THE FRAMEWORK PROGRAMME FOR RESEARCH AND INNOVATION

HORIZON 2020

CROSSMINER

Developer-Centric Knowledge Mining from Large Open-Source Software Repositories

Project Number 732223

D3.5 Mining Documentation and Code Snippets

Version 1.0
30 June 2019
Final

Public Distribution

Edge Hill University

Project Partners: **Athens University of Economics & Business, Bitergia, Castalia Solutions, Centrum Wiskunde & Informatica, Eclipse Foundation Europe, Edge Hill University, FrontEndART, OW2, SOFTEAM, The Open Group, University of L'Aquila, University of York, Unparallel Innovation**

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L′Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	Document outline	3 June 2019
0.2	Added Content related to Software Documentation Mining and processing	14 June 2019
0.3	Added Content relating to Identification of API Changes	10 June 2019
0.3	Added Content relating to Identification of API Changes	13June 2019
0.4	Added Content relating to Recommendation of snippets and discussions	13 June2019
0.5	Added Content relating to new readers, metric providers, risks/limitations and conclusion	21 June 2019
0.7	Document Ready of Internal Review	24 June 2019
0.8	Addressed Partner Feedback	27 June 2019
1.0	Final version	30 June 2019

Acronyms

Presented in the table below are a list of acronyms and their associated explications commonly used throughout this deliverable.

Acronym	Explication
OSS	Open Source Software
NLP	Natural Language Processing

Table of Contents

Executive Summary

In this deliverable, we present the work conducted to complete Task 3.4. Specifically, we present two readers developed for retrieving documentation files and three different tools for processing documentation and determine aspects such as documentation types, readability and license. We describe the tool created along with our partners from *Centrum Wiskunde & Informatica (CWI)* for detecting discussions revolving around API migration issues. The description of a recommendation system for code snippets and discussions is introduced along with new readers that have been created for CROSSMINER regarding communication channel sources. We also explain, new metrics that have been created in the last 6 months to process the data retrieved by CROSSMINER. An analysis of risks and limitation is presented at the end of the deliverable, followed by a conclusion summarising our progress presented in this deliverable.

1 Introduction

In this deliverable, we address the work required to fulfill Task 3.4 which is the last of 4 tasks required to complete work package 3 (**WP3**) of the CROSSMINER project. **Task 3.4** is aimed at developing tools/components that would facilitate (1) the analyses of open source software (OSS) project documentation to determine *completeness* and *readability*; and (2) the analyses of online resources in order to recommend *code snippets* and *discussions* relevant to the code being developed in the IDE. Thus, in this deliverable we introduced new tools that we have developed for CROSSMINER for these purposes. More specifically, we present the documentation readers and the tools for assessing documentation, a documentation classifier, a documentation readability tool and a license analyser for documentation files. We also explain in this deliverable, the progress that has been made for searching and indicating which discussions, i.e. software bugs, emails, forums posts, revolve to API migration issues. It is important to note that all the tools/components developed to address Task 3.4 are new tools that do not exist in OSSMETER.

The work completed in this deliverable builds on, and in some cases, extends the work completed in previous deliverables in order to fulfill any outstanding work from **WP3**. In deliverable 3.1, we investigated state-of-the-art methods for text representation in order to address the requirements of Task 3.1. In deliverable 3.2, we presented research work conducted to develop clear understanding of Task 3.2 which focuses on developing methods for reading and searching text sources associated to OSS projects. Task 3.3 which focuses on the development and evaluation of natural language components was addressed in two deliverables i.e., D3.3 and D3.4. Both deliverables presented the tools and components we developed to provide a collection of text mining components that will allow developers to analyse a wide range of textual content related to OSS projects, in order to produce useful results that could improve their performance and quality of code produced. In particular, both deliverables provided components we developed for reading and searching text sources associated to OSS projects such as bug trackers, newsgroups, mailing lists, forums, stack overflow etc. We extended the existing sources in the current deliverable to include 3 additional sources discussed in Section 5. To analyse information retrieved from these sources, we also developed a number of processing components capable of computing meaningful evidence for developers such as topics being discussed, the sentiment associated with user comments etc. These processing components were extended in the current deliverable to include 14 additional components discussed in Section 6 which are specifically aimed at addressing Task 3.4.

1.1 Overview

The remaining of this deliverable is organised into six sections. Section 2 provides details about the tools and metrics developed to mine project documentation in order to determine completeness and readability. Section 3, presents the tool created for detecting discussions that revolve around API migration issues. Section 4 provides details about the tools and metrics developed to make relevant recommendations to the code being developed in the IDE. Section 5 and Section 6 present the upgrades and additional components that have been developed in order to complement the number of sources that CROSSMINER is capable to analyse, as well as, the new metrics that our platform can process. Section 7 identifies the risks and limitations of the work presented in this deliverable. Finally, Section 8 presents the conclusion of this deliverable as well as reports on the current progress of WP3; including future & outstanding work.

1.2 Intentions

The aim of **Task 3.4** is to develop and integrate into CROSSMINER, tools capable of analysing a wide range of sources and recommend code snippets/online discussions relevant to code being developed. Fulfilment of the aim requires addressing the use case requirements presented in Table 1. Each requirement has an ID, a description and priority[1].

1.3 Outcome

The outcomes of this deliverable are the components related to documentation analyses and code/discussion recommender that have been developed and integrated into CROSSMINER. The developed components are designed to be compatible with the CROSSMINER platform, the knowledge base and can also be used to run bespoke workflows for knowledge extraction.

[1]Shall - must be fulfilled, Should - should be fulfilled, May - may be fulfilled

Ref	Description	Priority
D38	Shall collaborate with the source code mining work package to analyse documents that contain both natural language and code, e.g. documentation and bug reports.	Shall
D39	The knowledge base (Mining Cross-Project Relationships work package) shall provide the infrastructure for hosting the indexes populated by natural language analysis.	Shall
U35	Able to search documentation	Shall
U42	Able to detect in the data sources text referring to one or several bugs	Shall
U43	Able to detect in the data sources text referring to one or several commits	Shall
U45	Able to extract sentiment analysis from wikis	Should
U49	Able to detect in the data sources one or several commits hashes	Shall
U50	Able to list commits with bugs	Shall
U51	Able to list bugs with commits	Shall
U59	Able to identify code snippets that use old and new third-party API in forum threads concerning migration of the usage of the given third-party API	Shall
U62	Able to extract text from HTML and markdown to feed natural language analysis and identify code snippets	Shall
U63	Able to extract text from PDF to feed natural language analysis and identify code snippets	May
U64	Provides recommendations to add documentation commonly found in successful projects	Should
U65	Provides recommendations to improve the structure of the documentation	Should
U66	Able to analyse Java code snippets	Shall
U67	Able to analyse JavaScript code snippets	Should
U68	Able to analyse C code snippets	Shall
U69	Able to analyse PHP code snippets	Shall
U110	Able to analyse PDF documents	May
U111	Able to identify the documentation contains a Getting Started	Should
U112	Able to identify the documentation contains a User Guide	Should
U113	Able to identify the documentation contains a Developer Guide	Should
U114	Able to identify the documentation contains a Code Snippets	Should
U115	Able to analyse the documentation has a License	Shall
U116	Able to analyse readability of documentation	Shall
U185	Communication channel parsers use MBoxes	Shall
U188	Documentation parsers use data dumps	Shall

Table 1: WP3 Use Case Requirements related to Task 3.4

2 Software Documentation Mining and Processing

Software documentation is a collection of documents that can have different formats, from static text to dynamic graphics, that describe in detail a software product in order to be used by developers or end-users [43].

The success and usefulness of an OSS will reside not only on the quality of the software product or aspects related to users happiness and reactivity of developers to fix issues, it also resides on whether the OSS provides documentation and its quality. For example, in a survey done in 2017 by GitHub, it was found that 93% of developers complained about incomplete or confusing documentation [17].

According to [37, 34], a software product that is expected to be shared with the community must be documented in detail and following a standard formatting. In other words, documentation should contain enough information for developers and users to understand which is the purpose of the tool, how to use it and how to maintain it. As well, it should be written in clear way without leaving any kind of ambiguity. Moreover, the readability and understandability of software documentation is essential not only to use, but to maintain software [1].

In the following sections, we introduce the series of tools that we have created for CROSSMINER in order to retrieve documentation from different kind of repositories. As well, we present tools that we have created for processing and analysing documentation files.

2.1 Documentation Readers

The development of documentation readers started by asking our user case partners to present us with some examples of documentation sources, especially those that they would be interested in analysing. We recurred to this approach, instead of defining ourselves the documentation sources to exploit, because we want to fulfil as much as possible the necessities that our partners have. Moreover, we, as researches could have had a bias to specific sources that might or not assist our user case partners.

The response obtained from the user case partners consisted of a list of documentation sources that greatly variate on their format and type of access. For example, we had websites completely dedicated to a project documentation, websites that contained the information regarding a project and its documentation, documentation in the shape of wikis accessible through REST APIs, wikis stored as git repositories, documentation stored in servers, among others. Due to the great variety of documentation sources, and after an extensive analysis, we decided to categorise the documentation sources into two different groups. The first one are those sources that are stored using a git repository; the second one includes the documentation sources that can be retrieved using a web crawler. The decision of splitting documentation sources into these groups is to encompass a large portion of the documentation sources without increasing to a great extent the number of readers to develop.

In the following sections, we explain in detail the git-based documentation reader and the web-crawler-based one, that in CROSSMINER scope, is called systematic documentation.

2.1.1 Git-Based Reader

This reader covers documentation that is stored as or in a git repository. Examples of this documentation are Github Wikis, Gitlab Wikis[2] and Bitbucket Wikis. This reader has been implemented by creating an extension

[2]It should be indicated that Gitlab offers as well the possibility of linking the wikis to external websites, these are not covered by this reader as they are not git projects.

of the current git reader found in CROSSMINER. It provides a certain number of unique functionalities, such as the verification of the repository[3] or the complete processing of the documentation repository on the first day of analysis, regardless if a commit has been done or not[4].

The main characteristic of this documentation reader is that it is capable of analysing the evolution of files as everything is recorded through commits. Furthermore, if CROSSMINER users desire it, they can utilise along this reader not only the metrics created for documentation but also for the analysis of commits. The only limitation of this reader is that it cannot retrieve files that have in their name characters considered as illegal by operating systems (`\ / : * ? " < > |`). Files with illegal characters can happen in certain wikis online interfaces, such as those provided by GitHub, were the titles of wikis entries, which become files names, are not restricted in the use of any character.

In Table 2, we present the parameters needed to create a Git-Based Documentation reader in CROSSMINER.

Table 2: Parameters for Git-Based Documentation Reader

Parameter	Requirements	Description
URL	Mandatory	The location where the git repository is stored

2.1.2 Systematic documentation

The systematic documentation reader is a tool created to retrieve the files associated to software documentation that are stored in repositories different than git. For example, this reader is capable of retrieving information stored as a website in a specific URL or accessing to a server that only shows PDF files.

The reader is based on a web crawler, a tool that for a given URL explores the different hyper-links to find and download the totality or a portion of the website. Specifically, we make use of Crawler4j, an open-source library that implements a web crawler based on Java and provides different functionalities, such as a high customisation or access to websites protected by password[5].

Although this reader is versatile, in the sense that it can explore a great variety of websites, it cannot retrieve elements that existed previous the date of analysis or that are not accessible through a crawler. In other words, these websites do not provide access to historic versions of the website analysed and in consequence, we can only retrieve the most recent version of the files. As well, Crawler4j has been configured to follow the rules defined by websites administrators regarding the use of robots.

The decision to call these readers systematic instead of web-crawling ones, is the fact that these readers should not be executed on daily basis, but on defined periods greater than one day, as we risk the readers being banned or blocked by the servers hosting the documentation. More specifically, web crawlers can put a lot of stress to hosting servers, therefore, it is custom to wait several days before retrieving a portion of the website.

This documentation reader has two different modes. The first one is for websites in which documentation can be accessed without restrictions; in Table 3, we present the parameters necessary to create this reader. The second mode is for websites where it is necessary to log-in before accessing the documentation files; for this mode, the reader has to be set with a greater number of parameters, which are described in Table 4.

[3]In multiple cases, these wikis can be created or deleted by developers without notice.

[4]Heuristics have been defined to determine the version of the repository that should be analysed.

[5]The user needs to define the URL of the login page, the username, password, but also the fields of the names where the username and password should be pasted.

Table 3: Parameters for Systematic Documentation Reader with free access to the files that must be crawled.

Parameter	**Requirements**	**Description**
URL	Mandatory	Location that has to be crawled
Execution Frequency	Optional	It indicates how frequent, in days, should be executed the reader. The default value is 0, which means that the reader is executed just once

Table 4: Parameters for Systematic Documentation Reader where the files to crawl are protected behind a password.

Parameter	**Requirements**	**Description**
URL	Mandatory	Location that has to be crawled
Login URL	Mandatory	Location of the website were the login access has to be done
Username	Mandatory	Value that has to be used as username to log-in
Username field name	Mandatory	It indicates to the crawler which is the name of the text field where it has to be used the username
Password	Mandatory	Value that has to be used as password to log-in
Password field name	Mandatory	It indicates to the crawler which is the name of the text field where it has to be used the password
Execution Frequency	Optional	It indicates how frequent, in days, should be executed the reader. The default value is 0, which means that the reader is executed just once

2.2 Classification of Documentation

Software documentation is a collection of documents that have for objective to present in detail the software product [43]. Moreover, it is composed of different sections such as *Installation guide* or *Getting started* [34], that allows users and developers understanding clearly the functioning of a tool, library or product in general.

Although software documentation is a key element for any product that is expected to be shared internally or externally, in occasions its quality does not fulfil the minimum standards or, in the worst cases, is non-existing [34, 37]. These elements certainly put on risk the success of any software project or can affect negatively the happiness and productivity of users and developers [39]. Therefore, in CROSSMINER we have focused on different tools for the analysis of documentation, one of them residing on the detection of sections present in documentation files.

In the following sections, we present the work done in order to solve the problem related to the detection of software documentation segments. More specifically, we explain the classifier based on heuristics that is used to detect different documentation types within one or multiple files.

In Section 2.2.1, we present the state-of-the-art. The methodology is detailed in Section 2.2.2, while the evaluation is shown in Section 2.2.3. The discussion regarding the classifier developed in this deliverable is presented in Section 2.2.4. We conclude regarding this tool in Section 2.2.5.

2.2.1 State-of-the-art

In the state-of-the-art it is difficult to find works related to the classification of software documentation, in fact most research works are related, for example, to the evaluation of software documentation quality, utility and costs [43, 14]; issues that are caused by bad practices during the creation and maintenance of software documentation [2] or the automatic improvement of documentation elements [38, 21]. In the following paragraphs, we present the works, that to the best of our knowledge, have discussed the classification of software documentation.

