RESEARCH ARTICLE

A framework for creating knowledge graphs of scientific software metadata

Aidan Kelley[1] and **Daniel Garijo**[2]

[1]Washington University in St. Louis, St. Louis, USA

[2]Information Sciences Institute, University of Southern California, Los Angeles, USA

Keywords: FAIR, knowledge graphs, metadata, metadata extraction, scientific software

Citation: Kelley, A., & Garijo, D. (2021). A framework for creating knowledge graphs of scientific software metadata. *Quantitative Science Studies*, *2*(4), 1423–1446. https://doi.org/10.1162/qss_a_00167

DOI: https://doi.org/10.1162/qss_a_00167

Corresponding Authors:
Aidan Kelley
aidankelley@wustl.edu

Daniel Garijo
dgarijo@isi.edu

ABSTRACT

An increasing number of researchers rely on computational methods to generate or manipulate the results described in their scientific publications. Software created to this end—scientific software—is key to understanding, reproducing, and reusing existing work in many disciplines, ranging from Geosciences to Astronomy or Artificial Intelligence. However, scientific software is usually challenging to find, set up, and compare to similar software due to its disconnected documentation (dispersed in manuals, readme files, websites, and code comments) and the lack of structured metadata to describe it. As a result, researchers have to manually inspect existing tools to understand their differences and incorporate them into their work. This approach scales poorly with the number of publications and tools made available every year. In this paper we address these issues by introducing a framework for automatically extracting scientific software metadata from its documentation (in particular, their readme files); a methodology for structuring the extracted metadata in a Knowledge Graph (KG) of scientific software; and an exploitation framework for browsing and comparing the contents of the generated KG. We demonstrate our approach by creating a KG with metadata from over 10,000 scientific software entries from public code repositories.

1. INTRODUCTION

Computational methods have become crucial for making scientific discoveries in areas ranging from Astronomy (LIGO-VIRGO, n.d.) or High Energy Physics (Albrecht, Alves et al., 2019) to Geosciences (USGS, n.d.) and Biology (Prlić & Lapp, 2012). Software developed for this purpose is known as *scientific software*, and refers to the code, tools, frameworks, and scripts created in the context of a research project to process, analyze or generate a result in an academic publication. Examples of scientific software involve novel algorithm implementations, simulation models, data processing work flows, and data visualization scripts.

Scientific software is key for the reproducibility of scientific results as it helps others understand how a data product has been created or modified as part of a computational experiment or simulation and avoids replicating effort. Therefore, scientific software should adapt the principles for Finding, Accessing, Interoperating and Reusing scientific data (FAIR) (Wilkinson et al., 2016) to help scientists find, compare, understand, and reuse the software developed by other researchers (Lamprecht, Garcia et al., 2020).

Fortunately, the scientific community, academic publishers, and public stakeholders have started taking measures towards making scientific software a first-class citizen for academic research. For example, initiatives such as the Software Carpentry events (Software Carpentry, n.d.) or the Scientific Paper of the Future Initiative (n.d.) teach researchers about best practices for software documentation and description; community groups such as FAIR4RS are actively analyzing how to evolve the FAIR principles for Research Software (FAIR4RS, n.d.); institutions such as the Software Sustainability Institute (n.d.), OpenAIRE (n.d.) and the Software Heritage (n.d.) project help preserve and archive existing software; code repositories such as GitHub (n.d.) provide the means to store and version code; software registries such as ASCL (Shamir, Wallin et al., 2013) encourage scientists to describe software metadata; and container repositories such as DockerHub (n.d.) help capture the computational environment and dependencies needed for software execution. However, despite these efforts, two main challenges remain for efficiently and effectively finding, reusing, comparing, and understanding scientific software:

