation we only check the first few lines. Also, some licenses simply do not have a keyword we can look for in the actual license text. For example *“Licensed MIT“* is a common header and could be searched for, but the string *“MIT“* does not occur in the MIT license itself.

Running our test set with this identification produces bad results on first sight, however, if inspected closely, most of the misidentified licenses are GPL identified as AGPL (because of GPL §13 containing the keyword AGPL), or any very permissive license as any other. The GPL problem won’t occur when only a few lines are inspected and the confusion of permissive licenses doesn’t affect the number of conflicts, as explained at the beginning of the section.

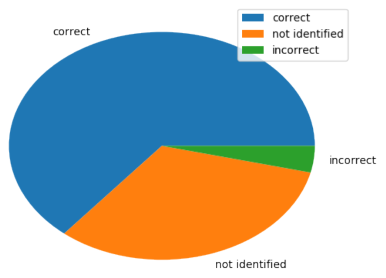
The keywords used are listed in table 2.2.

License	Keywords
GPL	GPL, GENERAL PUBLIC LICENSE, !AGPL, !LGPL
MIT	MIT
BSD	BSD
LGPL2	LESSER GENERAL PUBLIC LICENSE, LGPL
CC0	CC0
EFL	Eiffel Forum License, EFL
Apache	Apache
MPL	Mozilla Public License, MPL
AGPL	AFFERO GENERAL PUBLIC LICENSE, AGPL

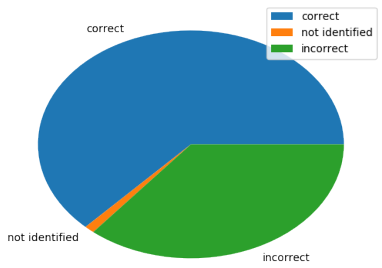
Table 2.2: Comma separated lists of keywords. Negation is symbolized an exclamation mark. License will match if any of the non negated and none of the negated matches.



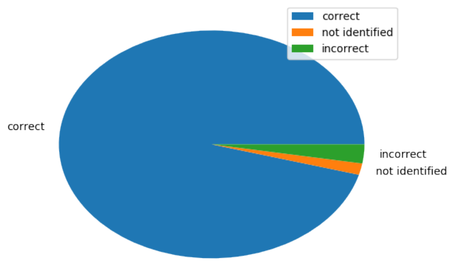
(a) unique subsequence search



(b) “licensecheck“ package

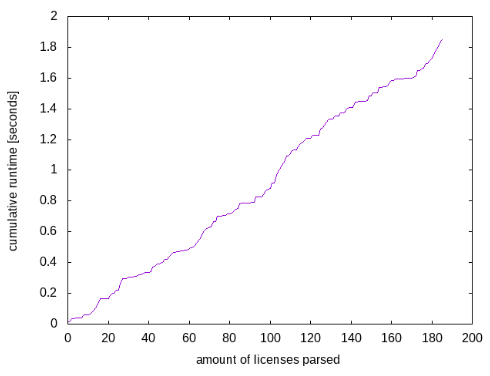


(c) keyword search with LSH and normal hashing

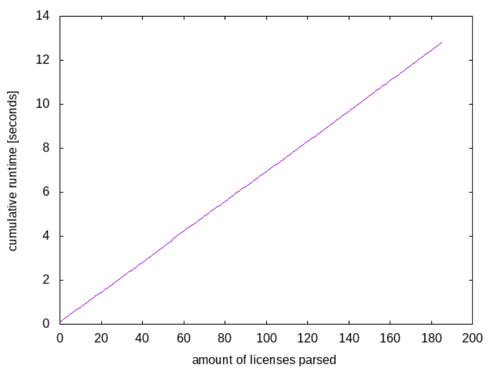


(d) unique subsequence search with LSH and normal hashing

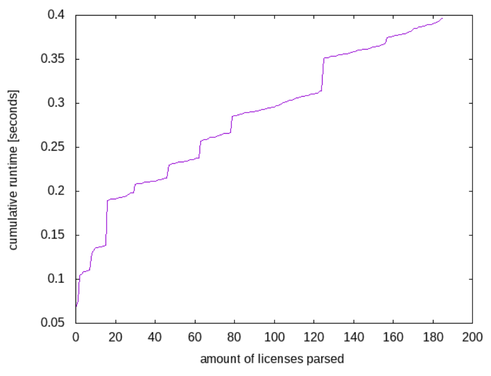
Figure 2.3: Recognition performance of different recognition methods.