According to a technical report from the National Aeronautics and Space Administration (NASA) [37], software documentation can be classified according to their level of detail and their format quality. Specifically, there are 4 levels of detail:

- Level A: It represents the best documentation that can be produced. It allows understanding, maintaining and extending a software component by any person with the minimal required technical qualifications without needing any external assistance. Aspects such as descriptions, functionalities and operation are described carefully that they do not leave space for any kind of ambiguity. This level of documentation is useful for software that is highly used or that is expected to be shared with other people.
- Level B: The elements described in this level of documentation are the same as those found in documents of Level A. Nonetheless, details are less specific and clear, making it difficult to people without the average technical qualifications to understand the software. In some cases, descriptions can be vague, leaving the comprehension of algorithms or methods to the developer discretion but never putting on risk the performance or quality of the software. This level of documentation can easily be upgraded to Level A with minor reworking. This level of documentation should be used in software that has a medium level of use, but it is not expected to be shared with third parties.
- Level C: In this level, documentation can only be understood by highly skilled people and their understanding can be used uniquely to maintain or develop code at an acceptable degree. New developers might need to be in contact frequently with the original programmers. With every change in the code there is a minimum risk of affecting the software quality and performance; moreover, it increases the time of debugging and exploring the code. This kind of documentation should not be used in projects that will be used by external people, that is expected to have a long life cycle or extended use.
- Level D: This level corresponds to documentation with the minimum acceptable degree of detail. It is frequently only suitable to the original developer(s). A lot of rework has to be done, in order to be useful by new programmers.

Independently of the level of detail, documentation can belong to one of four different categories of format quality:

- Category 1: This documentation has the highest levels of format quality as it belongs to software that is expected to be of general interest or widely used, either by internal or externals users. It is frequently edited and proofread by professionals and contains high quality images. This kind of documentation is normally used for software that is considered as stable.
- Category 2: It represents documentation that has a high format quality, however, some elements, such as editing or typography, do not follow the best standards. This documentation is expected to be read mostly by users within the organisation that developed the software. Although it might not be edited by professionals, documentation within this category has passed proofreading and a standard level of formatting has been used.
- Category 3: Documentation with this format quality belongs to software that is expected to be used only in-house, that has a short life cycle or that it is still considered unstable. It is expected to fulfil a standard formatting.
- Category 4: In this category we can find the documentation that has the minimal acceptable level of formatting. It is frequently used by programs that might not be used again, but due to organisation policies, it is necessary to have a record of the software implementation.

In [34], the author presents a different documentation classification, which focuses on first place on the different cycles of software development and on second place on the target reader. In the following listing, we describe in detail the proposed documentation classification:

- Process documentation: It registers the procedure followed by developers to create and maintain the software. It describes elements such as schemes, quality process, verification methods, team organisation and also it can record memos, reports or minutes.
- Product documentation: As its name indicates, this documentation focuses on the software that is being developed. It can be system-oriented and/or user-oriented.
  ▲ System-oriented: This documentation explains in detail how the software was implemented and tested, as well as its architecture. The descriptions of the software must be detailed enough to understand the software and being able to give it maintenance.
  ▲ User-oriented: It contains a description of how users can utilise the developed software. A user-oriented documentation should contain the following types of documents:
    ▼ Functional description: It is an overview of the software which allow users deciding whether the product is adequate to their necessities. It can describe as well the system requirements.
    ▼ System installation: It describes in detail, how the software has to be installed and configured.
    Introductory manual: This type of documentation presents how to get started with the product in a general aspect. It contains in most cases with examples or illustrations that can make any user to understand fast and clear how to use the software.
    ▼ System reference manual: It is a documentation type that describes formally and in detail the software.
    ▼ System administrator's guide: This type of documentation is optional and depends on the type of software. In summary, it provides information about how the software product can or should interact with other elements, software or hardware, which are possible error or status messages from this interaction, and how to maintain this interaction.

2.2.2 Methodology

Despite the existence of different software documentation classification described in Section 2.2.1, to the best of our knowledge, the automatic classification of software documentation has not been explored. Moreover, we did not find any corpus related to software documentation that could be used as base.

Thus, in order to solve this classification task, we decided to manually collect documentation of different OSS. Specifically, we selected 40 projects supported by Eclipse Foundation and analysed their documentation. This analysis consisted on determining which are the most frequent file formats used for documentation and which are the elements described in the documentation.

From the manual inspection, we determined that most projects use the following file formats: Microsoft Word, Portable Document Format (PDF) and HTML. From our knowledge and experience, we extended the possible file formats as well to: OpenDocument Formats, Manifest, WordPerfect, AbiWord, DJVU, Rich Format Text and Microsoft Powerpoint.

In addition, based on the manual analysis of the documentation files retrieved by us, our Work Package requirements (U111, U112 and U113) and the classification of product documentation provided by [34], we have defined the following types of documentation that are detected in CROSSMINER:

- **Getting started guide**: This type of documentation presents a global introduction to the OSS as it summarises aspects such as installation, utilisation and development.
- **Installation guide**: As the name indicates, it is a documentation type in which it is explained which are the requirements of the software, its installation and configuration.
- **User guide**: In this documentation type, the software is presented in detail to the user. In occasions, the user guide contains as well examples or tutorials that make easier the understanding and use of the software.
- **API guide**: This type of documentation is frequently found in software that can be accessed through an API. It explains all the elements necessary to use the software API.
- **Development guide**: In this kind of documentation, it is explained either how to use the software to create tools, or to expand its capacities. In some cases, this documentation explains how to contribute directly to the project.

We observed as well, that documentation can be split in multiple files, and those might contain one or more types. Thus, this classification task is multi-label. For example, a file containing a Getting started guide might as well contain an installation guide. Or a User guide might incorporate a Getting started and an Installation guide.

Due to the lack of a corpus annotated regarding the types of documentation, and the expensive aspects of creating one, we decided to deal with the documentation classification using an approach based on heuristics.

This first step followed to achieve our goal consisted in finding tools for extracting text from the files. Although this task seems easy and straightforward, it is not the case, as not all the files format store in the same way the information. We decided to use Apache Tika[6] an open-source library that collects multiple tools that can be used to extract metadata and text from different types of file. It contains methods for the automatic detection of file formats and use automatically the most adequate text extractor. Apache Tika is capable to present the extracted text in two formats, plain text and HTML. The latter format, depending on the processed file, can be used to get information regarding the file formatting, such as portions in bold text, headings, listings, among others. Moreover, its licence, Apache License 2.0, is compatible with the licence of CROSSMINER.

Secondly, we explored the two possible Apache Tika output format, HTML or plain text, to determine which was the most beneficial for the detection of documentation types using heuristics. After experimenting with the corpus described previously, we considered that the HTML format presented an advantage over the use of plain text as format. This benefit resides in the fact that files headings, such as titles and subtitles, can be easily identified and extracted. Moreover, we observed that headings follow similar patterns and could be a good indicator of which types of documentation was present.

For the detection of patterns in headings, we considered two possible options. The first one based on edit distance and the second one based on regular expressions. In the first option, we considered specifically:

- *Levenshtein Distance*: It is defined as the number of insertions, deletions or substitutions that have to be done to match two strings. If no insertions, substitutions or deletions have to be done, it means that two strings are exactly the same.
- *FuzzyWuzzy algorithm*[7]: This algorithm is based on Levenshtein Distance, however it implements a series of heuristics to improve the matching of two strings. For example, FuzzyWuzzy is able to calculate how well two strings match even if their tokens are in different order.

With respect to the second option, a regular expression is a string, written following the rules of a *regular language*, that allows creating, searching or matching patterns [13]. They are founded on *formal language*

[6]https://tika.apache.org/
[7]chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python

theory and are used extensively in searching tools, text validators and programming languages parsers. An example of a regular expression is *[0-9]?[0-9]:[0-9][0-9] ?(am|pm)*, which can match strings such as *9:30am* or *01:40 pm*.[8]

After the analysis of our possible methods for detecting patterns in headings, we decided to choose regular expressions. The main reason is that with regular expressions we can have a better control of the patterns to match and in a structured and understandable format. We considered as well that with edit distance methods, we need to define not only patterns, as pure text strings, but also thresholds that indicate whether the match should be considered as correct.

The regular expressions were conceived by manually analysing the headings used by each documentation type, but also by extending patterns found in other types of headings. For example, we found in one documentation type the use of the pattern *how-to*, while in another documentation type we observed the pattern *how to*; at the end, and for both documentation types, we created a pattern that could support both: *how(|-)to*. In total, we have created 21 regular expressions; specifically, we have generated 4 for getting started, 6 for installation guide, 2 for user guide, 3 for API guide and 6 for development guide. Although the number of regular expressions might seem low, it should be understood that one regular expression can encompass more than one pattern.

2.2.3 Evaluation

Despite the lack of any annotated corpus, we decided to generate a small evaluation set that we annotated in house. Being more specific, we downloaded the documentation of 64 different OSS projects[9] from sources such as GitHub[10], GitLab[11], CTAN[12], GNU[13] and Apache Software Foundation[14]. From the documentation from these 64 OSS projects, and as the analysis of documentation is done by files, we selected randomly a total of 90 files[15]. These files were manually annotated by 4 researchers using the following criteria:

> *In the present directory, you will find files randomly selected, and in their original format, from documentation of different OSS projects. Each file can contain the totality of the documentation or just a portion of it. Your task is to determine whether each file contain one or more of the following sections: . . . During the file annotation, it is necessary to indicate whether the file contains clearly the section (set a value of 1), i.e. marked by a heading or title, or the text in the file belongs to a larger section (set a value of 2). Listings, bullet points, menus or headings making reference to an empty section should not be considered. If a section is not present, set a value of zero.*

Each of the researchers annotated 45 files, 30 of these files were common for every researcher. The common annotation was done to calculate the inter-annotator agreement rate using *Fleiss' Kappa* [10].

[8]A more complex regular expression can be generated to prevent the matching of strings such as *99:99am*.
[9]These projects are completely different from the 40 projects described previously in Section 2.2.2.
[10]github.com
[11]gitlab.com
[12]ctan.org
[13]gnu.org
[14]apache.org
[15]Software documentation can be presented in different ways, it can be contained in a unique file while sometimes it can be spread in multiple files.

Table 5: Results in terms of Fleiss' Kappa of the inter-annotator agreement for the 30 common documentation files.

Data	Fleiss' Kappa
Original	0.174
Strict	0.252
Relaxed	0.246

Table 6: Thumbs' rule for Fleiss' Kappa according to [25]

κ	Interpretation
< 0	poor agreement
$0.01 - 0.20$	slight agreement
$0.21 - 0.40$	fair agreement
$0.41 - 0.60$	moderate agreement
$0.61 - 0.80$	substantial agreement
$0.81 - 1.00$	almost perfect agreement

As the problem is multi-label, i.e. a documentation file might contain more than one section, to calculate Fleiss Kappa, we created vectors using a hierarchy of decimal numbers. In other words, we set to *Getting started* the value of $10,000$, *Installation guide* the value of $1,000$, *User guide* a value of 100 and so on. For example, if an annotator marked a file to contain *API* and *User Guide* the vector would be 110, while a file containing a portion of the installation guide, it would have a vector of 2000. As there were two possible values for each section, i.e. 1 if the section was clearly delimited or 2 if text belonged to a larger section, we calculated in three different ways Fleiss' Kappa. One in which we kept the original values, one where we changed the values of 2 to 1 (*strict*) and one where we changed the values from 2 to 0 (*relaxed*). In Table 5, we present the results regarding the inter-annotator agreement for the 30 commons files; in Table 6, we present the thumbs' rule for Fleiss' Kappa [25].

We can observe in Table 5, that the agreement between annotators is between slight and fair. Due to the low values of agreement between annotators, we have decided to determine the agreement rate of the annotators by types of documentation. The results are presented in Table 7.

From Table 7, we can determine that the most difficult sections to annotate corresponded to *Getting started* and specially to *Development guide*, while the easiest is *Installation guide*. In Section 2.2.4, we will discuss in depth these results.

Because the agreement between annotators was low, we decided to select the final labels for the 30 commonly annotated files using two approaches: one where at least 3 annotators have chosen the label (*majority*) and

Table 7: Values of Fleiss' Kappa for each section type found in the 30 common files of documentation manually annotated.

Data	Fleiss' Kappa				
	Getting started	Installation guide	User guide	API guide	Development guide
Original	0.124	0.684	0.594	0.459	0.0993
Strict	0.101	0.736	0.594	0.670	0.144
Relaxed	0.216	0.745	0.367	0.369	0.0872

Table 8: Results in terms of Precision (P), Recall (R) and F-score (F1) for each of the types of documentation.

Data	Getting started			Installation guide			User guide			API guide			Development guide		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Strict-Majority	0.18	0.38	0.25	0.29	0.17	0.22	0.50	0.06	0.11	0.10	0.18	0.12	0.29	0.42	0.18
Strict-Half	0.33	0.36	0.34	0.23	0.13	0.17	0.50	0.05	0.09	0.15	0.25	0.18	0.12	0.30	0.17
Relaxed-Majority	0.14	0.36	0.21	0.29	0.21	0.25	0.50	0.10	0.17	0.10	0.25	0.14	0.12	0.50	0.19
Relaxed-Half	0.25	0.35	0.29	0.29	0.20	0.23	0.50	0.08	0.14	0.10	0.22	0.13	0.12	0.33	0.17

one where at least two annotators have selected the label (*half*). In Table 8, we present the results in terms of precision and recall for each type of documentation.

In Table 8, we can observe that, in terms of F-score, the best predicted type of documentation is *Getting started* while the worst predicted one is *User guide*. However, all the results are far from the desired performances. In the next section, we will discuss these results.

2.2.4 Discussion

As we observed in Table 5, the low values of inter-annotator agreement rates indicates that the manual annotation of documentation files is not an easy task. After a manual analysis of some annotated files where there was a disagreement, and a discussion with the annotators to know their point of view. In the following listing, we present the most representative:

- Some annotators considered that the description of software features is a or part of *Getting Started* guide.
- Difficulty to determine if the text in a file described a full section or not. For instance, a file describing a portion of a user guide, was considered by two annotators as the full user guide, by one annotator as getting started and another did not annotate it as a user guide at all.
- The visual aspect of a file affected the annotation. For example, a file describing one API method was considered by 3 annotators as a full API section, the other as a portion of an API documentation.

Taking into account the previous discussion, we can observe, that the lack of context, i.e. full access to all the documentation files, has affected the perception of the annotators. However, it is not clear for us how to solve this issue: should we make the annotators to label all the files of a software documentation or just give them access to them in order to understand the context. This is specially in large projects, were the number of files composing the documentation ca be very large; thus, annotating all the files can be very tedious, while finding the correct context of a file can be difficult.