1. Software metadata is **heterogeneous, disconnected, and defined at different levels of detail**. When researchers share their code, they usually include human-readable instructions (e.g., in readme files) containing an explanation of its functionality, installations instructions, and how to execute it. However, researchers do not often follow common guidelines when preparing these instructions, structuring information in different sections and with usage assumptions that may require a close inspection for correct interpretation. This heterogeneity makes reusing and understanding existing scientific software a time-consuming manual process. In addition, support files (e.g., sample input files, extended documentation, Docker images, executable notebooks) are becoming increasingly important to capture the **context** of scientific software, but they are often disconnected from the main instructions, even when they are part of the same repository.
2. **Finding and comparing** scientific software is a manual process: According to Hucka and Graham (2018), the means followed by researchers to find and compare software are by doing a keyword search in code repositories; reading survey papers; or following recommendations from a colleague. The scientific community has developed general-purpose software metadata registries (CERN & OpenAIRE, 2013; FigShare, n.d.) to help reuse and credit scientists; and in some scientific communities, software metadata repositories have started collecting their own software descriptions to facilitate software comparison, credit and use (Gil, Ratnakar, & Garijo, 2015; Shamir et al., 2013). However, populating and curating these resources with metadata is, overall, a manual process.

In this paper we address these issues by proposing the following contributions:

- A SOftware Metadata Extraction Framework (SOMEF) designed to automatically capture 23 common scientific software metadata categories and export them in a structured manner (using JSON-LD (Champin, Longley, & Kellogg, 2020), RDF (Miller & Manola, 2004), and JSON representations). SOMEF extends our previous work (Mao, Garijo, & Fakhraei, 2019) (which recognized four metadata categories with supervised classification and seven metadata categories through the GitHub API) by expanding the training corpus (from 75 to 89 entries); by applying a wider variety of supervised classification pipelines; by introducing new techniques for detecting metadata categories (based on the structure used in the different sections of a readme file and regular expressions); and by detecting 12 new metadata categories and auxiliary files (e.g., notebooks, Dockerfiles) in a scientific software repository.
- A methodology for extracting, enriching, and linking scientific software metadata using SOMEF.

- A framework for browsing and comparing scientific software based on the results of the previous methodology.

We use Knowledge Graphs (KGs) (Bonatti, Decker et al., 2019) to represent scientific software metadata, as they have become the *de facto* method for representing, sharing, and using knowledge in AI applications. In our KG, nodes represent software entries linked to their associated metadata (creators, instructions, etc.) and their context (examples, notebooks, Docker files, etc.) through different edges. We illustrate our methodology and framework by automatically building a KG with over 10,000 scientific software entries from Zenodo.org (CERN & OpenAIRE, 2013) and GitHub.

The remainder of the paper is structured as follows: We first describe our framework for scientific software metadata extraction and how it structures metadata from readme files in Section 2. Next, in Section 3, we describe our methodology for extracting, enriching, and linking scientific software metadata in a KG, followed by our approach to exploit its contents by browsing and comparing different entries. We then discuss the limitations of our work in Section 4 and compare our approach against related efforts in the state of the art (Section 5) before concluding the paper in Section 6.

2. SOMEF: A SCIENTIFIC SOFTWARE METADATA EXTRACTION FRAMEWORK

An increasing number of researchers and developers follow best practices for software documentation (Guides, n.d.) and populate their repositories with readme files to ease the reusability of their code. Readme files are usually markdown documents that provide basic descriptions of the functionality of a software component, how to run it, and how to operate it. Therefore, in our work, we target readme files as the main source from which to extract metadata. In this section we first introduce the metadata fields we focus on and our rationale for extracting them, followed by the supervised and alternative methods we have developed to extract as many metadata fields as possible. We end this section by describing the export formats we support, extending existing software metadata representation vocabularies and community standards.

2.1. Common Scientific Software Metadata in Code Repositories

Despite existing guidelines (Guides, n.d.), readme files do not have a predefined structure, and scientific software authors usually structure them in creative ways to communicate their software instructions and setup. When we started our work, we had four main requirements for metadata categories to extract:

- **Description**: To *discover and understand* the main purpose of a software component.
- **Installation instructions**: How to set up and *use* a software component.
- **Execution instructions**: Which indicate how a software component can be *used* and how.
- **Citation**: To *attribute* the right credit to authors.