(a) unique subsequence search



(b) keywords with normal hashing



(c) unique sub sequence approach with LSH and normal hashing

Figure 2.4: Runtime performance of different recognition methods.

2.5 Conflict recognition

2.5.1 Compatibility of analyzed licenses

Determining the exact compatibility of licenses is far beyond the scope of this work. There may even be different situations in different countries, depending on their respective copyright laws. The goal of this work is to provide an estimation of license conflicts and not an exact value. Therefore we will only count license conflicts which are certain beyond reasonable doubt - or, in case of compatibility towards GPL, if the FSF thinks that a license is GPL compatible. The FSF itself states, that ultimately it cannot give real legal advice, but its opinion, especially on its own licenses, is likely among the most reliable one that exists.

2.5.2 Compatibility rules

Our algorithm for determining compatibility works as follows, with “B“ being the included/copied project.[10]

```
if PROJECT B has no license:
    return CONFLICT
else if LICENSE A == LICENSE B:
    return NO CONFLICT
else if LICENSE A is restrictive and LICENSE B is permissive
    return NO CONFLICT
else if LICENSE A is protective and
    ( LICENSE B is GPL-compatible or LICENSE B is permissive )
    return NO CONFLICT
else:
    return CONFLICT
```

[10]With the exception of the check for the lack of a license, this is implemented exactly the same way in the function “is_compatible“.

2.5.3 Overview of analyzed licenses

license	**type**	**copyleft**	**sub license as**	**GPL compatible**
MIT	permissive	no	all	yes
ISC	permissive	no	all	yes
unlicense	permissive	no	all	yes
MIT/X11	permissive	no	all	yes
MPLv1	permissive	no	none	yes
MPLv2	permissive	no	none	yes
EPLv1	permissive	no	none	yes
zlib	permissive	no	none	yes
zlib/libpng	permissive	no	none	yes
CC0	permissive	no	all	yes
PD	permissive	no	none	yes
WTFPL	permissive	no	none	yes
PSF 2.0	permissive	no	none	no
BSD 0	permissive	no	all	yes
BSD 1	permissive	no	none	yes
BSD 2	permissive	no	none	yes
BSD 3	permissive	no	none	yes
BSD 4	permissive	no	none	no
Apache v1	permissive	no	none	no
Apache v2	permissive	no	none	yes
EFL	permissive	no	none	yes

Artistic	permissive	no	GPL	no
CeCILL	permissive	no	GPL	yes
GPL	protective	yes	none	yes
LGPL	protective	yes	none	yes
AGPLv1	network protective	yes	none	no
AGPLv3	network protective	yes	none	yes
Perl	dual license	no	GPL	yes
any non CC0 Creative Commons	restrictive	unclear	none	no
no license	restrictive	n/a	none	no

Some explanations

For Creative Commons (CC) licenses other than CC0, the FSF and Creative commons themselves agree, that they aren’t suited for code and that the wording cannot be clearly interpreted when dealing with software. This is why we declare it as restrictive and de facto incompatible to non-permissive licenses.

If a project has no license, then nobody is allowed to use it. This mean “no license“ is restrictive. Our implementation may sometimes treat this as a special case. Just because we didn’t *find* a license, doesn’t mean there *isn’t* one.

“Perl“ refers to a dual license that allows a software to be used under the terms of the Artistic License or the GPL.

BSD 1-4, meaning BSD with 1, 2, 3 or 4 clauses, are variations of the BSD 0-clause license, containing additional requirements and restrictions.

“v1“, “v2“ or “v3“ symbolize respective versions of licenses. All other abbreviation can be found in the acronym section of the appendix.

2.6 Results

2.6.1 Detection and recognition rate

Figure 2.5 shows the success rate of our license search and recognition. It's higher than what GitHub itself thought it to be in 2015.[11] This does not necessarily mean that there are actually more licensed repositories now. The Dejavu project already filtered out any projects that did not fall into it's four big language ecosystems and it seems reasonable that this already excluded a lot of unlicensed micro projects.

For Python and C++ we found the most licensed projects, followed by JavaScript and lastly, with only half as much licensed projects, Java. Recognition rate was good across all languages, with the amount of unidentifiable license lying below or around 1%. These unidentifiable licenses were mostly empty files and files that contained too much noise and therefore fell under the certainty threshold for recognition. This for example occurred when documentation of the project was written into the license file. A small number of files appeared indeed to be licenses of some sort, but were unable to be found outside of the project itself.