Another point that affected the annotation was the visual aspect of the documentation. We provided to the annotators files that in occasions have lost their format during the retrieval, for example, HTML files were not accompanied by a CSS file. This has made that aspects such as coloured boxes, menus or footnotes, were completely lost, making the annotators unable to determine, for example, borders between sections or whether the file contained a heading or not. To prevent this issue, we might need to give annotators the links to the documentation file instead a downloaded version of it.

The results obtained by the classifier are far from the expected performance, however, taking into account the difficulty of manual annotators to agree, we consider that the results are adequate. In occasions, the lack of matching between the prediction and the annotations is because a case not covered by the regular expressions. For example, in a file the section that describes how to install and run the software is called simply *Running*.

In some other cases, it is because the words can have multiple interpretation. For instance, in a file, there is a heading indicating *how to contribute?*, nonetheless, the text in this section expresses how to contribute monetarily to the support the developers, instead of how to participate in the development of the software.

There is a lot of work to do in order to improve the output of the classifier. More rules could be created and possibly word sense disambiguation tools could be used. However, we consider that in order to truly improve the outcome of the classifier, it is mandatory to create an annotated corpus. From this collection of files, we could extract patterns or, if it is large enough, we could use machine learning approaches. In all the cases, the creation of this corpus should consider documentation context and visual aspect, in order to increase, in theory, the agreement between annotators.

Although it can be discussed that the approach used for documentation classification is not useful for documentation without headings, based on the literature review done in Section 2.2.1, we can conclude that for any project that is expected to be shared, the software documentation should have a minimal formatting. In other words, according to works, such as [34, 37], good documentation must use elements such as headings, in order to make visible and clear the elements that users or developers need to understand and utilise the software product.

During an informal testing, some documents described at the beginning of this section, despite being correctly opened by tools, such as Adobe Reader, they used (internally) non-standard ways to store the text, making impossible to extract the data using Apache Tika[16]. This means that it is impossible to extract the text from a file and in consequence determine the types of documentation present. This is a limitation of our approach, in the sense that we rely on the correct extraction of text using a third-party tool. Although we could rely as well on the extraction of text using Optical Character Recognition (OCR) tools in case Apache Tika fails, the problem of the classification is not solved. In first place OCR tools are not 100% accurate but most importantly, they are incapable of indicating us whether a portion of text is a heading or not.

Another limitation of our approach is the fact that, it cannot be applied to files that after extracting their text do not generate headings, such as plain text files. We could still apply the regular expressions to the text without headings from these files. However, we increase the risk of matching portions of text incorrectly , since headings are useful for providing context in certain documentation types.

2.2.5 Conclusion

In order to increase the success rate of any software product that is expected to be shared with other potential users, it is necessary to provide a collection of documents that describe in detail the functioning and structure of the software product. This collection of documents are called software documentation and are composed of a variety of sections such as *Installation guide*, *Getting started* or *API guide*.

However, despite the creation of software documentation is considered to be part of the best practices that should be followed by developers, in occasions software documentation do not have the minimal standards of structure or formatting. The identification of the sections described in software documentation can provide information to developers indications of possible missing elements, while to the users indication of how easy could be the use and understanding of the software.

Therefore, in CROSSMINER, we have decided to incorporate a completely new tool which consist on the determining whether software documentation contains elements such as *User guide*, *Getting started* and *Development guide*. This documentation classifier has been created using heuristics. More specifically on the

[16]In issues.apache.org/jira/browse/PDFBOX-3742 we find an example of this kind of errors. The developers of PDFBox, the tool used by Apache Tika for extracting text from PDF, explain that these problems can only be solved using heuristics.

matching of regular expression in headings found in documentation files. To achieve this, we analysed a collection of software documentation files created by us in order to create a series of regular expression that can detect 5 different types of documentation: *Getting started guide*, *Installation guide*, *User guide*, *API guide* and *Development guide*.

The developed tool was evaluated using a small corpus manually annotated. From the outcomes obtained, we can tell that the task of classifying automatically documentation types is not easy to do, either for humans or machines. For humans, aspects such as visual formatting or lack of context, affected severely the annotation process. While for machines, the use of regular expressions to match patterns, were not enough to deal with the great variety of words used to present the sections, even less, to cope with words with multiple meanings. Despite the outcomes obtained, we have presented in the discussions the elements necessary to improve in the future the automatic classification of documentation types.

2.3 Document Readability

Software documentation not only has to be composed of different sections [34], it has to fulfil certain levels of details and visual formatting [37], but most importantly it must readable and understandable [1]. For this last reason, we have included in CROSSMINER a tool for calculating the readability of the software documentation.

In CROSSMINER, readability has been defined as which should be the educational level of a reader in order to understand a particular text. This definition is based on a series of works, such as [5], [20], [6], that have explored how to determine the difficulty of a text [41]. It should be noted that the selected definition of readability, as stated by [40], do not consider aspects such as grammaticality, coherence or structure. However, to assess these elements automatically is still challenging, despite multiple approaches have been proposed as indicated by [9].

In the following section, we summarise the most relevant state-of-the-art metrics regarding text readability. Then we present the metric that we have chosen for CROSSMINER and its justification and some discussion revolving this subject.

2.3.1 Related Work

In the next paragraphs, we explain the most representative metrics that have been defined for determining the readability of a text based on the level of education that a reader must have in order to comprehend the text.

- **Automated Readability Index (ARI):** The ARI [33] is designed to gauge the understandability of a text. Its output is an approximate representation of the U.S. grade level needed to comprehend the text as shown in Table 9. The ARI relies on a factor based on characters per word, number of words per sentence and number of sentence per document. The formula for calculating ARI is given by Equation 1:

$$\text{ARI} = 4.71\left(\frac{\text{characters}}{\text{words}}\right) + 0.5\left(\frac{\text{words}}{\text{sentences}}\right) - 21.43 \tag{1}$$

- **Coleman-Liau Index:** The Coleman-Liau Index [5] is another metric designed to gauge the understandability of a text. Like the ARI, it relies on characters and its output approximates the U.S. grade level thought necessary to comprehend the text. However, unlike ARI which takes the totality of words

Score	**Age**	**Grade Level**
1	5 - 6	Kindergarten
2	6 - 7	First/Second grade
3	7 - 9	Third grade
4	9 - 10	Fourth grade
5	10 - 11	Fifth grade
6	11 - 12	Sixth grade
7	12 - 13	Seventh grade
8	13 - 14	Eight grade
9	14 - 15	Ninth grade
10	15 - 16	Tenth grade
11	16 - 17	Eleventh grade
12	17 - 18	Twelfth grade
13	18 - 24	College student
14	24+	Professor

Table 9: U.S. Grade Levels with Age

and sentences, Coleman-Liau takes the average number of words and sentence in a scope of 100 words. Its formula is given by Equation 2:

$$\text{Coleman-Liau} = 0.0588L - 0.296S - 15.8 \tag{2}$$

where L is the average number of letters per 100 words and S is the average number of sentences per 100 words.

- **Flesch-Kincaid Index:** The Flesch-Kincaid Index [24] was designed to indicate how difficult it is to understand a document. There are two types, the Flesch Reading Ease Index and the Flesch-Kincaid Grade Level Index. Both of them use the same core metric i.e., a hybrid of character and syllables, although they have different weighting scale. Readability with Flesch Reading Ease index is done by scoring a document between 1 and 100. However, the result is not immediately obvious to the user because its interpretation requires a conversion table to make sense of the score. For example, scoring between 70 to 80 is equivalent to school grade level 8, which means that the document should be easy to read, but this is not immediately obvious as people would normally associate higher scores to 'more difficult'. This was resolved in Flesch-Kincaid Grade formula which uses the U.S. grade table directly to represent scores. Thus, a document scored 8, means that the average reader has to have a grade 8 level of reading, or above, to understand it. Basically, the results of the two methods correlate approximately inversely such that a text with a comparatively high score on the 'Reading Ease' have a lower score on the 'Grade-Level'. The 'Reading Ease' is calculated with Equation 3 while the 'Grade-Level' is calculated with Equation 4.

$$\text{Reading Ease} = 206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right) \tag{3}$$

$$\text{Grade Level} = 0.39\left(\frac{\text{total words}}{\text{total sentences}}\right) + 11.8\left(\frac{\text{total syllables}}{\text{total words}}\right) - 15.59 \tag{4}$$

- **Gunning-Fog Index:** The Gunning-Fog Index [20] is a test indicating the number of years of formal education required by an individual in order to easily understand a text on the first reading. For example, a document with fog index of 13, has the reading level of a U.S. college student (see Table 9). Documents produced for a wide audience generally require a fog index of less than 12. Fog index uses a hybrid of character and syllables and its formula is given by Equation 5. A word is 'complex' if it consists of three or more syllables.

$$\text{Gunning-Fog} = 0.4\left[\left(\frac{\text{total words}}{\text{total sentences}}\right) + 100\left(\frac{\text{complex words}}{\text{total words}}\right)\right] \tag{5}$$

- **Dale-Chall Index:** The Dale-Chall Index [6, 4] provides a numeric score that represents comprehension difficulty when reading a text. Unlike Gunning-Fog index that computes complex words with syllables, Dale-Chall considers a word to be difficult, if it does not appear on a list of $3,000$ words that groups of fourth-grade American students could reliably understand. Its readability scale is slightly different from the other indices as shown in Table 10. The formula to calculate the scores is given by Eq 6. If the percentage of difficult words is above 5%, then 3.6365 is added to the raw score to get the adjusted score; otherwise the adjusted score is equal to the raw score.

$$\text{Dale-Chall} = 0.1579\left(\frac{\text{difficult words}}{\text{total words}} \times 100\right) + 0.0496\left(\frac{\text{total words}}{\text{total sentences}}\right) \tag{6}$$

Score	**Grade Level**
4.9 or lower	easily understood by an average 4th-grade student or lower
5.0 - 5.9	easily understood by an average 5th or 6th-grade student
6.0 - 6.9	easily understood by an average 7th or 8th-grade student
7.0 - 7.9	easily understood by an average 9th or 10th-grade student
8.0 - 8.9	easily understood by an average 11th or 12th-grade student
9.0 - 9.9	easily understood by an average 13th to 15th-grade (college) student

Table 10: Dale-Chall Readability Scale

2.3.2 Readability Tool for CROSSMINER

The document readability tool implemented in CROSSMINER is based on the Dale-Chall index. We have selected this metric due to multiple aspects. In first place, splitting words into syllables is not a straightforward task, as it relies mostly on phonetics [26][17] In addition, using vowel count i.e., syllables to determine word complexity, has some limitations. For example, the word 'interesting' has four syllables but is not generally thought to be hard-to-read. A short word can be difficult if it is not used very often by most people. This makes Dale-Chall index more suitable because it considers a word to be 'difficult' if it is not on a pre-defined word list[6]. Finally, to implement this tool, we can rely on tools that we have already introduced in CROSSMINER and that were presented in Deliverable 3.4.

As shown in Equation 6, Dale-Chall index requires the total number of 'words', 'sentences' and the 'difficult words' found in the document to be analysed. To determine these counts, we have used in CROSSMINER the tools NLP4J, a library that allow us processing text and get information regarding tokens, lemmas, sentences and Part-of-Speech (POS); NLP4J has been previously described in Deliverable 3.3 and Deliverable 3.4. Specifically, in order to calculate Dale-Chall, we first extract the lemmas from the text to analyse and filter them according to their POS tag; the filtration consist in getting only lemmas that correspond to words,

[17]In the state-of-the-art there have been described some tools, such as the one from [41], to determine the syllables in a text, however we did not arrive to have access to these tools.

such as nouns, verbs, adjectives. This has to be done because the output given by the lemmatiser includes normalised punctuation marks, numbers, among other information that should not be considered as a word. With the filtered lemmas, we can determine the number of words present but also the number of difficult words; this last task is done by determined how many lemmas from the text match the words that have been defined as familiar words. As NLP4J calculates automatically the number of sentences, then calculation of Dale-Chall is straightforward once the number of words and difficult words has been set.

2.3.3 Discussion

A document may yield low readability score because they used difficult words less frequently. However, a document may still be difficult to read if the grammar is bad. Likewise, badly executed ideas written with mostly easy-to-read words may yield good but misleading readability score.

Furthermore, readability varies massively between readers, particularly in analysing software engineering documents in which the presence of specialised terms or phrases is inevitable. For example, the phrase ‘on the fly’ is generally used in software engineering to indicate that a computer program runs without interruption. All the words in the phrase are easy to read so its readability would most likely be good. However, less technical-minded readers would find it difficult while experienced developers would find it easy.

2.4 License Analyser

Open source software projects (OSS) plays a crucial part in modern software engineering whether it be for personal, academic, scientific or commercial projects. Software developers can utilise OSS projects to create new innovations by alleviating their time and financial resources, but also to avoid “reinventing the wheel” [15]. OSS projects are more than just source code and may contain other assets which accompany the source code such as; models, configuration files and supporting documentation.

Open source software projects and their assets (intellectual property) are often accompanied by an open source license which governs their use. A license is a legal instrument which specifies the capacity in which the asset it accompanies may be used, modified and or shared under [23]. As they are a legal instrument, the language used is often formal. So why is it important for software developers, and product managers to be aware of open source licenses? Understanding the license (and its contents) is vital to avoid implications caused by derivative works[18]. Implications caused by derivative works could arise due to incompatibility with a business model/objective or by violating clauses within the license agreement [8]. Derivative work(s) with license implications or compatibility may lead to legal consequences, impact the ability to make profit (both financially and or from contributions made by the open source community) or affect commercial objectives [42, 29].

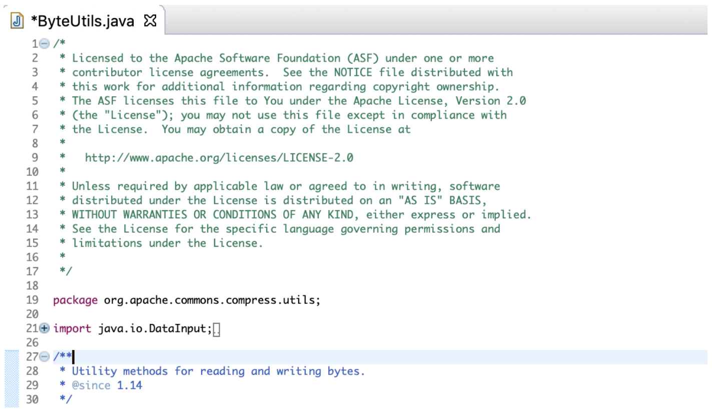
To further complicate things, projects often combine multiple OSS projects to achieve a common goal. However, not all licenses are legally compatible with one another. For example if you were to develop software which uses a library with an Apache Licence version 2.0 and another with a GPL version 2.0, you would legally not be able to since Apache Licence version 2.0 is not compatible with General Public License (GPL) version 2.0 as it contains certain patent termination and indemnification provisions. However, if an alternative library was found with a GPL version 3.0 these two libraries from a legal perspective would be compatible [12].