These categories can be easily expanded to gather more details to help findability (e.g., domain keywords), usability (e.g., requirements, license), support (e.g., how to contribute), and understanding (e.g., usage examples) of scientific software. In fact, related work has already categorized software metadata by interviewing domain scientists (Gil et al., 2015) and creating community surveys to identify ideal metadata that scientists would prefer to better find, understand, and reuse scientific software (Hucka & Graham, 2018). Using these efforts as a reference, we conducted an experiment to assess the common documentation practices followed by scientific software authors for their software: We built a corpus of repositories from different scientific disciplines, and we analyzed the structure of their readme files to find common metadata fields to extract.

The corpus consists of 89 Markdown readme files from GitHub repositories. GitHub is one of the largest code repositories to date (Gousios, Vasilescu et al., 2014), with a wide diversity in documentation maturity, software purpose, and programming languages. Our criteria for selecting repositories included repositories with high-quality documentation; popular repositories (measured by the number of stars, releases, contributors, and dependent projects); and repositories designed to support scientists in their research. Scientific software contributed the most to the selection of repositories, although we included other tools typically used by scientists to implement their applications (e.g., Web development tools such as React). We also used as reference the Awesome curated list of repositories for different scientific domains (Awesome, n.d.), and popular tools using GeoJSON, Open Climate Science, etc., which provided links to relevant scientific projects. To be as diverse as possible, repositories covered a wide variety of programming languages, ranging from C++ and Python to Cuda, with a predominance of Python and C (30% each).

To analyze the corpus, we manually inspected the headers of the sections of the readme files included on each repository, grouping them together by category and counting the number of occurrences. As a result, we grouped 898 section headers into 25 metadata categories derived from related work. Headers that were unrelated to any identified metadata category were dismissed. Figure 1 shows the results of the 15 most common categories we found. As expected, installation, usage, and citation are among the most common categories, followed by the software requirements needed to install a given software component, the license or copyright restrictions, where to find more documentation, and how to contribute or how to deal with software-related problems. Some of the categories have an overlap and in some cases it becomes challenging to correctly identify a metadata field. For instance, the description of repositories is often found in the `Introduction/Overview`, or in the `Getting started` categories. `Example` can include invocation commands, `Support` and `FAQs` often refer on how to address problems with code, etc.

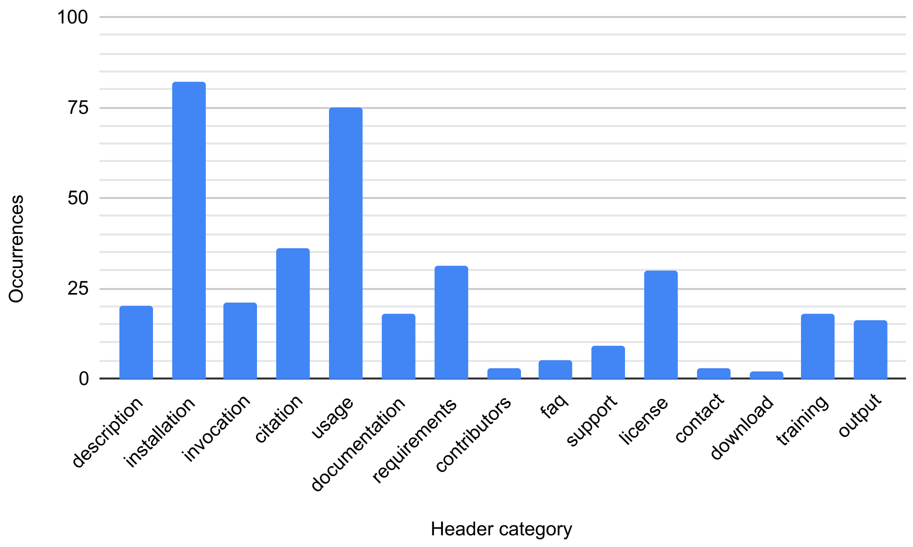


Figure 1. Distribution of the common header categories in our 89 readme files corpus.