	Python	C++	Java	JavaScript
none found	664994 (73%)	278315 (75%)	1361427 (91%)	675703 (79%)
identified	235015 (26%)	87011 (23%)	115125 (8%)	166118 (19%)
not identified	9281 (1%)	4114 (1%)	4916 (< 1%)	4633 (< 1%)
total	909290	369440	1481468	846454

Figure 2.5: Success rates of license search (stage 1) and recognition (stage 2).

Checking recognition results by hand, we found a number of misidentified licenses. However these misidentification were very localized and do only affect small permissive licenses. The specific problems were:

- Apache version 2 (header only) misidentified as unlicense
- Apache version 2 (header only) misidentified as WTFPL
- BSD 4-clause misidentified as BSD 3-clause
- ISC misidentified as unlicense

The most important impact of this to our following statistics will be, that the number of occurrences of the "unlicense" and the "WTFPL" are strongly inflated. The number of conflicts will stay unaffected because both licenses are permissive.

[11] https://blog.github.com/2015-03-09-open-source-license-usage-on-github-com/

2.6.2 Number of conflicts

JavaScript had the highest portion of conflicts in relation to the number of projects closely followed by C++. Python was significantly lower and Java was extremely low. These numbers are influenced by the amount of copies recorded in the Dejavu data set. The relation of conflicts to amount of recorded copies shows more similar numbers for Python, C++ and Java. To make the results comparable, the total amount of copies for JavaScript has been halved in table 2.3, because we also only parsed about half of the projects.

	Python	C++	Java	JavaScript
conflicts	12272	9745	1892	23280
total projects	909290	369440	1481468	846454
total copies	27362564	35925821	10169471	130000000
conflicts/projects	1.34%	2.64%	0.13%	2.75%
conflicts/copies	0.045%	0.027%	0.019%	0.017%

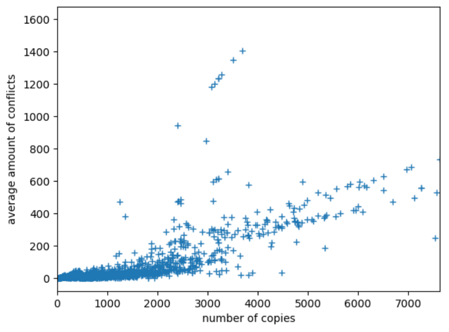
Table 2.3: Amount of conflicts in relation to amount of projects and amount of copy-relations per language.

2.6.3 Correlation of copies and conflicts

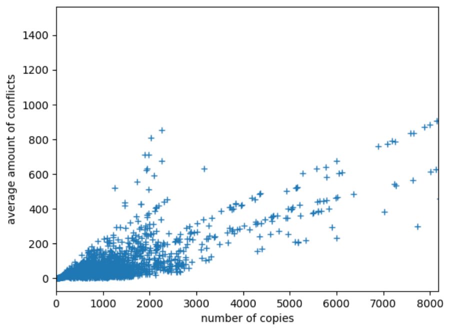
There is expectedly a strong correlation between the amount of copies and the amount of conflicts a project has, which can be seen in figures 2.6 and 2.7. Again C++ and Python look quantitatively similar and have a linear increase of conflicts with an increasing amounts of conflicts. Java is different from the rest, mostly because the most copied project only has 2000 copies, compared to the over 7000 copies for the biggest C++, Python and JavaScript projects. However apart from that, it also has a nearly linear increase.

JavaScript has a steep increase in conflicts starting, from 3000 copies and upwards. Most of the dots behind this point appear to be versions of common libraries. The overall amount of conflicts is still much lower than in the Python or C++ ecosystem. This is likely because permissive licenses are more common in JavaScript (figure 2.8).

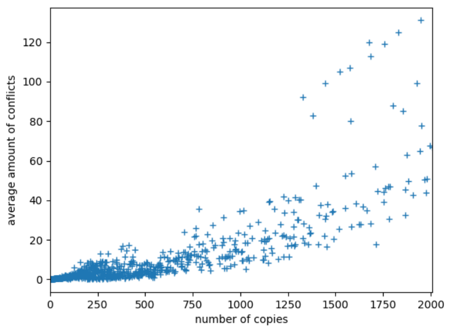
The apparent clustering of points on certain probability levels in figure 2.7 is a statistical artifact that exists because the graph is essentially a histogram with bucket size one. So it is way more common to have "buckets" with zero, one or two projects and therefore probabilities of 0.0, 0.5 and 1.0.