The licenses associated with intellectual property can be found in several forms and locations depending on the chosen license. For example, works licensed with Apache 2.0 are required to include a boiler plate (header) in every file related to the project, Figure 1 is an example taken from source code. Licenses such as EPL require the body of the license to be included alongside your work, the example shown in Figure 2 has been taken from the CROSSMINER project. Other such as BSD requires fields to be populated within a template and include either alongside or within your work as presented in Figure 3.

Knowing what licenses are linked to specific assets within an OSS project is valuable for any software engineer and product manager, in the sense that they can evaluate if this OSS is suitable and compatible with their software needs. The ability of doing this kind of analysis was not present in OSSMETER. Thus, in CROSSMINER, we have developed a classifier based on language models that detects open source licenses.

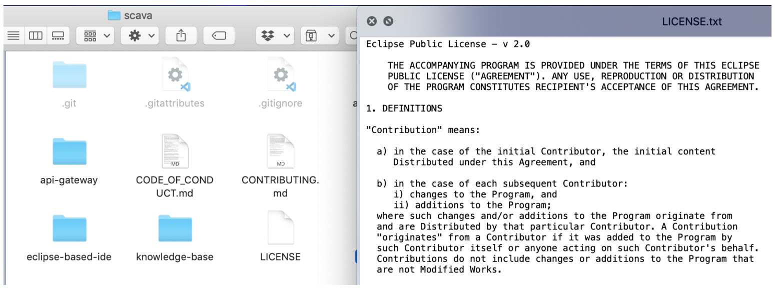
In the remaining sections, we present background information and related works specifically the detection of licenses. In succession, we proceed to present details of the methodology used for experimentation, followed by a discussion about the data set utilised for the experiments. In the latter part of this section we present details of the experimental and evaluative settings, results obtained from experimentation, a discussion of the results and a conclusion.

[18]Derivative works— Are works that either use, extend or make changes to the intellectual property



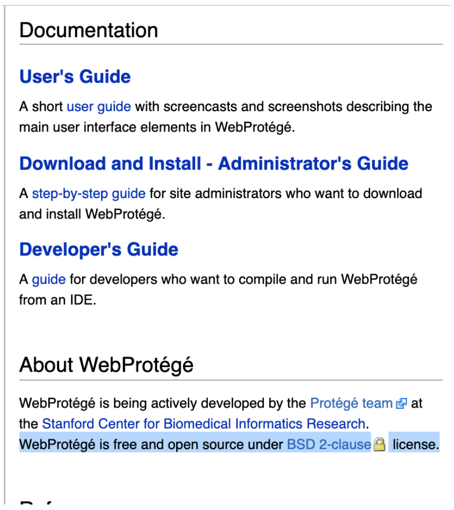
```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */

package org.apache.commons.compress.utils;

import java.io.DataInput;

/**
 * Utility methods for reading and writing bytes.
 * @since 1.14
 */
```

Figure 1: Apache License 2.0 Boiler plate taken from *org.apache.commons.compress.utils*



```
Eclipse Public License - v 2.0

    THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE
    PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION
    OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

  a) in the case of the initial Contributor, the initial content
     Distributed under this Agreement, and

  b) in the case of each subsequent Contributor:
     i) changes to the Program, and
     ii) additions to the Program;
  where such changes and/or additions to the Program originate from
  and are Distributed by that particular Contributor. A Contribution
  "originates" from a Contributor if it was added to the Program by
  such Contributor itself or anyone acting on such Contributor's behalf.
  Contributions do not include changes or additions to the Program that
  are not Modified Works.
```

Figure 2: EPL License Screenshot from CROSSMINER



Documentation

User's Guide

A short user guide with screencasts and screenshots describing the main user interface elements in WebProtégé.

Download and Install - Administrator's Guide

A step-by-step guide for site administrators who want to download and install WebProtégé.

Developer's Guide

A guide for developers who want to compile and run WebProtégé from an IDE.

About WebProtégé

WebProtégé is being actively developed by the Protégé team at the Stanford Center for Biomedical Informatics Research. WebProtégé is free and open source under BSD 2-clause license.

Figure 3: WebProtégé BSD licensed documentation

2.4.1 State-of-the-art

As already highlighted, the purpose of this work is to develop a License Analyser which is capable of data-mining textual sources for existence of a software license and if present, identify which licence it is. Throughout this section we present the state-of-the art and related works surrounding license detection and analysis affiliated to software engineering followed by a discussion and summary of the works presented and their compatibility with the CROSSMINER platform. Please note that works presented in this section are a selection from scientific, commercial and open source communities.

Developed by GNU, *LibreJS* is a web browser add-on written in JavaScript (JS) for GNU IceCat and Mozilla Firefox which detects *free* OSS licenses in order to block the execution of *non-free* / *non-trivial* JS on web-pages. It achieves this using a 2 layered approach. The first layer intercepts HTTP responses and rewrites their contents after analysing the JS code for license headers. The second layer analyses the web-page, that is scheduled to be executed, by observing the outcome of the web-browser parser; this acts as redundancy method in case the first layer is unable to detect JS. Currently, the tool is capable of identifying a total of 22 free OSS licenses[19][7].

Osler Code Detect is a free web-based application, provided by Osler[20]that locally scans software project directories to detect OSS licenses found in source code[29]. Based on the outcome of the analysis, this tool provides information relating to the licence distribution, license considerations and which were the files containing the license(s). It is worth noting that currently this tool has no publicly available API, does not disclose which licenses it is able to detect and it is required to manually select the project to be analysed.

FOSSID is a commercial software composition analysis tool kit which scans code for open source licenses and vulnerabilities[11]. It is available as both a web-based application and CLI (command line interface) . The web-based application has been designed for software developers to conduct open source compliance and audits. The CLI tool can be integrated into a DevOps life cycle to perform automatic compliance checks on software products. It boasts having the fastest open source software scanning engine which includes artificial intelligence to eliminate false positive results.

Developed by German *et al.* [15], Ninka is a license detection algorithm, written in Perl, specifically designed for identifying 112 types licenses embedded in source code. The algorithm consists of 6 stages, each is summarised below:

1. **License statement extraction** - *Extracts comments at the head of the source code*
2. **Text segmentation** - *Performs text pre-processing and sentence segmentation*
3. **Equivalent phrase substitution** - *Replaces known phrases with equivalent normalised variations from a dictionary consisting of 12 phrases*
4. **Sentence filtering** - *Splits a sentence into legal and non-legal parts*[21]
5. **Sentence-token matching** - *Using regular expression extract known sentence tokens*[22]
6. **License rule matching** - *Using a series of 126 license rules to match the results from the previous step to determine licenses*

[19]LibreJS - A full list of the supported *free* OSS licenses can be found https://www.gnu.org/software/librejs/manual/librejs.pdf

[20]Osler is a leading Canadian Law firm with expertise in areas of technology such as AI, Intellectual Property and Block Chain.

[21]Achieved using a list of 82 known (keywords)legal terms

[22]There are a total of 427 known sentence-token expressions

The authors of Ninka, evaluated their tools against other existing tools including FOSSology [18], ohcount[23] and OSLC[24] and concluded that it was both faster and more accurate than each of it competitors.

The *go-License Detector* [36], developed by the *source{d}* team, is an open source license detector written in GO . The tool utilises the SPDX license list and is capable of identifying ≈380 OSS licenses. It favours annotating projects with false positives over false negatives (fuzzy matches) for data mining purposes. The algorithm consists of 8 stages:

1. Find files in the root directory which may represent license files (E.g. LICENSE, license.md).
2. Convert structured text into plain text.
3. Normalise text according to SPDX guidelines.
4. Split the text into unigrams and build the weighted bag of words.
5. Calculate Weighted MinHash.
6. Apply Locality Sensitive Hashing and pick the reference licenses which are close.
7. For each of the candidate, calculate the Levenshtein distance.
8. Identify license based on a computed similarity score.

The tool was evaluated using a reference data set consisting of 1000 most starred repositories on GitHub[25] against other projects including GitHub's built-in license detector [16], Google's licenseclassifier[19] and Amazon's askalono[3]. The evaluation concluded that the go-license-detector has the highest detection rate (99%) and the lowest time to scan (13.5 seconds)[35]. Despite the high level of accuracy, it should be noted that the testing data set is not manually annotated. In fact, the authors consider that as all the downloaded repositories have licence, their detector should always find one; but there is no evaluation regarding if the detected license detected the correct license.

ScanCode [28], developed by nexB, is a comprehensive tool kit, implemented in Python, capable of detecting licenses, copyright notices, package manifests and dependencies from a given code base. It was originally developed to support their own software auditing services and is used by many industry leading open source organisation including Eclipse Foundation, Red Hat and OpenEmbedded.org. In contrast to the other related works, ScanCode is capable of extracting text from various sources including; source code, binaries and compressed archives. The text extracted is then passed through an extensible rules engine to detect ≈1000 open source licenses.

Unlike other works, the work of Vendome *et al.* [42] focuses on classifying exceptions which are appended to OSS licenses, rather than detecting license types. Vendome *et al.* specifies that under certain scenarios, licenses are modified by developers to include additional restrictions, referred to as exceptions. Exceptions that are appended to a license modify the standard and widely understood terms of the original license and are important factor to consider for license compliance analysis. Using a synthetic data set, they utilised Weka[26] to experiment classifying license exceptions using the following machine learning techniques (decision trees, Naïve-Bayes, Random Forest and State Vector Machines).

Except for the work of Vendome *et al.* [42], each tools presented throughout this section share a single commonality, which is that they all provide methods for identifying licenses within source code. Some also looked for files which could be considered as a license in the case of Osler. However, ScanCode is the most comprehensive as it extends the search of licenses further and is capable of identifying licenses inside binary files.

[23]ohcount - https://www.openhub.net/p/ohcount

[24]https://sourceforge.net/projects/oslc/

[25]as of February 2018

[26]Weka - Waikato Environment for Knowledge Analysis is a suite of machine learning software written in Java, developed at the University of Waikato, New Zealand.

Interestingly, each of the related works approach the detection of licenses differently. Some have opted to use artificial intelligence and machine learning for classification such as FOSSID or the work of Vendome *et al.* [42], others have utilised rules based on keyword matching. Although the results presented in Vendome *et al.* [42] looked promising, machine learners in the context of licenses detection are not as easily extended when compared to rule-based and keyword identification approaches. Furthermore, a large and robust dataset would be required for training and a multi-class machine leaner capable of assigning over 380 labels. That said, introducing new rules could just as easily have a negative impact and would require vigorous testing. Finally, we have not observed any tool during our search which also considers software documentation as a potential source of licenses. With regard to CROSSMINER's requirements we have not found a suitable tool which satisfies both our requirements and complies with our own licensing guidelines.

2.4.2 Dataset

During the search for a suitable data set, we discovered 2 viable data sets from reputable open source organisations. The first source of data we considered was developed by Open Source Initiative (OSI) [27]. OSI's mission is to actively maintain the "Open Source Definition" for the open source community by approving licences that are compliant with the Open Source Definition. At the time of writing, there are a total 97 (82 active and 15 retired/superseded) licences approved by OSI. Licenses are presented on-line and have to be manually extracted. Resulting in the dataset consisting of a license name and text from the license body.

The second dataset we considered was "SPDX License List"[28] developed by the SPDX workgroup[29]. Unlike the OSI dataset, this list is composed of 389 (single) licences commonly used in free, open source and other collaborative software or documentation. In addition to just the licence text, the data set provides meta-data associated with licence which included elements such as; SPDX identifier, its deprecated status, If it is OSI compliant, If it is free software (Fsf Libre), licence body template, licence header template, licence comments, Relationship to other licences. Furthermore, the data is made freely accessible and is available in various open source formats including JSON. It is worth noting that this data set is also provided with some guidelines[30] on how the data should be used.

We opted to use the SPDX Licence List over the OSI data as this was superior from may different perspectives. Not only was the number of licences greater, the data was structured and well-defined providing us with the capability to recognise licence bodies and license headers too. However, this data set does not contain any instances of multi-license licenses.

2.4.3 Methodology

Our intention for CROSSMINER is to provide a license analyser tool which is capable of detecting and specifying open source licenses found within documentation.

We proposed to experiment detecting open source licenses using a language model. A language model, by definition, refers to the probability distribution of word sequences[31] in a given document or corpus[22]. Language modelling has been applied in areas of NLP were predicting upcoming words or the likelihood of a sentence

[27]Open Source Initiative - *https://opensource.org/licenses*

[28]SPDX Licence List - *Available at : https://github.com/spdx/license-list-data*

[29]The SPDX workgroup is hosted by the Linux foundation and consists of representatives from more than 20 organisations

[30]SPDX License List Guidelines - https://spdx.org/spdx-license-list/matching-guidelines

[31]In the context of this work a sequence of words refers to n-Grams.

is vital for completing a task such as in speech/hand writing recognition, spell checking and predictive text. The decision of using language models, instead of any other machine learning approach, is that we do not have enough data to train, in the sense that once we have observed a license text we might have seen all the licenses of the same time. Furthermore, by recollecting licenses that are actually used by software projects, a classifier might learn to differentiate them only by elements such as date or names of organisations. In fact, we have considered the detection of licenses as a language identification problem, where for a given text it is necessary to calculate the probability of being written in specific languages.

Language models can be computed via numerous techniques such as Hidden Markov Models (HMM), Trigram frequency vectors and n-grams based text categorisation [30]. We have opted to use n-grams based text categorisation approach for computing our language models, which in turn is based on conditional probabilities. To estimate the conditional probability of an n-gram, we utilised *Maximum Likelihood estimation* (MLE). MLE estimates the conditional probability of an n-gram model by "dividing the observed frequency of a particular sequence by the observed frequency of a prefix" [22] and is achieved using formula presented in Equation 7

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_N)}{C(w_{n-N+1}^{n-1})} \tag{7}$$

where $P(w_n|w_{n-N+1}^{n-1})$ is the conditional probability of seeing word w_n after having observed a sequence of words (w_{n-N+1}^{n-1}); $C(w_{n-N+1}^{n-1}w_N)$ is the number of times that the sequence of words $w_{n-N+1}^{n-1}w_N$ has been seen in a corpus and $C(w_{n-N+1}^{n-1})$ is the number of times this collection of words has been observed in the same corpus.