Table 1. Number of ground truth excerpts and their mean length by metadata category

Category	# Excerpts	Mean length (words)
Description	336	27.95 ± 28.46
Installation	929	9.24 ± 11.39
Invocation	1,134	7.74 ± 9.88
Citation	316	8.20 ± 7.40

Using the results of our analysis, we expanded our initial software metadata list with the metadata categories listed below. We excluded metadata categories that did not appear in at least 10% of the corpus, (i.e., at least nine times):

- **Usage instructions, examples, and notes**: Assumptions and considerations recorded by the authors when executing a software component, or examples on how to use it.
- **Documentation**: Information on where to find additional documentation about a software component (besides the readme file).
- **Requirements**: Prerequisites and dependencies needed to execute a software component.
- **Support**: Guidelines and links of where to obtain support for a software component.
- **License**: License and usage terms of a software component.
- **Long name**: A longer version of the name of a software component, as the repository name is sometimes not enough for proper identification.

We decided not to include the categories related to `Training` and `Output` as they often refer to domain-specific scientific software (in the context of Machine Learning projects). We also considered the following categories, which are not present in Table 1 but are important auxiliary files that may be needed to set up or understand scientific software:

- **Digital Object Identifier (DOI)**: In some cases authors include a reference publication and a DOI (e.g., in Zenodo) for their code, which helps tracking snapshots of their work.
- **Dockerfiles**: Needed to create a computational environment for a scientific software component. Some code repositories include more than one.
- **Computational notebooks**, which often showcase how to execute the target software, how to prepare data for its usage, or how to operate with the produced results. More recently, links to live environments such as Binder (n.d.) are starting to appear as part of readme files as well, although they are not yet a common practice.

2.2. Supervised Methods for Software Metadata Classification

We extend our previous work (Mao et al., 2019) to train supervised binary classifiers to extract descriptions, installation instructions, citations, and invocation excerpts from readme files. The rationale for developing supervised classifiers for these categories was to attempt to extract them at a granular level, as their related excerpts can often be found scattered across different sections in readme files (e.g., invocation commands can sometimes be found in examples, installation, or usage sections).

2.2.1. *Training corpus*

We trained our classifiers using the 89 readme files from our preliminary section analysis (expanding the 75 readme files in the initial corpus from Mao et al. (2019)). All readmes

Table 2. Best classification results for each metadata category

Classifier	Best pipeline	Precision	Recall	F-Measure
Description	CountVectorizer + LogisticRegression	0.85	0.79	0.82
Installation	TFIDFVectorizer + StochasticGradientDescent	0.92	0.9	0.91
Invocation	CountVectorizer + NaiveBayes	0.88	0.9	0.89
Citation	CountVectorizer + NaiveBayes	0.89	0.98	0.93

consist of **plain text rendered Markdown** (i.e., without markup), divided in paragraph excerpts (separated by newline delimiters). We built the ground truth by manually inspecting the readmes and annotating them with the right category by hand. As a result, we ended up with the paragraph excerpts shown in Table 1.

To balance each corpus, we sampled negative examples for each category to obtain a 50% positive and 50% negative sample distribution. For each category, the negative class contained random samples from the other three categories (12.5% from each), plus control sentences from the Treebank corpus (Marcus, Kim et al., 1994) (up to 12.5%), to make the system more robust (i.e, to ensure that the classifiers do not devolve into a code vs. natural text classifier).