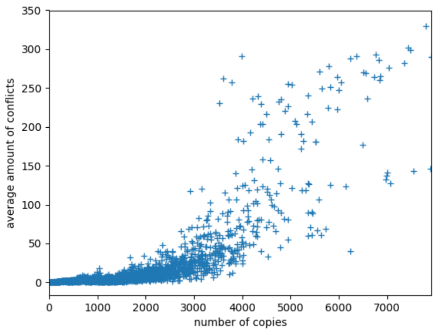
(a) Python



(b) C++

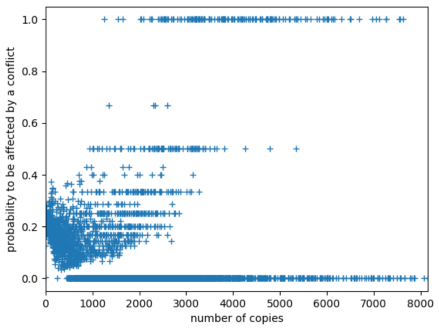


(c) Java

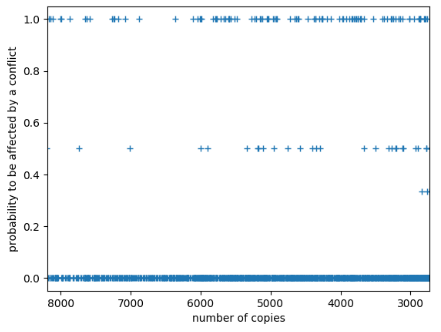


(d) JavaScript

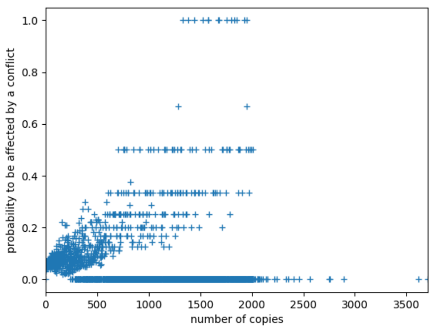
Figure 2.6: Average amount of conflicts for a given amount of copies.



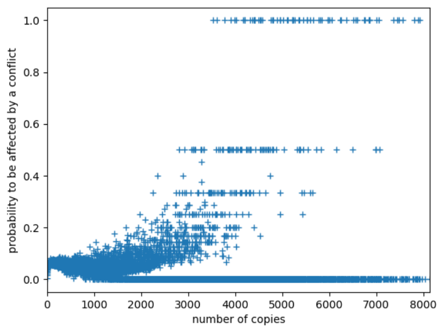
(a) Python



(b) C++



(c) Java

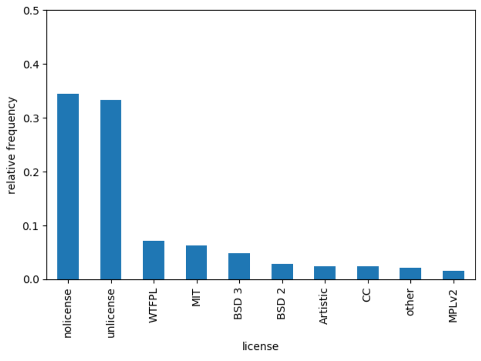


(d) JavaScript

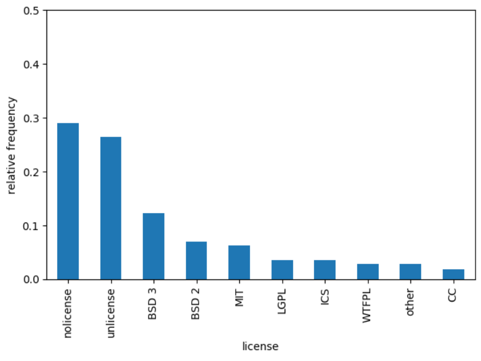
Figure 2.7: Portion of projects with a given amount of copies that is directly affected by a conflict.

2.6.4 Nature of the conflicts

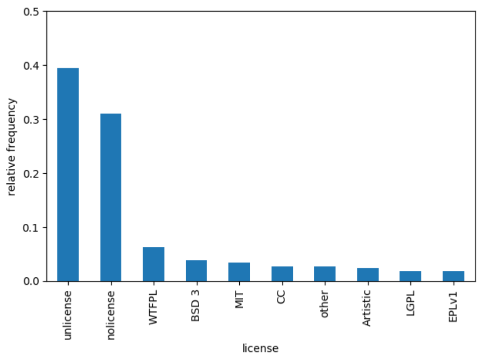
Figure 2.10 shows which licenses were violated the most. Java is missing because the only license violated was the GPL. Figure 2.9 shows what license the projects violated the former with. The occurrences of conflicts are mostly tied to how restrictive that license is and how often it is detected in the first place (see figure 2.8). Lastly, figure 2.11 shows the most common violating pairs.