The probability of a text, i.e. a collection of n-grams, is calculated using the *Chain Rule Probability* as shown in Equation 8, where we multiple the conditional probabilities of each n-gram present in the text.

$$P(X_1...X^n) = P(X_1)P(X_2|X_1)P(X_3|X_1^2)...P(X_n|X_1^{n-1}) \tag{8}$$

One of the limitations of language models is that they are sensitive to unseen n-grams, i.e. sequence of words that did not occur in the corpus in which the language model was build. Unseen n-grams causes that the probability of a text is zero, because the conditional probability of an unseen n-gram is zero. To surpass this limitation, there are methods, such as Laplace Smoothing, that consists in smoothing the language model, in other words, to remove a small portion of the probability of every seen n-gram and giving it to a conditional probability that will represent unseen n-grams [22].

To solve the cases of unseen n-grams in the license analyser, we have used a naïve approach that consists in giving to every unseen n-gram a conditional probability of $log(100)$. This decision is based on the fact that this probability will be used only to differentiate between languages models, but most importantly to prevent that languages models with fewer n-grams produce the greatest probability values.

The language models used for the license anayser were generated by processing the data set regarding *SPDX License List* (see Section2.4.2). Therefore, we incorporated into our methodology SPDX's guidelines on how this data should be processed to ensure that matches are consistent to avoid potential non-matches and confusion for end users. In Table 11 that follows we describe the conditions of use and the actions we have take to comply with the guidelines that are applicable to this work.

Table 11: Methodology : Conditions and Actions

	Condition	Action
License Name	To avoid a license mismatch merely because the name or title of the license is different than how the license is usually referred to or different than the SPDX full name.	Any license capable of being detected by the license analyser should only be identified by their SPDX Full Name only.
Capitalisation	To avoid potential non-matches due to the use of upper case and lower case letters	All text that is processed should treat all letters as lower case
White space	To avoid the possibility of a non-match due to different spacing of words, line breaks, or paragraphs.	All white space will be considered as a single blank space.
Punctuation	Because punctuation can change the meaning of a sentence, punctuation needs to be included in the matching process	All punctuation is included as-is in the source.

The methodology consisted of two stages, presented in Section 2.4.3.1 and Section 2.4.3.2, respectively. The first focuses on the formulation of a license hierarchy and the second focuses on language modelling.

2.4.3.1 Creating a Grouped License Hierarchy

Licenses where grouped automatically based upon license name to form. Each license group consisted of a minimum of one license text and where applicable license revisions and license headers.The objective creating the hierarchy was to reduce the time required for computing scores for each model as a threshold is used to disregard low scoring groups, license groups. This processed resulted in a total of 266 license groups[32]. Presented below in Table 12 is a sample taken from the hierarchy consisting for example the license groups which is concerned with Apache, EPL and NASA licenses.

Table 12: Grouped License Hierarchy Example

Apache Licenses	**Eclipse Public License**	**NASA License**
Apache License v1.0 Apache License v1.1 Apache License v2.0 Apache License v2.0 (header)	Eclipse Public License v1.0 Eclipse Public License v2.0	NASA v1.3

2.4.3.2 Processing: Creating a language model

Each of the 383 JSON documents associated with licenses within the hierarchy was subjected to the same procedure described below. Note text refers to the *licenseText* and where applicable *standardLicenseHeader* fields found within the JSON document.

1. To ensure that we are compliant with the guidelines defined by SPDX the text was subjected to a pre-processing stage which consisted of the actions described in Table 11.
2. Using the pre-processed text from step 1, a language model was computed for the associated license text based upon trigrams and stored.

[32]Due to its size the complete license hierarchy has been omitted from this deliverable

2.4.4 Implementation

The License Analyser implemented into CROSSMINER has been developed using the methodology described in Section 2.4.3 and was built using the *Prediction* Manager[33] and the third party library Jackson[34]. The License Analyser provides CROSSMINER with the ability to detect over 380 open source licences within plain text sources, which includes software documentation, source code and configuration files. It works by subjecting the input to the same processes described in Table 11. It then proceeds to process the collection of trigrams from the text, first calculating for each license group a score. Groups which surpass a threshold then have a score computed for each license and license header present. It is important to note for unknown n-grams we implemented a "add-on" smoothing method which assigned a static value. In both cases, scores are calculated by accumulating the probabilities of known trigrams and unknown trigrams.

The License Analyser accepts a *String* or a *List<String>* as input and returns a *LicensePrediction* or *List<LicensePrediction>*. Table 13 defines the fields present in the License Prediction object.

Table 13: LicensePrediction object description

Variable Name	**Type**	**Description**
licenseFound	*Boolean*	Flags true if a licence has or false if a license has not been found.
isGroup	*Boolean*	Flag true if a the prediction is a license group or false if the prediction is not a license group.
isHeader	*Boolean*	Flag true if a the prediction is a header or false if the prediction is not a header group.
licenseName	*String*	If a license is found the SPDX full license name is provided
licenseGroup	*String*	The name given to the license group (this is populated only if a license is found)
score	*Double*	The score represents the confidence value awarded by the language model) closer to 0 indicates higher accuracy.
nGramsMatchedPercent	*Double*	The percentage of ngrams that matched perfectly with the language model for the predicted license.

The License Analyser is also accompanied by two other tools. The first tool extends the use of the dataset, by allowing software developers to query the hierarchy model for the body and header templates of known licences, so they can be quickly added into source code or documentation during development or access the licenses' associated meta-data to obtain additional information concerning the licence. The second tools enable new versions of the language, license hierarchy and licence statistic models to be generated when new licenses have been added to the SPDX License list.

[33]Prediction Manager was developed during Deliverable 3.4

[34]Jackson - https://github.com/FasterXML/jackson

2.4.5 Evaluation

This section presents the evaluation of the License Analyser. More specifically, we provide details surrounding the data set, metrics and evaluation settings used to evaluate the License analyser. This section concludes with a presentation and discussion of the results.

2.4.5.1 Testing Dataset

To evaluate the License analyser, we chose to exploit the same testing corpus[35] developed by nexB to evaluate ScanCode license detection tool. Several factors contributed to our decision to use this data set. Out of the datasets that we discovered it was the largest. Second it consisted of a mixture of sources of textual data related to software engineering and finally, it was the only that was a dataset that was annotated.

The nexB dataset consists of **1191** text-based files in various file formats. Each file has a corresponding annotation file in YML format, which contains meta-data that includes information such as *License Expressions* and *Notes* [36]. A *License Expressions* refers to the license or licenses that have been disclosed with the file. These will be referred to as *annotation labels* for the remainder of the evaluation. The *Notes* field includes miscellaneous comments left by nexB developers and this is also where they denote that a file contains no license.

However, in order to effectively utilise the nexB data required addressing several issues.

- The first relates to how we should handle licenses unknown to the License Analyser. As discussed in Section 2.4.2, the License Analyser from a theoretical perspective is capable of detecting 389 OSS licenses. However, the *scancode-toolkit* is also capable of detecting propitiatory licenses which our tool has not been trained to detect. Therefore, for this evaluation we treat all files with unknown licenses as if they did not contain one, i.e. - *no license found*.

- The second issue is related to the naming convention used. In the nexB dataset, annotation labels exist for OSS and priority licenses since the *scancode-toolkit* was designed to detect both. However, upon initial investigation, the naming convention used for the annotation labels do not align with those included within the SPDX License List with respects to the naming convention or guidelines. For example, BSD family of licenses names also are known by different aliases throughout the open source community. For example the BSD 2 Clause License is also referred to as *FreeBSD License* or *Simplified BSD License*[37]. In the case of the nexB dataset it is annotated as *FreeBSD*. However, in the SPDX License List it is identified as *BSD-2-Clause*.

- The final issue is regarding annotation labels for files with multi-license licenses. Within the nexB dataset, these are identified by the presence of the keyword *AND* for example:*gpl-2.0-plus AND lgpl-2.1-plus AND mpl-1.1*. Furthermore, the corpus also identifies known licenses which include custom expressions and declarations; treating them as a unique annotation label by appending the keywords *expression* and *declaration* to the end of the label.

Therefore to address the challenges discussed above, it was important to pre-process the nexB data set to define a mapping procedure to allow a comparison between the output from the License Analyser and expected

[35]nexB testing corpus can be located here:
https://github.com/nexB/scancode-toolkit/tree/develop/tests/licensedcode/data/licenses
[36]Not all meta-data included within the annotation file is useful for this evaluation
[37]BSD Licenses Wikipedia Article: `https://en.wikipedia.org/wiki/BSD_licenses`

annotation labels. We addressed these challenges by subjecting, annotation labels to a normalisation procedure, similar to that described in Table 11. This removed all spaces, symbols and punctuation from the name as-well-as convert it into lower-case. Additionally, rules were also included in the normalisation method to address the remaining outliers. Annotation labels were then also parsed to detect keywords such as *AND, OR, EXPRESSION* and *DECLARATION*, converting them into multiple annotation labels. Finally, unknown licenses annotations were converted to *no license found*.

After the normalisation procedure defined above, the testing corpus contains a total of **95** known labels from SPDX License List(since all unknown licenses were converted).

2.4.5.2 Evaluation Metrics

The results obtained will be evaluated based from various perspectives using the following evaluation metrics described below:

Precision : measures the ratio of correct positive predictions to the total predicted positives. It is calculated using Equation 9. Where TP is the number of true positives, and FN is the frequency of false negatives.

$$\text{Precision} = \frac{TP}{TP + FN} \tag{9}$$

Recall : measures the models ability to identify all relevant instances. Recall is calculated using Equation 10 below; where TP is the number of true positives, and FP is the number of false positives.

$$\text{Recall} = \frac{TP}{TP + FP} \tag{10}$$

F-Score : The F-Score or F-1 is the harmonic mean of Precision & Recall and measures the models accuracy. It is calculate using the Equation 11 below.

$$\text{F-Score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{11}$$

2.4.5.3 Evaluation Settings

Then each file within the testing corpus will be subjected to the same evaluation procedure as described in the steps below.

1. Extraction of *raw text* from the file document, using Apache Tika as discussed in Section 2.2.2.
2. Convert *raw text* into *plain text* using the Plain Text Converter developed in Deliverable 3.3
3. Pass the *plain text* to the license analyser (this will pre-process the text based upon the actions described in Table 12) and return a classification instance. The instance will be processed and the resulted stored. Please note that a result is considered is considered correct if the output from the License Analyser is present in the list of annotation labels for that particular document.

2.4.5.4 Results

The results per evaluation metric were computed for the language modelling approached used with the License Analyser. For comparison, we also computed values for the *Most Frequent Class* (MFC) and *Random Class* baselines (RC). The Most Frequent Class baseline, considers all predictions to be that of the most frequent class present within the dataset, whereas the Random Class assigns a prediction at random. The results for all are presented in Table 14.

Table 14: Licence Analyser Evaluation Results

	Precision	***Recall***	***F-Score***
Random Class Baseline	0.002	1.000	0.003
Most Frequent Class Baseline	0.147	1.000	0.259
Language Modelling	0.205	0.810	0.327

It is important to consider each evaluation metric in order to effectively evaluate the language modelling approach from varying perspectives. Each of the tests performed similarly with respects to *Recall* shared a commonality in which each had a considerably high recall rate. This means suggests that each test was inclusive. However, when also taking into consideration the precision, it identifies that many of the predictions made were actually incorrect.

Arguably the most important metric to consider is the F-Score as this is an indicator of the models' overall accuracy and takes into consideration both precision and recall. As expected the RC baseline with regards to F-Score had the worst performance, this is attributed to there being over 380 potential classes available to predict and only 96 are found within the data set. The MFC baseline is more performing in comparison RC baseline. However, the language modelling approach shows a marginal (+ 0.068%) improvement over the MFC.

Although not ideal, the language model approach is more performing than both the RC and MFC baselines. However, there is still room for considerable improvement.

2.4.5.5 Discussion

Detecting licenses is not a straight forward task. The results presented in Section 2.4.5.4 represent our initial evaluation of the current License Analyser. All though there is plenty of work surrounding the detection of licenses in source code, to the best of our knowledge, there is no work in scientific, commercial or open source communities which identifies licenses in documentation related to software.

One clear difficulty we faced is related to the limited availability of data-sets that are both balanced and large enough for training and testing of a classifier. The data sets found during the course of this work were all considerably small in comparison to a large number of OSS licenses. To put things into some perspective, the SPDX dataset was the most comprehensive list we discovered, containing over 380 commonly found OSS licenses. Each language model that we trained was only on a single document that consisted of the license template. Creating a dataset from scratch, that is annotated, balanced and large enough, requires a significant amount of investment with respects to time and resources and was not feasible for this deliverable.

During the normalisation procedure applied to the nexB dataset, we transformed all annotations that our model is not trained on i.e. proprietary licenses, to have the annotation *no license found*. However, in hindsight, this decision may have been naive, as theoretically these files still contain some kind of license. We hypothesise that there may be some overlapping between the vocabulary used in proprietary licenses and OSS licenses since both types are indeed legal documents and the terminology used is more restrictive. This hypothesis is also supported by the outcomes of some informal testing performed during the development of the License Analyser. In one particular case, we used a document which contained an MPL (Mozilla Public License) and it was classified incorrectly as NPL (Netscape Public License). However, when we investigated it was discovered that the scores computed by the tool were almost identical, suggesting that both licenses shared a common vocabulary. Upon inspection there where only two words that distinguished one license from another, making it difficult to classify. Therefore further work is required in order to address or implement methods to reduce the impact of overlapping vocabulary between language models.

As with many of the tools discussed in Section 2.4.1, the License Analyser attempts to identify the exact license present within a document. More specifically, it outputs a single label that received the best score. As mentioned in the introduction to this section, this knowledge is important for assessing if an OSS project is compliant with the objectives of the software being developed. However, in future work, it may also be worth considering the possibility of returning multiple labels in a similar approach to that used in the *go-License Detector*, preferring coverage over single incorrect predictions to assist developers and managers in the identification process; since are a legal document and no classier is right 100% of the time. Again this would require further exploration and testing.

2.4.6 Conclusion

Understanding the type of license used by open source projects is a vital factor for software engineers and product managers. Whether it is during the evaluation of an OSS project to determine if it is compatible with their objectives, to understand their legal obligations when using the works of other or to select a compatible license for their derivative work. Unlike other works discussed, the objective of the License Analyser for CROSSMINER was to design and develop a tool which is capable of analysing textual sources such as software documentation to data-mine the existence of license and determine what license it is based upon the language used in order to fulfil requirement U115 presented in Table 1. Although the results are not ideal, the language models are more performing than both the random class and most frequent class baselines. However, further evaluation, optimisation and work in the near future still required to improve its performance.

3 Identification of API changes in textual sources

APIs can be in constant evolution, for example, they can create new methods in order to support new features, or modify current ones to improve outcomes, even remove methods that are unsupported. Frequently, API changes are for good, however, in occasions these changes come with a price, products that use old versions of an API and want to migrate to more recent version might break. The approach that users and developers tend to follow, when migrations problems arise, is to look at the documentation or search on the Internet. However, to search a solution in question answering websites, such as Stack Overflow, is not only time-consuming, it can be erroneous or without the expected quality for a good understanding of the necessary changes to do [31, 27]. Even more, in occasions the software documentation do not provide enough information for dealing with an API migration [39].