2.2.2. *Classification results*

We used the Scikit-learn framework (Pedregosa, Varoquaux et al., 2011) to train different binary supervised classifiers. Because the corpora are based on text, we first transformed each excerpt into a feature vector (using the CountVectorizerand TfidfVectorizer methods from Scikit learn library). We then applied available binary classifiers (namely StochasticGradientDescent with log as loss function, LogisticRegression, NaiveBayes, Perceptron, RandomForest, AdaBoost, XGB, DecisionTree and BernoulliBayes) and selected the pipelines with best results in average. All results are cross-validated using stratified fivefold cross-validation. The best results for each category can be seen in Table 2, and have an average above 0.85 precision. We prioritized pipelines that maximized precision and F-Measure to favor the extraction of correct results. That said, our approach works best with paragraphs containing multiple sentences (short paragraphs with one sentence may miss some of the context needed for the correct classification). All model files from our experiments, as well as the rankings from each vectorizer and classifier combination we tried, are available online with a Zenodo DOI (Mao, vmdiwanji et al., 2020).

We also considered removing stop words and using stemming algorithms in our excerpt feature extraction as they have proven to be useful in texts to prevent a feature matrix from becoming too sparse. However, the computer science domain includes very precise words (e.g., within invocation commands), and we did not see an improvement when incorporating these methods in our analysis pipelines. Hence, we discarded stemming and stop word removal from our final results.

2.3. Alternative Methods for Software Metadata Classification

While our supervised classification results show appropriate results for the Description, Installation, Invocation, and Citation categories, the remaining metadata categories do not appear in a consistent manner in the selected repositories, and finding representative corpora for training requires a significant effort. Therefore, we explored three main alternative methods for recognizing metadata categories, further described below.

2.3.1. Header analysis

Leveraging the results of our readme header analysis, we designed a method to annotate readme sections based on their headers. The intuition is that if a section is named after one of the main categories identified (e.g., "Description" or "About"), then the body of that section will contain the right metadata (e.g., description) of the target software. Authors use very different names for their sections, but following our initial analysis we learned how different keywords can be grouped together using synonyms. For each metadata category, we looked at the most common keywords and retrieved their Wordnet sinsets (Miller, 1995), paying special care to select those with the correct meaning (e.g., "manual" may refer to something that requires manual effort, or an installation manual). We then created a method to automatically tag each header of a readme file with the closest category in meaning (if any), annotating the respective section text as its value. To evaluate our results, we created a new corpus labeling each of the 898 headers present in the 89 readme files.

This approach is less granular than supervised classification (multiple paragraphs may be annotated under a single category), and weak against typos in headers, but yields surprisingly good results for some of the target metadata categories. Table 3 includes an overview of the F-Measure results of the extracted headers for the repositories in our corpus. Metadata categories such as License, Requirements, Invocation, and Documentation have very high F-Measure, indicating an agreement from the community when describing them in software documentation. Citation and Installation have a high F-Measure, although not as good as the supervised classification results. The Description and Usage categories behave slightly worse, which indicates ambiguous usage by authors in their documentation (this is also the case of the Support category, which yields the worst results). Upon further inspection, we also discovered that a small number of the errors are not caused by ambiguity problems, but rather by formatting errors in the markdown files. Appendix B includes a full table with the precision and recall metrics used to calculate the F-Measures of Table 3.

2.3.2. Regular expressions

Some metadata categories can be recognized using regular expressions in the readme files. Some examples are when authors include citations following the BibTeX syntax (used to manage references in LaTeX) or when authors include badges that display as clickable icons, such as the ones for computational notebooks, Zenodo DOIs, package repositories, etc. Figure 2 shows an example for the Pandas code repository, where many badges are displayed (including one to the Zenodo DOI). We currently support regular expressions for extracting BibTeX citations as well as Zenodo DOIs and Binder executable notebooks (Binder, n.d.) from badges.

2.3.3. File exploration

We downloaded a snapshot of the code of each analyzed repository and searched for the following files:

- **License**: The best practices for code repositories in GitHub include adding a license file (LICENSE.md) stating which type of license is supported by the code.
- **Dockerfile**: Files that include a set of instructions to create a virtual environment using Docker. These files are becoming popular to facilitate reproducibility, and they are easily recognizable by their name (Dockerfile).
- **Executable notebooks**: We support the recognizing of Jupyter notebooks (with format . ipynb), whichare usually added as part of Python projects to showcase the functionality of the software component.