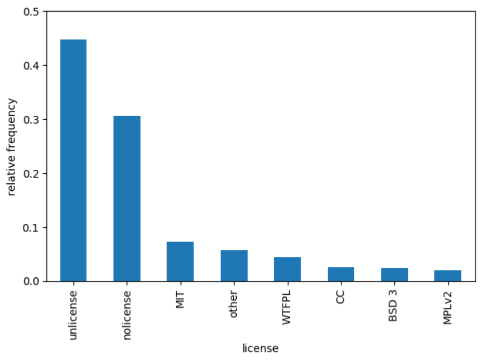
(a) Python



(b) C++

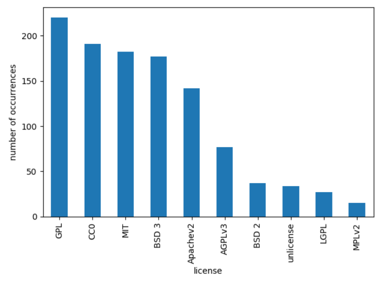


(c) Java

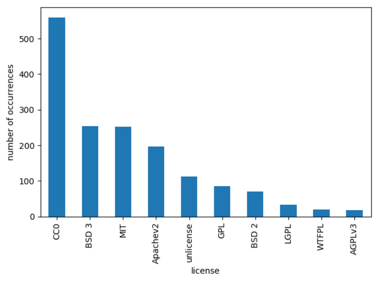


(d) JavaScript

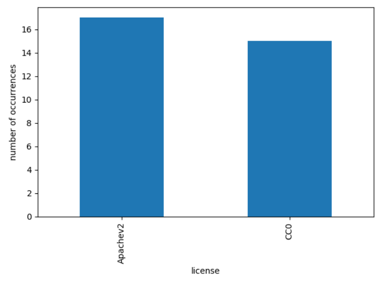
Figure 2.8: Most detected licenses.



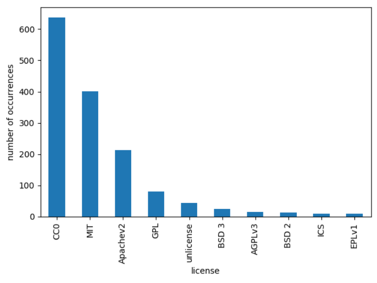
(a) Python



(b) C++

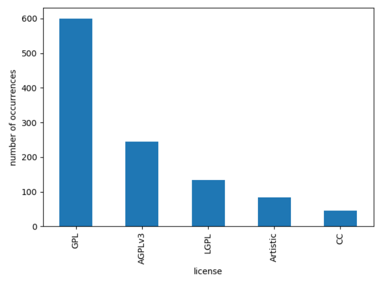


(c) Java

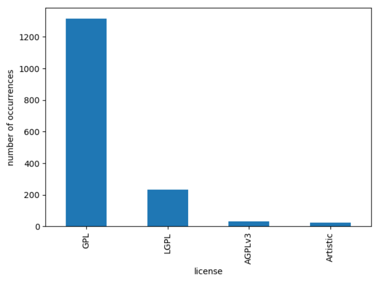


(d) JavaScript

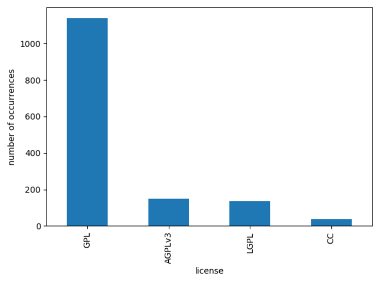
Figure 2.9: Violating licenses.