To solve the problem described previously, the scientific community has worked on different approaches to provide users and developers with methods that can make easier the implementation of changes in APIs, such as those described in Deliverable 2.7[38]. In this part of Deliverable 3.5, we present the work that has been product from the collaboration between Centrum Wiskunde & Informatica (WP2) and Edge Hill University (WP3) for assisting software developers and users in migration API issues. Specifically, we explain how the detection of changes in Java APIs can be used along NLP tools to determine which discussions and posts talk about migration problems. The goal is to provide information to developers regarding issues that users are having due to changes in an API, and in consequence, encourage them to improve their software documentation, while to users present them with discussions that could help them to solve API migration issues.

3.1 Detecting changes in Java APIs

For detecting changes occurring in a Java API, we are using the framework and tool developed by our partners from Centrum Wiskunde & Informatica, called *Maracas*. This tool has for objective the analysis of Java APIs through the time and assist developers in the process of migration from one version of an API to a more recent one. Maracas, more specifically, indicates which elements, i.e. packages, methods and fields, in a Java API have been added, changed or removed. As well, it determines the possible effect of these changes on software that implements the modified API. See Deliverable 2.7 and Deliverable 2.8, for a more detailed explanation of Maracas.

In this deliverable, we make use Maracas to get the list of APIs' elements that changed and create searching patterns that can be used to detect textual sources describing topics regarding the migration of API versions. Specifically, every day of analysis in CROSSMINER, Maracas return us a delta which contains the elements changes in the API (See Deliverable 2.8), from which we extract the data that will be used in the next process.

3.2 Searching and matching migration problems

In order to find migration problems within discussions from all our different sources of information, such as issue trackers or mailing lists, we have designed a method that consists of analysing titles and (mail) subjects, but also the labels defined by a clustering method in order to select which discussions, e.g. bugs, emails, forums posts, are related to a migration problem. Then, for each selected discussion, we look for the names of methods that were obtained by Maracas. In the following paragraphs we explain in detail the approach followed.

[38]Deliverable 2.7 was developed by *Centrum Wiskunde & Informatica* (CWI)

In first place, if the textual source to analyse, e.g. issue tracker, communication channel, contains a subject or title name, we look for patterns that could lead us to discussions related to API. These patterns have been defined manually and can express aspects such as “migration problem”, “how to upgrade to”, “method deprecated”; moreover, these patterns are based on regular expressions.

As not all the sources to analyse contain a title or subject, and we know that the analysis of subjects and titles is not 100% accurate[39], we decided to take into account as well the output of the clustering tool used in CROSSMINER. Specifically, we make use of Lingo, a clustering tool that for a group of texts it generates clusters but most importantly labels as described in Deliverable 3.4. The generated labels, which are a concatenation of keywords, are then searched for patterns that could indicate that a group of texts are related to a migration issue.

Each of the API changes found by Maracas, will be searched in the list of texts and discussion, that was obtained with the previous approaches, and that might be related to migration issues. This search is done using a simple match of words in text. If a match is found, we mark the corresponding text or discussion with a flag that indicates that the topic might have revolved to a migration problem.

[39]In occasions, it can happen as well, that a user has a migration problem, but it is not aware of it, thus the title or subject do not make any reference to this.

4 Recommendation of snippets and discussions

A main characteristic of CROSSMINER is the possibility of accessing different types of information without having to leave Eclipse IDE.

In this deliverable, we present a recommender of code snippets and discussions that complement the work done by our partner from the University of L'Aquila. This recommender, unlike those presented in WP6, is based on the data that is retrieved, processed and indexed by CROSSMINER. Therefore, it can cover a large variety of sources, such as bug tracking systems, mailing lists, instant conversations, software documentation, question answering platforms and social media. Furthermore, it has been designed to be general enough to be used to retrieve data regarding different programming languages.

The approach used for this recommender consists in generating firstly a query based on the code that an Eclipse IDE user is implementing at a certain moment. With this query, we retrieve the most relevant entries from CROSSMINER indexes using ElasticSearch. Then, we focus on the natural language aspects of the retrieved entries, in order to create a query that is based on natural language elements only. We decided to utilise this approach due to two reasons. In first place, our partners from Univeristy of L'Aquila have developed a series of recommenders that are based on the analysis of code. In second place, the expertise of Edge Hill University in CROSSMINER is the processing of natural language. Therefore, we considered appropriate to boost the queries for this recommender from a natural language perspective instead from a code one. In the following sections, we explain in detail the creation of each type of query and which is the method used for ranking the retrieved elements.

4.1 Query based on code elements

As we indicated previously, the recommender uses in first instance a query that is based on the code that is being developed by an Eclipse IDE user. This means that we process the code and convert the code into a query that can be understood by ElasticSearch.

To have access to the code being developed by a programmer, we worked with our partners from FrontEndArt and University of L'Aquila. CROSSMINER knowledge base can connect directly to Eclipse IDE through the tools developed by FrontEndArt, while our recommender can have access to the code being developed through CROSSMINER knowledge base. As we expect to process the code from different programming languages and not only from Java, we need as input for the recommender the name of the programming language being developed. Currently, our tool can support: *C*, *Java*, *JavaScript* and *PHP*. These languages were selected to fulfil our requirements U66, U67, U68 and U69.

Once the code has been introduced in the recommender, the next step consist in analysing the code and retrieve the tokens. For this task, we have decided to utilise ANTLR[40], a library that generates lexical analyser and parsers automatically given a grammar. More specifically, we downloaded the grammars that correspond to the programming languages supported by our recommender, i.e. C, Java, JavaScript and PHP, from ANTLR repository[41] and generated their respective lexical analyser. A lexical analyser is a tool that tokenise a text but also determines the type of each token. How detailed the type of each token is indicated by the lexical analyser will depend on the grammar. For example, the grammar downloaded for C, indicates that in `int myVar;` there is an element of type *int* and one of type *identifier*; the grammar for Java, in contrasts, will indicate that

[40]antlr.org
[41]github.com/antlr/grammars-v4

there are two elements of type *identifier*. A lexical analyser for code would be analogous in natural language processing to the output that would be obtained from using a tokeniser with a Part-of-Speech tagger.

After the tokenisation of the code, we filtered the tokens to get only those that would be related to names of variables, types of variables, names of libraries and names of methods. This was done by identifying the types of tokens that would be returned by the lexical analyser. In this aspect, we found out that all the generated lexical analysers, except the one for C, would name the type of this code elements as *identifier*; in the case of the lexical analyser for C, only the types of variables would receive a different name, but the rest would be names as *identifier*.

One advantage of using only a lexical analyser, instead of including as well a parser[42], is that the former are not affected by syntax errors, while the second ones are. Therefore, the recommender can still provide recommendations even if the code being programmed contains syntax errors.

In order to determine the relevance of the filtered tokens, we have decided to follow the approach used by our colleagues from L'Aquila in Deliverable 6.5. This consists in calculating how informative each token is, and which is the probability of seeing this token. These two values are the basis of Shannon's Entropy which is defined in Equation 12.

$$\text{Shannon's Entropy}(X) = \sum p(t) \times log(1/p(t) \tag{12}$$

where X is the text to analyse, t is a token from text X and $p(t)$ is the probability of occurring token t in X. The informativeness of a token is given by $log(1/p(t)$.

Thus, for each token, we obtain its probability $p(t)$ and its informativeness $log(1/p(t)$; the factor obtained from multiplying is the indicator of how relevant the token is. After all the relevance scores have been defined for each token, we normalise then between 1 and 4. This normalisation is done by splitting in 4 equal portions the sorted list of tokens by relevance. A score of 4, is given for first quarter of most representative tokens, while we give a score of 1 to the last quarter of the list that corresponds to the less representative tokens.

Once we have determined which tokens represent the code and their relevance, we need to construct a query that can be used in ElasticSearch. In this case, we decided that the best type of query would be a boosted one. A boosted query, is a request where weights are defined to fine-tune the relevance score of each index entry that the query retrieves.

```
import javax.swing.*;
public class ButtonExample {
public static void main(String[] args) {
         JFrame f=new JFrame("Button␣Example");
         JButton b=new JButton("Click␣Here");
         b.setBounds(50,100,95,30);
         f.add(b);
         f.setSize(400,400);
         f.setLayout(null);
         f.setVisible(true);
         }
}
```

Listing 1: Example of a code being developed that will be used as input for the recommender.

[42] A parser, depending on the grammar, can provide more information if the code is implementing a method or if it is calling a library.

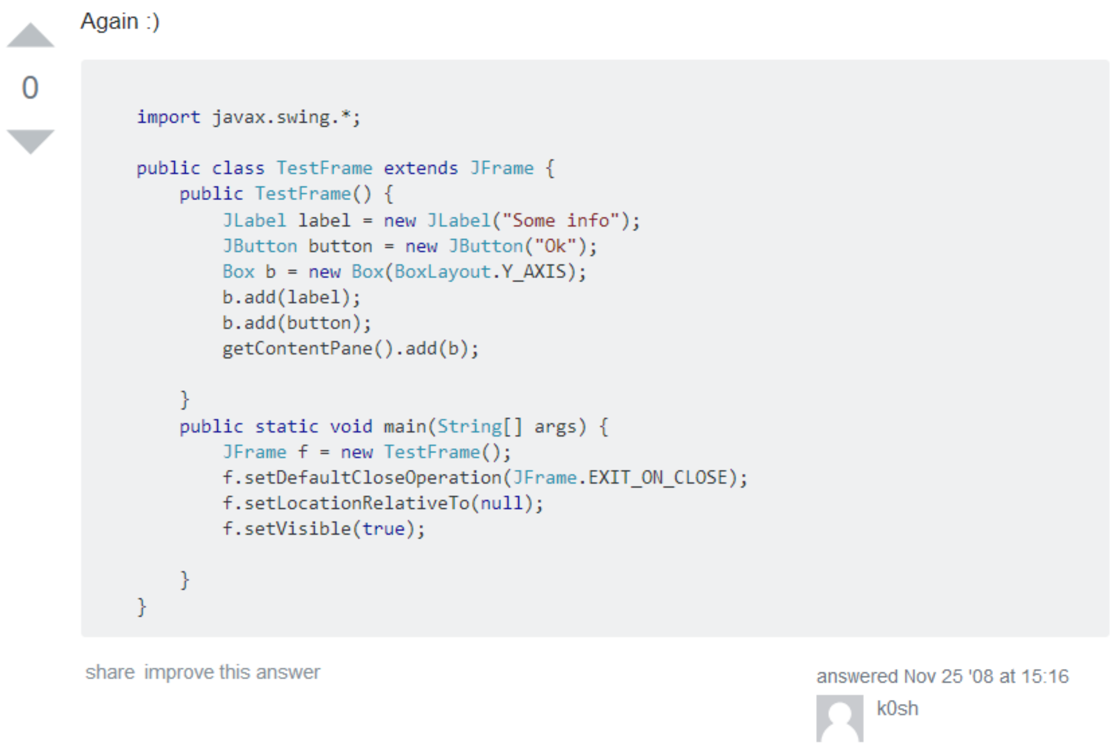
To further demonstrate how our tool works, let us consider that we are developing the code presented in Listing 1. The code is in Java and describes the creation of a button using *Swing*.

After processing the code described in Listing 1, we can generate the boosted query presented in Listing 2. In Listing 2, we can observe the different tokens from the code presented in Listing 1 which are boosted in accordance to how informative each token is, and what the probability of seeing this token is.

```
GET /_search
{
        "query": {
                "query_string" : {
                        "default_field" : "code",
                        "query" : "add^1.0 OR setVisible^1.0 OR b^2.0 OR
f^2.0 OR JButton^1.0 OR main^1.0 OR String^1.0 OR setLayout^1.0 OR
setBounds^1.0 OR args^1.0 OR swing^1.0 OR ButtonExample^1.0 OR
setSize^1.0 OR javax^1.0 OR JFrame^1.0"
                }
        }
}
```

Listing 2: Example of a boosted query, which is result from processing Listing 1.

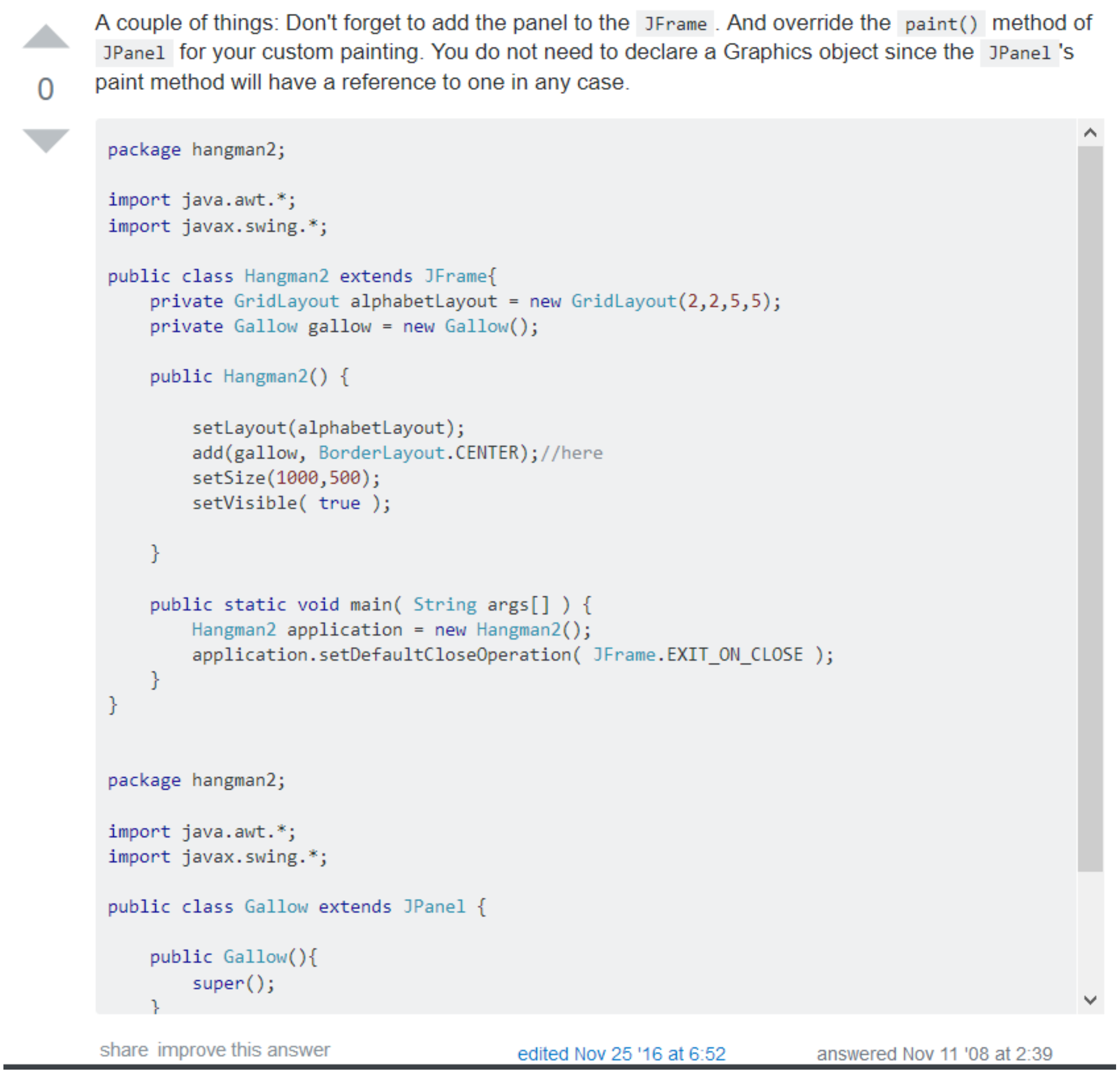
From this query we can retrieve a series of posts, such as the ones represented in Figure 4.1 and Figure 4.1.[43]