Table 3. Summary of the different categories supported by SOMEF and their main extraction techniques. Supervised classification techniques operate in a paragraph-based basis, while header analysis reports results by sections. Support for detecting a metadata field with regular expressions, file exploration and GitHub API is indicated with an "X"

	Extraction method				
Category	**Supervised classification (F-Measure)**	**Header analysis (F-Measure)**	**Regular expression**	**File exploration**	**GitHubAPI**
Description	0.82	0.68			X (short)
Installation	0.91	0.85			
Invocation	0.89	0.91			
Citation	0.93	0.87	X (bibtex)		
Usage		0.68			
Documentation		0.95			X (readme)
Requirements		0.93			
Support		0.52			
License		1		X	X
Name					X
Long Name			X		
DOI			X		
Dockerfile				X	
Notebooks			X	X	
Owner					X
Keywords					X
Source code					X
Releases					X
Changelog					X
Issue tracker					X
Programming languages					X
Download URL					X
Stars					X

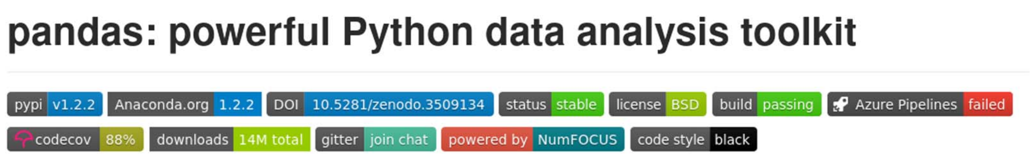


Figure 2. Badges displayed in the Pandas code repository.

Because multiple notebooks or Dockerfiles may exist in one software repository, we annotate all of them when exploring a repository.

2.3.4. GitHub API

GitHub provides an API with basic repository metadata filled by the authors, and we exploit it to obtain additional metadata. We extract the following categories:

- **Name**: Short name of the repository (typically the id of the target repository).
- **Owner**: Person or organization in charge of the repository.
- **Keywords**: Author-selected keywords to improve findability of their software.
- **Source code**: URL of the source code repository.
- **Releases**: Links to the different snapshot versions of the software component.
- **Download URL**: URL where to download the target software associated with a release (typically the installer, package, or a tar ball to a stable version).
- **Changelog**: Description provided by authors for each release, typically listing the main novelties and issues addressed for a given release.
- **Issue tracker**: Link to the issue list of the target repository.
- **Programming languages**: Main programming languages used in a repository. If auxiliary files are included (e.g., notebooks, setup scripts, etc.), this will return all the available languages and their distribution.
- **Stars**: Number of stars assigned by users. Note that this feature is time-dependent, as users may star or un-star a repository.

While some of these metadata categories were not identified as critical by related work or our main category analysis (e.g., owner, stars), we consider them useful metadata that can help in understanding how a software component has evolved or how it is supported by the community. Hence, they are included in the metadata extraction reports.

Table 3 shows a summary of all the metadata categories we support, along with the methods that can be used to extract them. We note that some of the categories may be extracted by more than one method or be tagged in more than one category (e.g., requirements and installation instructions), leaving to users the choice of selecting the preferred one.

2.4. Exporting Scientific Software Metadata

To ease the reusability of our results, we support exporting our extracted metadata in three main serializations with different levels of detail. As the supervised classification methods print out a confidence in their classification, we have set up the ability to set a threshold (which by default is 0.8) to filter out nonsignificant results. Results from header analysis, regular expressions, and the GitHubAPI are assigned the highest confidence.

2.4.1. Codemeta export

Codemeta (Jones, Boettiger et al., 2017) is a JSON-LD (Champin et al., 2020) vocabulary which extends the Schema.org (Guha, Brickley, & Macbeth, 2016) *de facto* standard to provide basic markup of scientific software metadata. Codemeta is lightweight and is gaining popularity and support among code registries as it provides a cross-walk between different vocabulary terms for scientific software metadata. However, Codemeta does not support some of the metadata terms we extract (e.g., invocation command, notebooks, Dockerfiles), which are thus not included in the export. The methods used for extracting each metadata category (e.g., classifiers, GitHubAPI) and their confidence are also not included.