(a) Python

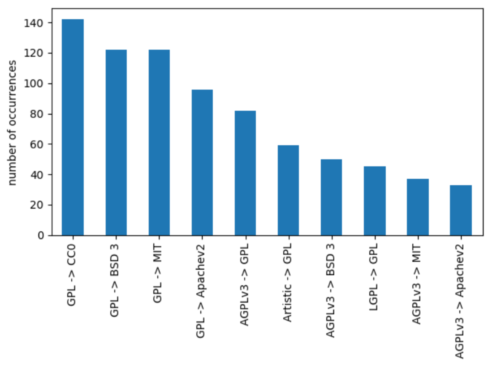


(b) C++

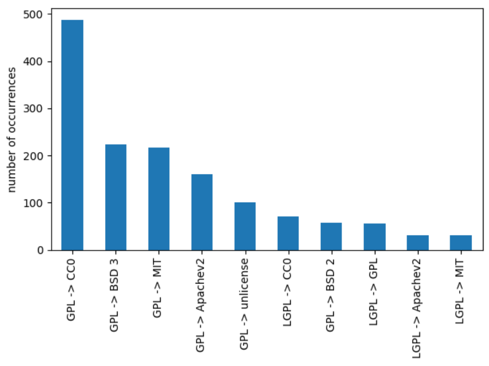


(c) JavaScript

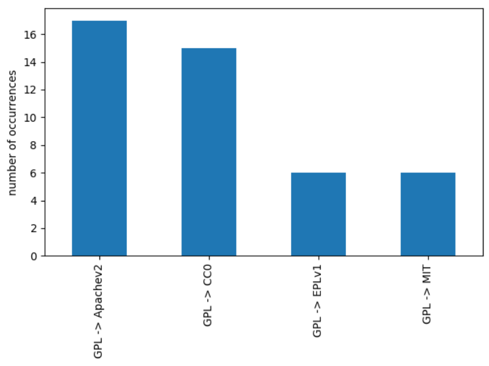
Figure 2.10: Violated licenses.



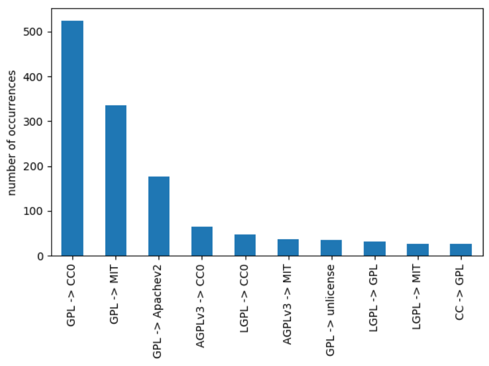
(a) Python



(b) C++



(c) Java



(d) JavaScript

Figure 2.11: Violating pairs (violated license → violating license)

2.6.5 Indirect conflicts

License conflicts are transitive in a sense, that a project using a project that contains a license conflict has a license conflict itself. If we factor this circumstance into our results, the amount of conflicts increase by 50-70% depending on the language.

	Python	C++	Java	JavaScript
direct conflicts	12272	9745	1892	23280
indirect conflicts	7054	4325	1148	15889
cumulative conflicts	19326	14070	3040	39169
cumulative conflicts/projects	2.1%	3.81%	0.21%	4.63%

Table 2.4: Indirect and cumulative conflicts per language

2.7 Result discussion

2.7.1 Impact on the individual project

The numbers in tables 2.3 and 2.4 clearly show that the problem of license conflicts, even when only looking at top level licenses, is definitely existent.

Table 2.4 furthermore shows that, when factoring in indirect license conflicts, the difference in amount of conflicts in Java, in relation to the other language ecosystems, increases. In other words: the cumulative number of direct and indirect conflicts increases proportionally to both: the amount of copies and the amount of conflicts.

In turn this means that once other forms of copying like forking or cloning and the inclusion of libraries are factored in, the amount of conflicts is expected to increase further, based on the average number of licensed works included in a project. The few percentages of direct conflicts found by analyzing the Dejavu data set, should therefore not be dismissed as an insignificant amount.

2.7.2 Languages specifics

The low amount of conflicts in Java can be explained by the low number of copies. Table 2.3 clearly shows that, the ratio of conflicts to copy-relationships is comparable. It's just 50% lower than the Python value while the conflict to project ratio is 100 times lower.

Python and C++ seem comparable in their numbers. While C++ does have more conflicts per project, it has less per copy-relationship. In other words, despite C++ having significantly more copied projects, it has less overall conflicts. This is likely due to the higher prevalence of permissive licenses in the C++ ecosystem as can be seen in figure 2.8.

JavaScript has the highest conflict to project ratio, comparable to C++, but at the same time the lowest conflict to copies ratio, which is more comparable to the Java percentage. The Dejavu paper (Lopes et al., 2017) found that a lot of JavaScript copies are detected, because of node-js package manager (NPM) libraries included as files. This is the main way that projects are interacting, even though, in the case of the other languages, it is very uncommon to actually include the files rather than referencing them some other way. JavaScript has almost a factor of ten more copies recorded in the Dejavu data set. However we observe the conflict to copies ratio to be in the same order of magnitude as the values for the other languages. This indicates that the amount of projects with conflicts scales linearly with the amount of detected copy or library inclusions.