Again :)

```
import javax.swing.*;

public class TestFrame extends JFrame {
    public TestFrame() {
        JLabel label = new JLabel("Some info");
        JButton button = new JButton("Ok");
        Box b = new Box(BoxLayout.Y_AXIS);
        b.add(label);
        b.add(button);
        getContentPane().add(b);

    }
    public static void main(String[] args) {
        JFrame f = new TestFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setLocationRelativeTo(null);
        f.setVisible(true);

    }
}
```

Figure 4: One of the posts retrieved from quering ElasticSearch with the request presented in Listing 2

[43]stackoverflow.com/questions/311876/ and stackoverflow.com/questions/279781/



0

A couple of things: Don't forget to add the panel to the `JFrame`. And override the `paint()` method of `JPanel` for your custom painting. You do not need to declare a Graphics object since the `JPanel`'s paint method will have a reference to one in any case.

```
package hangman2;

import java.awt.*;
import javax.swing.*;

public class Hangman2 extends JFrame{
    private GridLayout alphabetLayout = new GridLayout(2,2,5,5);
    private Gallow gallow = new Gallow();

    public Hangman2() {

        setLayout(alphabetLayout);
        add(gallow, BorderLayout.CENTER);//here
        setSize(1000,500);
        setVisible( true );

    }

    public static void main( String args[] ) {
        Hangman2 application = new Hangman2();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}


package hangman2;

import java.awt.*;
import javax.swing.*;

public class Gallow extends JPanel {

    public Gallow(){
        super();
    }
```

share improve this answer   edited Nov 25 '16 at 6:52   answered Nov 11 '08 at 2:39

Figure 5: Another pos retrieved from quering ElasticSearch with the request presented in Listing 2

4.2 Query based on textual elements

In order to generate the query based on textual elements, we decided to process the index entries that were retrieved by the query based on code elements. For each index entry retrieved, we extract the field related to plain text and apply an algorithm of keyword extraction. More specifically, we extract the keywords of each text processed using *Rapid Automatic Keyword Extraction (RAKE)* [32].

RAKE is a fast, unsupervised, domain and language independent algorithm for extracting keywords, either single or multi-word, from individual documents. The algorithm of RAKE is quite simple and starts by splitting the input text into candidate keywords. This is achieved by first tokenising the text[44] and storing the tokens in an array. Then the array of tokens is split in sequences of contiguous words that are not separated either by a stop-word or punctuation mark; these contiguous tokens are the candidate keywords. The following step

[44] A very simple tokeniser can be use, such as those based on white spaces and punctuation marks.

Table 15: Sample of the keywords extracted by RAKE for the posts presented in Figure 4.1 and Figure 4.1.

Post from Figure 4.1		Post from Figure 4.1	
Keywords	Score	Keywords	Score
public class testframe extends jframe	18.62	private gridlayout alphabetlayout = new gridlayout	14.75
public static void main	17.37	public static void main	14.50
jbutton button = new jbutton	11.61	private gallow gallow = new gallow	13.44
jlabel label = new jlabel	11.61	public class gallow extends jpanel	12.81
= new box	7.28	public class hangman2 extends jframe	11.48
= new testframe	6.53	hangman2 application = new hangman2	10.71
string[] args	6.0	graphics object since	10.0
import javax	6.0	public void paint	7.0
public testframe	4.62	string args[]	6.0
box	3.0	custom painting	6.0
setvisible	2.0	paint method	5.0

is to create a matrix of co-occurence of words, which consists of counting the number of times that each word occurred in the text, and how frequent each of this word occurred with others. Then, it is necessary to calculate the score of each word by dividing the degree of a word (i.e. number of times the word is observed, plus number of times the word occur with another in the candidate keywords) by the frequency of the word (number of times the word is observed). The score of each candidate keyword is the sum of scores of the words that compose the candidate keyword. To obtain the keywords, it is necessary to sort the candidate keywords, from the greater to the lesser score, and then only consider the first T candidate keyword. The value of T, according to the authors, is set as one third of the number of words in the co-occurence graph.

We present in Table 15, a sample of the keywords extracted by RAKE for the post presented in Figure 4.1 and Figure 4.1. As we cab observe in Table 15, RAKE is able to extract multi-word terms, and although it is created for natural language, it is also possible to apply it in mixed entries with code.

Although RAKE can determine the relevance of each keyword, it is done within the scope of only one text entry. However, we can have multiple entries returned by the code-based query and it is necessary to determine globally, which are the most relevant keywords. Thus, in first place, we decided to normalise, between 1 and 0, the score given by ElasticSearch to each retrieved index entry. The normalisation consisted in dividing every score by the maximum score retrieved. The normalised score of the retrieved index entry is then used to multiply the weight of each keyword returned by RAKE. This is done to merge, the output of RAKE and the normalised relevance score provided by ElasticSearch into a single score which considers how relevant the text was for the query and how relevant the keyword was for the text.

Once this has been done, we join all the list of keywords into a unique list of keywords. If a keyword appears multiple times, we merge all the occurrences while summing up the scores. To determine which are the most relevant keywords of all the keywords extracted, we follow a similar approach used by RAKE to determine the final keywords of a text. In other words, we sort the keywords according to their weight, and we select one third of all the keywords found in all the retrieved index entries.

With the selected keywords and weights, we construct a new boosted query that is sent to ElasticSearch. The results returned are a based on a query which is expected to provide more contextually relevant results from on-line discussions, which may or may not contain code, to the end user in relation to the code that is being developed in Eclipse IDE.

5 New readers for communication channels

In a typical natural language processing (NLP) workflow, a reader is the first component. Its role in CROSSMINER is to retrieve textual information related to an open source software project, hence, Task 3.2 which focuses on searching and reading NLP sources.

As noted in D3.4, CROSSMINER was upgraded to support a total of 12 sources of text. Since then, it has undergone several upgrades to include 5 additional readers; two of them have been described in Section 2.1. The final list of textual data sources supported by CROSSMINER is presented in Table 16. The aim is to further enrich the knowledge base and thus, widen the scope of recommendations provided to users. Detailed description of the new readers are presented in sections 5.2, 5.1, and 5.3.

Source	**Status**	**Package name**
Bitbucket	Existing	org.eclipse.scava.platform.bugtrackingsystem.bitbucket
Bugzilla	Existing	org.eclipse.scava.platform.bugtrackingsystem.bugzilla
GitHub	Existing	org.eclipse.scava.platform.bugtrackingsystem.github
GitLab	Existing	org.eclipse.scava.platform.bugtrackingsystem.gitlab
JIRA	Existing	org.eclipse.scava.platform.bugtrackingsystem.jira
Mantis BT	Existing	org.eclipse.scava.platform.bugtrackingsystem.mantis
Redmine	Existing	org.eclipse.scava.platform.bugtrackingsystem.redmine
Sourceforge	Existing	org.eclipse.scava.platform.bugtrackingsystem.sourceforge
Eclipse Forums	Existing	org.eclipse.scava.platform.communicationchannel.eclipseforums
Zendesk	Existing	org.eclipse.scava.platform.communicationchannel.zendesk
NNTP	Existing	org.eclipse.scava.platform.communicationchannel.nntp
Sourceforge	Existing	org.eclipse.scava.platform.communicationchannel.sourceforge
Sympa Emails	New	org.eclipse.scava.platform.communicationchannel.sympa
Mbox Emails	New	org.eclipse.scava.platform.communicationchannel.mbox
IRC	New	org.eclipse.scava.platform.communicationchannel.irc
Git-based Documentation	New	org.eclipse.scava.platform.documentation.gitbased
Systematic Documentation	New	org.eclipse.scava.platform.documentation.systematic

Table 16: Crossminer Readers

5.1 Sympa

Sympa[45] is an open source mailing list manager. It operates very similar to most email servers, where emails sent by users can be stored in archives. Sympa is a *Multipurpose Internet Mail Extensions (MIME)* compatible which simply means that it provides support for:

- Text in character sets other than ASCII
- Non-text attachments such as audio, video or images
- Message body with multiple parts such as replies
- Header information in non-ASCII character sets

CROSSMINER relies on a number natural language components, in order to compute quality metrics. This means that Sympa emails needs to be converted into plain text so that its content can be analysed. Thus, we

[45]https://www.sympa.org/

developed a reader, capable of reading and parsing Sympa emails stored in an archive. The reader uses *javax-mail*, *apache commons mail* and *apache commons net*; and currently supports compressed 'tarballs' archives, i.e., tar.gzip or tgz. The parameter set required to process a Sympa project on CROSSMINER are shown in Table 17. The URL may contain any number of log archives stored such that each folder within the log contains emails sent per day. This is to facilitate delta based analysis required by our use case partners.

Parameter	**Requirements**	**Description**
URL	Mandatory	The Log archive to be processed
Name	Mandatory	Name of the project e.g., Mbox, Sympa or Irc
Description	Mandatory	Description of the project
File extension	Mandatory	Log archive file extension e.g., .tgz
Authentication	Optional	Username and Password to access the log archive

Table 17: Sympa, Mbox and IRC Project Parameters

5.2 Mail Box (Mbox)

Mbox stands for mail box and refers to a file that contains a collection of multiple email messages as 7-bit ASCII text. Messages in Mbox are stored as a single text file of concatenated e-mail messages where each email message is stored after another, starting with the 'From' header. Mbox files were predominantly used by Unix hosts but are now supported by other email clients such as Apple Mail, Mozilla Thunderbird, Microsoft Entourage and Qualcomm Eudora.

Unlike the Internet protocols used for email exchange, the format used to store emails is not formally defined through the request for comments (RFC) standardisation mechanism. Email storage option has been entirely left to the developer of an email client which poses some issues when migrating between clients or in our case, reading the content. For example, Mbox generally stores email messages as 7-bit ASCII text such that each newly added message is preceded and terminated with a completely empty line, which makes it easy to determine the message body. However, this was not the case with the sample Mbox archive we used to develop and test the reader for CROSSMINER. For example, some messages were not preceded or terminated with an empty line and some were stored as UTF-8 instead of ASCII text, causing the reader to miss them completely. In addition, Mbox format uses a 'From' string to delimit email messages, and this can create ambiguities if an email message contains the same sequence in the message body.

To mitigate these issues, we used a pre-processing step that runs through the original Mbox file to identify and fix abnormalities. Its output is stored in a temporary file that is then forwarded to the main parser that reads the file content. Like Sympa, Mbox is MIME compatible so the reader uses *apache.james.mime4j* library for parsing content. Currently the reader supports compressed 'tarballs' archives, i.e., tar.gzip or tgz and the parameter set required to process an Mbox project on CROSSMINER is shown in Table 17. The URL may contain any number of log archives stored such that each folder within the log contains emails sent per day. This is to facilitate delta based analysis required by our use case partners.

5.3 Internet Relay Chat (IRC)

IRC is an application layer protocol based on a server and client model, that facilitates real time communication between people in textual form.

IRC servers accept and relay messages to connected users, who must run an IRC client either locally or web-based. There are many IRC networks on the internet, each with one or more servers working together to relay messages. Each network has many channels, commonly called rooms, where users can interact with each other. A channel usually have a specific topic, and a name that starts with a "#", such as "#crossminer"; although users may discuss other topics that interests them. Users can join channels they are interested in or even start their own channel. User messages can be logged by the server such that each message is time stamped and marked with the sender's username.

Although new applications now exist that provide similar functionalities such as Slack, IRC is time-tested and still popular in software communities and organisations. Therefore, CROSSMINER would benefit from analysing IRC messages about OSS in order to determine the level of support offered to users. We developed a delta based reader based on archived IRC message logs. Since each channel focusses on a specific topic, the reader assumes that all user messages within each channel are related to the specified topic.

The parameter set required to process an IRC project on CROSSMINER is shown in Table 17. The URL may contain any number of log archives stored such that each file within the log represents an IRC channel and its user messages per day. This is to facilitate delta based analysis required by our use case partners.

6 Metric Providers

In CROSSMINER, metric providers compute heuristics that enrich the knowledge base with useful information, that open source software developers can use to make informed decisions. The purpose of this section is to present all the new metrics integrated into CROSSMINER as part of Task 3.4. The remaining of this section is organised into 3 subsections, namely: Transient metrics, Historic metrics and Indexing metrics.

6.1 Transient Metrics

As the name suggests, transient metrics are used to calculate heuristics that are associated with a particular period in time, i.e. a day in the case of CROSSMINER. They are stored temporarily within the knowledge base and their output is passed as parameters in the calculation of other transient and historic metrics. The transient metrics developed as part of this deliverable, are related to two areas: commit messages and documentation analysis.

6.1.1 Transient Metrics for Commit Messages

The transient metrics presented in this section are associated with the processing of commit messages. These metrics process the input text and return the expected values as described in Table 18 .It should be noted, that for the extraction of topics, we are not processing the messages for detecting code elements. Commits messages tend to be very short, and they should contain mostly natural language rather than code.

Table 18: CROSSMINER Transient Metrics related to commits messages

Metric	**Description**
commits.message.plaintext	This metric pre-processes each commit message to get a split plain text version.
commits.message.references	This metrics search for references of commits or bugs within commit messages.
commits.message.topics	This metric computes topic clusters for each commit message.

6.1.2 Transient Metrics for Documentation

The transient metrics presented in this section are associated with the processing of documentation. These metrics process the input text and return the expected values as described in Table 19.