If the number of conflicts does scale linearly with the amount of other projects included, it would make the number of conflicts for JavaScript surprisingly low. This has two reasons. Firstly, figures 2.10 and 2.9 show that GPL, a copyleft license, responsible for a lot of conflicts in the other ecosystems isn't among the commonly used licenses for JavaScript. In fact all of the most used licenses are permissive. Secondly, while the detection rate for JavaScript isn't higher than for the other languages, the detection rate of the most commonly copied projects actually is higher. This means there are less, frequently copied, projects for JavaScript that are identified as having no license, additionally reducing the number of conflicts.

2.7.3 Threats to validity

False legal interpretation of licenses

While we have tried our best to interpret the license texts of our analyzed licenses, primarily by using the statements of the license creators themselves and the FSF, this would be work for a legal expert. Some clauses might not even be allowed in some jurisdictions or have be interpreted differently.

Unchecked license requirements

We have given each project the benefit of the doubt that it has followed any additional requirements of licenses, that would be impossible or extremely difficult to automatically detect. Example for this include, but are not limited to:

- requiring to include a "brief summary" of changes (i.e. Python Software License)
- usage of trademark names (i.e. MPL)
- inclusion of a *"NOTICE"* file (i.e. Apache license)
- all regulations regarding patents
- usage of the name of project for advertisement (i.e. BSD 4-clause)
- actually shipping code or license information with potentially distributed binaries
- network protective requirements (AGPL)

If any such requirements are not full filled, it would mean that license is null, either forever or until these problems are corrected, depending on the license.

Semantic of multiple licenses

Multiple licenses, especially within one file can have semantics that aren't really parsable, or we cannot differentiate. Table 2.5 shows the most common ones, with last being an example from the massive project *ffmpeg*.[12]

"License A applies to the documentation, license B applies to the code.".
"Project is dual-licensed under license A and license B.".
"License A applies if compiled, with the [...] flag, otherwise license B applies."

Table 2.5: Common examples of multiple license with an unparsable semantic.

We do not see any way to reliably find and understand such semantics, let alone within this work. This is why, as stated before, projects with more than one license per file are treated differently. As our manual investigation seems to suggest that dual licensing is very rare, we have only checked wherever the two licenses are compatible according to our compatibility algorithm. Therefore all dual licensed projects have the potential to add an error to our estimates.

Reduced license set

We mapped some licenses to other licenses. Most notably we did not differentiate between GPL version two and three or any of the non public domain Creative Commons licenses, even though our license system could recognize them individually, to keep our conflict analysis more simple and avoid the scattering of similar conflicts into subcategories. This is very unlikely to introduce any meaning full error to the analysis at hand but should be considered if the implementation provided is ever used for something else.

Project of origin

We assume that the first (oldest) project is always the project of origin. While this assertion holds true in the majority of cases, the Dejavu study (Lopes et al., 2017) already pointed out some projects for which the oldest project is not the project of origin. This happens, for example, if the original project was hosted on another platform before eventually migrating to GitHub. One might be able to find a semantic like the amount of "stars" (GitHub's "liking" system) or amount of commits but this would require a lot of manual examination.

[12]https://www.ffmpeg.org/

Missed license information

Neither will we detect, nor is there any reasonable method for doing so, license information in source code files. While this is not common practice, and the legal binding of it would be unclear because hiding a single line of license information within a long file might be incompatible with laws like §305c BGB[13]. An example of such practice would be the distributed CSS framework “SemanticUI“[14]. For our work, we assumed that there is no license, if none was found. This could have inflated the number of conflicts, since lack of a license is basically the most restrictive “license“ there is.

Automatically generated content and bad copies

The Dejavu study already mentioned something they called “autogenerated“ content. Not everything they call autogenerated content is a problem for us. Automatically included libraries will in fact improve our results. The problem is non-copyrighted content, for example the output of a framework. The Dejavu project would have declared such projects to be copies, while in terms of licensing, they must be seen as completely unrelated.

[13] “Brgerliches Gesetzbuch“, a part of German law

[14] https://github.com/Semantic-Org/Semantic-UI

2.8 Future work

2.8.1 Running the pipeline on the entire Dejavu data set

Why this wasn't done

Primarily due to hardware limitations we didn't acquire the full data set from which the Dejavu results were created which the people behind the project offer to provide. The main hardware requirement would be an absolute minimum 25 TB of storage for the source data, results and the permanent storage of intermediate data like generated hashes (without the JavaScript part).

Usage of Dejavu results in preselection

The results of the Dejavu project can be used to identify identical files, but they were compared without looking at the comments, which would include the license. In other words, a possible optimization could be to only analyze the same files once, or a limited number of times. The disadvantage of this approach is that one would miss potentially interesting changes in the licenses headers within the files themselves in favor of comparing what likely are included libraries and their licenses. It should be carefully considered if that is a trade off one is willing to make.

Prediction of performance

How fast such a full pipeline would run isn't trivial to predict. If we assume that we can hold a 97% overall hit rate, we would expect a run time of 100 days - based on the size of our set - which should scale down almost linearly with the amount of CPUs used to the point where the hard drive becomes the limiting factor. On a 8 core machine that has enough RAM to keep all continuously required files in memory, it should be possible to run the set withing two weeks. The minimum amount of RAM would be approximately 12 GB for the project clone information, all other stages can be chunked easily.

2.8.2 Reasons for conflicts

Our work showed that, as a general rule, the more restrictive a license is, the more likely it is to be violated. It would be interesting to explore if this is because people don't respect those licenses in the first place or if they are violated because length and complexity of licenses obviously often correlate with the restrictiveness of the license. It would be interesting to know if people are aware that the lack of license does by no means imply silent acceptance of the usage or distribution of the such work by other people.

2.8.3 Cross referencing other data

Since the Dejavu project found that many of the duplicate code segments are libraries, the Dejavu pipeline, mainly the hashing step could be applied to the data from a package manager like NPM. Doing this would allow to correctly identify projects of origin and their licenses. However we cannot think of an simple way to acquire this data, it would require somewhat dedicate scrapers for each language, that somehow create a package list and then fetch the packages from that list.

2.8.4 Transference of copyright

Our normal license recognition method ignores the copyright, as it may be different for the same license in different files or projects. While at first we thought this could be easily checked by simply comparing the, easily detectable, copyright lines separately from the license, we underestimated the possible complexity of accurately parsing such information. Even if a project fully contains another project, the top level license might not include the original copyright owner. This approach could be applicable when dealing with forked projects, but even then migration of such copyright information in the cases when there seemed to be discrepancies at first, is very common.

2.9 Conclusion

In order to get an estimation about the danger of running into license conflicts when dealing with open source software, we developed an analysis tool that is able to collect data and recognize licenses. We then used the existing results of the Dejavu paper and cross referenced them with the licenses we detected. For this we also collected information about the compatibility of licenses and combined them in a machine readable format which we use to determine if two licenses can be used within the same project. The analysis tool is able to process large amounts of data in relatively short time.

The amount of conflicts we detected is between 0.1-2.8%, depending on the language ecosystems system. This number seems low at first, but even medium sized software projects often include multiple other projects which in turn contain more. If any of the projects in that dependency tree contains a license conflict, the top level project does as well. Including such indirect conflicts in the results increases the total number of conflicts by up to 70%. Companies using open source should therefore be very careful to not only check the licenses of the projects they directly include, but also of those that are included indirectly.

Though projects like SPDX have made efforts in this area, like for example providing standardized versions of licenses and license headers we found that there is still a massive deficit in standardization of license inclusions which hinder automatic analysis to a high degree.

Appendix A List of Acronyms

A.1 Organizations

- CC - Creative Commons
- FSF - Free Software Foundation

A.2 Licenses

- AGPL - GNU AFFERO General Public License
- BSD - Berkeley Software Distribution
- CC0 - CC0 1.0 Universal Public Domain Dedication
- EFL - Eiffel Forum License
- EPL - Eclipse Public License
- GPL - GNU General Public License
- ISC - a license published by the Internet Systems Consortium
- LGPL - GNU Lesser General Public License
- MIT - a license published by the Massachusetts Institute of Technology
- MIT/X11 - X-Consortium variation of the MIT license
- MPL - Mozilla Public License
- WTFPL - do What The Fuck you want to Public License

A.3 Other

- CSV - comma separated values, a simple format for structuring data
- LSH - locality sensitive hashing
- NPM - node-js package manager
- SPDX - Software Package Data Exchange

References

Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories* (pp. 233–236). MSR '13. San Francisco, CA, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2487085.2487132

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig. 3*, 91–97. doi:10.1016/j.diin.2006.06.015

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., … Vitek, J. (2017). Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang. 1*(OOPSLA), 84:1–84:28. doi:10.1145/3133908

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K. & Lopes, C. V. (2015). Sourcerercc: Scaling code clone detection to big code. *CoRR*, *abs/1512.06448*. arXiv: 1512.06448. Retrieved from http://arxiv.org/abs/1512.06448