Metric	Description
documentation	This metric process the files returned from the documentation readers and extracts the body (in format HTML or text).
documentation.classification	This metric determines which type of documentation is present.
documentation.detectingcode	This metric processes the plain text from documentation and detects the portions corresponding to code and natural language.
documentation.plaintext	This metric process the body of each documentation entry and extracts the plain text.
documentation.readability	This metric calculates the readability of each documentation entry.
documentation.sentiment	This metric calculates the sentiment polarity of each documentation entry.

Table 19: CROSSMINER Transient Metrics related to Documentation

6.2 Historic Metrics

As the name suggests, historic metrics keep track of various heuristics associated with a specific open source project over its lifetime. In CROSSMINER, historic metrics are calculated based on information from transient metrics. The historic metrics developed as part of this deliverable, are related to two areas: commit messages and documentation analysis. These metrics are presented in Table 20.

Table 20: CROSSMINER Historic Metrics related to Commits and Documentations

Metric	Area	Description
commits.messages.topics	Commit	This metric computes the labels of topics (thematic clusters) in commit messages pushed by users in the last 30 days.
documentation.readability	Documentation	This metric stores the evolution of the documentation readability.
documentation.sentiment	Documentation	This metric stores the evolution of the documentation sentiment polarity.

6.3 Indexing Metrics

As the name suggests, indexing metrics are used to sort data from the various sources we process in CROSSMINER such as bug trackers, communication channels etc. Unlike the historic and transient metrics, indexing metric does not compute knowledge from the data sources. Rather, the main purpose is to store information produced from these sources (via the various transient metrics) in a way that can be easily retrieved through queries. The indexing metrics developed as part of this deliverable are presented in Table 21.

Table 21: CROSSMINER Indexing Metrics

Metric	**Description**
indexing.commits	This metric prepares and indexes documents relating to commits.
indexing.documentation	This metric prepares and indexes documents relating to documentations.

7 Risks & Limitations

In the following paragraphs we expose the limitations found for each of the tools developed in this deliverable. As well, we indicate the possible risks that entail the use of the methods here presented.

Concerning the Git-based documentation reader, its main limitation is the impossibility of downloading files that on their name contain illegal characters. This happens especially on the web version of GitHub Wikis, which allow naming a wiki entry with illegal characters. We cannot provide a solution for this limitation, but we expect that in the future Github will either convert the illegal characters or forbidden from the web version of the wiki creator.

With respect to the Systematic-documentation, we can name two limitations and one risk. One of the limitation is that the reader can only download the most recent version of the documentation. The second limitation is that the crawler implemented in CROSSMINER respect the robots configuration, which might lead to impossibility of downloading the files. The main risk of this reader is that if it is used too frequently, the reader can be banned from the server that stores the documentation. We have included in this reader some heuristics to prevent the second issue, in the sense that it will wait at least one actual day before crawling again the website containing the documentation.

Our tool for documentation classification has limitations as well. In first place, this tool is founded on the extraction of text from files. As we indicated in its respective section, this task is not easy to solve, as there can be multiple types of files formats and sometimes files contain internal errors that prevent the extraction of text. However, we expect that Apache Tika will determine and manage in the best way possible, files formats and tools to extract text correctly. Related to this, is the fact that depending on how the text has been formatted, e.g. double column, or the elements that contain, e.g. tables or images, the quality of the extracted text will be reduced. The only solution to this limitation is to recommend users, if possible, to select documentation in a format that stores data in XML or HTML formats, such as Microsoft Word, instead of streams or image formats, such as PDF or DJVU. In second place, the documentation classification uses headings and regular expressions to detect the possible types of documentation present in a file. If a file do not contain headings, or they are not detectable once they have been converted into text, the classifier is not being able to work. Furthermore, as our classifier is based in regular expressions, we might not be able do detect correctly certain patters. To prevent this last limitation, we have created as many regular expressions as possible, and we have made them as flexible as possible.

With regard to the License Analyser, there are three limitations. The first is that the tool is only capable of detecting OSS software licenses only meaning propriety licenses go undetected. The second issue relates to instances where there is some overlap between language models. This is attributed to licenses either being superseded by new (minor) revisions or that the basis of one license is used for another. For example, the Netscape Public License and Mozilla Public License have an overlap since the Netscape web-browser eventually became Mozilla Firefox and the only difference in the licenses are the name of the licenses. Finally, in its current form the license analyser is only capable of detecting a single license for each text passed to the tool. However, it should be noted that to mitigate this issue, the tool has the capability to process lists of texts in an attempt to identity instances where works have multiple licenses. As well, in the future, we expect to have backup language models using bigrams, to refine the detection when a trigram has not been found in the main language model.

The identification of API changes in textual sources presents the following limitations. As it happens with the documentation classification, the tool created for solving this current task is based on matching of patterns. This means, that one of the limitations is that we will not be able to identify all the possible cases that revolve around a migration issue. Although we try to prevent this, by providing different sources of text where to match these patterns, i.e. titles, subjects and cluster labels, these not ensure that there is going to be a match. In some cases, the titles or subjects might never contain keywords related to a migration issue, while the cluster label might contain other keywords that were considered as more representative to the analysed texts.

The recommender created in this deliverable will be limited by the quantity and quality of the data indexed by CROSSMINER. This aspect is related to the CROSSMINER users, as the greatest part of information that will be stored in CROSSMINER indexes, will come from the projects analysed by users. Therefore, if no projects are analysed or indexed, then the quality of the recommendations given by CROSSMINER regarding discussions and code snippets can be poor or null. However, we expect that CROSSMINER will be used for analysing several projects thus, the issue should only be present during a short time.

One limitation of SYMPA and MBOX readers is related to the processing of dates. In theory, there are standards that should be used for both mailing list services, however, as we observed during the development of the readers, in many cases there are inconsistencies. Normally, emails not adhering the standards would break the parsers, however, in order to reduce this limitation, we have created a series of rules and heursitics, that try to extract and format correctly the data present in emails from SYMPA and MBOX.

8 Conclusion

In this deliverable we have presented the progress done to fulfil the goals related to Task 3.4, which consists in recommending code snippets and on-line discussions relevant to code that it is being developed by a programmer. As well, we explained the work that was done to include new readers for communication channel sources, new metrics regarding the analysis of textual sources, such as commits messages and documentation, or the tools for searching discussions regarding API migration issues.

Specifically, we created two different readers for retrieving software documentation. One of the readers is for documentation stored in the format of a git repository, while the other reader is focused on documentation that is stored, generally speaking, in a website, from a blog to a webpage protected behind a username and password. We introduced in this deliverable, as well, three different tools for processing the documentation. In first place, we created a classifier based on heuristics that determine which are the types of documentation, e.g. *getting started* or *installation guide*, that are present within a documentation file. In second place, we have a tool that evaluates the readability of the documentation by computing the number of familiar words and sentences used in the text. Finally, we created a classifier that determines whether a documentation file contains a open source license, and if it is the case, it determines which it is.

We have also explored how to detect discussion, such as forums posts or software issues, that are related to API migrations issues. In this case, we have created a tool that search in discussions titles, email subjects and clustering labels, for patterns that could indicate the presence of migration issues. Furthermore, this work has been done along with our partners from Centrum Wiskunde & Informatica (WP2), which provide us with the changed elements from an API in order to improve the search of migration issues. This tool was developed to alert developers that users are having issues for migrating from one version of the library to another, and at the same time, propose to users, struggling with API migrations, discussions that could be of relevance for solving this kind of issues faster and easier.

Concerning the recommendation of code snippets and on-line discussions, we have created a recommender based on two types of knowledge, code and natural language. More specifically, we have created a tool that processes, in first instance, the code that it is being developed by a user, extracts the most relevant elements from it and queries CROSSMINER indexes the most related entries. In second instance, the tool processes the natural language text from the returned index entries to extract the most relevant keywords and query again CROSSMINER indexes based on natural languges features.

Finally, we present at the end of this deliverable, new readers that have been created to expand the sources of information that can be processes by CROSSMINER. At the same time, we explain the new metrics that have been created regarding the analysis of documentation files and commits messages.

In Table 22, we present a summary of the requirements explored in this deliverable along with their current implementation status in CROSSMINER.

Ref	Description	Priority	Status
D38	Shall collaborate with the source code mining work package to analyse documents that contain both natural language and code, e.g. documentation and bug reports.	Shall	●
D39	The knowledge base (Mining Cross-Project Relationships work package) shall provide the infrastructure for hosting the indexes populated by natural language analysis.	Shall	●
U35	Able to search documentation	Shall	●
U42	Able to detect in the data sources text referring to one or several bugs	Shall	●
U43	Able to detect in the data sources text referring to one or several commits	Shall	●
U45	Able to extract sentiment analysis from wikis	Should	●
U49	Able to detect in the data sources one or several commits hashes	Shall	●
U50	Able to list commits with bugs	Shall	●
U51	Able to list bugs with commits	Shall	●
U59	Able to identify code snippets that use old and new third-party API in forum threads concerning migration of the usage of the given third-party API	Shall	●
U62	Able to extract text from HTML and markdown to feed natural language analysis and identify code snippets	Shall	●
U63	Able to extract text from PDF to feed natural language analysis and identify code snippets	May	●
U64	Provides recommendations to add documentation commonly found in successful projects	Should	◐
U65	Provides recommendations to improve the structure of the documentation	Should	◐
U66	Able to analyse Java code snippets	Shall	●
U67	Able to analyse JavaScript code snippets	Should	●
U68	Able to analyse C code snippets	Shall	●
U69	Able to analyse PHP code snippets	Shall	●
U110	Able to analyse PDF documents	May	●
U111	Able to identify the documentation contains a Getting Started	Should	●
U112	Able to identify the documentation contains a User Guide	Should	●
U113	Able to identify the documentation contains a Developer Guide	Should	●
U114	Able to identify the documentation contains a Code Snippets	Should	●
U115	Able to analyse the documentation has a License	Shall	●
U116	Able to analyse readability of documentation	Shall	●
U185	Communication channel parsers use MBoxes	Shall	●
U188	Documentation parsers use data dumps	Shall	●

Table 22: WP3 Use Case Requirements related to Task 3.4
[Early stage: ○; Half done: ◐; Fully done: ●]

References

[1] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 235–241, January 2002.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software Documentation Issues Unveiled. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*, pages 1199–1210, Montreal, Quebec, Canada, 2019. IEEE Press.

[3] Amzon. Askalono, 2018.

[4] J.S. Chall and E. Dale. *Readability revisited: the new Dale-Chall readability formula*. Brookline Books, 1995.

[5] Meri Coleman and T. L. Liau. A computer readability formula designed for machine scoring. *Journal of Applied Psychology*, 60(2):283–284, 1975.

[6] Edgar Dale and Jeanne S. Chall. A formula for predicting readability. *Educational Research Bulletin*, 27(1):11–28, 1948.

[7] Loic Duros. GNU LibreJS. Technical report, 2016.

[8] A. Engelfriet. Choosing an open source license. *IEEE Software*, 27(1):48–49, Jan 2010.

[9] Liana Ermakova, Jean Valère Cossu, and Josiane Mothe. A survey on evaluation of summarization methods. *Information Processing & Management*, 56(5):1794 – 1814, 2019.

[10] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[11] FOSSID. Open Source Compliance - FOSSID, 2019.

[12] Free Software Foundation. Various Licenses and Comments about Them, 2019.

[13] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, Sebastopol, CA, USA, 3 edition, 2006.

[14] Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664 – 682, 2015.

[15] Daniel M. German, Yuki Manabe, and Katsuro Inoue. A sentence-matching method for automatic license identification of source code files. page 437, 2010.

[16] Github. Licensee, 2017.

[17] GitHub. Open Source Survey, 2017.

[18] Robert Gobeille. The fossology project. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 47–50, New York, NY, USA, 2008. ACM.

[19] Google. LicenseClassifier, 2017.

[20] R. Gunning. *The technique of clear writing*. McGraw-Hill, 1968.

[21] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. An Unsupervised Approach for Discovering Relevant Tutorial Fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, pages 38–48, Buenos Aires, Argentina, 2017. IEEE Press.

[22] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[23] Georgia M. Kapitsaki and Demetris Paschalides. Identifying Terms in Open Source Software License Texts. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2017-December:540–545, 2018.

[24] J.P. Kincaid. *Derivation of New Readability Formulas: (automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel*. Research Branch report. Chief of Naval Technical Training, Naval Air Station Memphis, 1975.

[25] J. Richard Landis and Gary G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977.

[26] Jan Lánsky and Michal Žemlička. Text compression: syllables. In Karel Richta and Jaroslav Pokorny, editors, *Proceedings of the Dateso 2005 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS*, volume 129, pages 32–45, Desná, Czech Republic, 2005.

[27] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, Trento, Italy, September 2012.

[28] NexB. nexB/scancode-toolkit.

[29] Osler. Osler Code Detect FAQs | Open source license checker tool.

[30] Muntsa Padro and Luis Padro. Comparing methods for language identification. *Procesamiento del lenguaje natural*, 2004.

[31] M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, November 2009.

[32] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. Automatic Keyword Extraction from Individual Documents. In Michael W. Berry and Jacob Kogan, editors, *Text Mining: Applications and Theory*, pages 1–20. John Wiley & Sons, Ltd, 2010.

[33] E A Smith and R J Senter. Automated Readability Index. *AMRL-TR. Aerospace Medical Research Laboratories (U.S.)*, pages 1–14, may 1967.

[34] Ian Sommerville. Software Documentation. In *Software Engineering*, page 19. Addison-Wesley, 6th edition edition, 2001.

[35] SourceD. Detecting licenses in code with Go and ML.

[36] SourceD. Go-license-detector, 2018.

[37] R. C. Tausworthe. Standard classification of software documentation. Technical Report NASA-CR-145932, JPL-TM-33-756, Jet Propulsion Lab., California Inst. of Tech., Pasadena, CA, United States, January 1976.

[38] C. Treude and M. P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE 2016)*, pages 392–403, Austin, Texas, USA, May 2016.

[39] G. Uddin and M. P. Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, July 2015.

[40] Ravikiran Vadlapudi and Rahul Katragadda. On Automated Evaluation of Readability of Summaries: Capturing Grammaticality, Focus, Structure and Coherence. In *Proceedings of the NAACL HLT 2010 Student Research Workshop*, pages 7–12, Los Angeles, California, USA, 2010. Association for Computational Linguistics.

[41] Philip van Oosten, Dries Tanghe, and Véronique Hoste. Towards an improved methodology for automated readability prediction. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedins of the seventh conference on international language resources and evaluation (LREC 2010)*, pages 775–782, La Valleta, Malta, 2010. European Language Resources Association (ELRA).

[42] Christopher Vendome, Mario Linares-Vasquez, Gabriele Bavota, Massimiliano Di Penta, and Daniel German. Machine Learning-Based Detection of Open Source License Exceptions. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 118–129, 2017.

[43] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175 – 198, 2015.