2.4.2. RDF export

We have aligned all extracted metadata categories with the Software Description Ontology (Garijo, Osorio et al., 2019), an ontology that extends Codemeta and Schema.org to represent software metadata, and provides the ability to serialize the results in W3C Turtle format (Carothers & Prud'hommeaux, 2014). However, to avoid complicating the output, this export does not serialize the method used on each extraction or its confidence.

2.4.3. JSON export

We provide a JSON representation that indicates, for each extracted metadata category, the technique used for its extraction and its confidence, in addition to the detected excerpt. The JSON snippet below shows an example for the Description category of a Python library. This way, the provenance associated with each extraction is recorded as part of the result.

```
"description": [

        {

        "excerpt": "KGTK is aPython library …",

        "confidence": [0.8294290479925978],

        "technique": "Supervised classification"

        }

]
```

3. TOWARDS KNOWLEDGE GRAPHS OF SCIENTIFIC SOFTWARE METADATA

In this section we describe how, using SOMEF, we create, populate and exploit KGs of scientific software metadata.

3.1. Knowledge Graph Creation Methodology

Figure 3 shows the main steps of our methodology for enriching and linking scientific software metadata by integrating a code repository and a software metadata registry. Arrows represent the dataflow, while numbers represent step execution order. First, we scrape a list of software entries from a target software registry (e.g., Zenodo). Then, for each entry, we retrieve its version data, extract all code repository links, if present, and download the full text of its readme file. The readme file is parsed by SOMEF, and the results are combined and then aggregated into a KG. Finally, we enrich the resultant software entries by extracting keywords and generate a second KG which is combined with the first. An assumption of our methodology is the existence of the link between the software metadata registry and the code repository where the readme files reside.

3.2. Representing Scientific Software Metadata

Figure 4 shows a snapshot of the main classes and properties we used to represent software metadata in our KG. We use a simple data model that reuses concepts from the Software

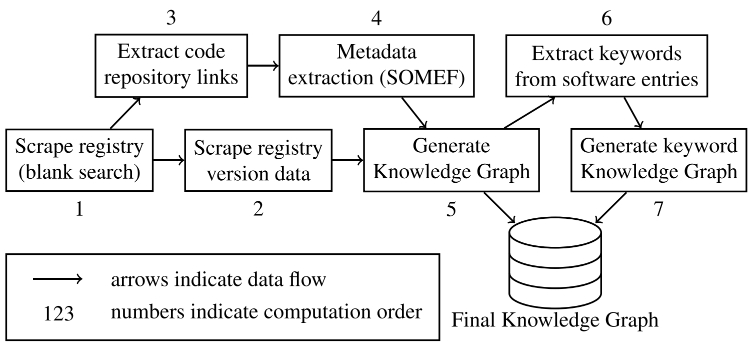

3
Extract code repository links
4
Metadata extraction (SOMEF)
6
Extract keywords from software entries
Scrape registry (blank search)
1
Scrape registry version data
2
Generate Knowledge Graph
5
Generate keyword Knowledge Graph
7
arrows indicate data flow
123 numbers indicate computation order
Final Knowledge Graph

Figure 3. Scientific software Knowledge Graph creation methodology.

Description Ontology (Garijo et al., 2019) to represent software, software versions, and their authors, as indicated in the figure. We then used an N-ary relationship pattern (Rector & Noy, 2006) to qualify found keywords with additional metadata (e.g., whether they are title keywords or description keywords), which we use for search purposes.

3.3. SOSEN: A Knowledge Graph of Scientific Software Metadata

To demonstrate our approach, we built SOSEN, a KG integrating Zenodo.org, an open source metadata registry with thousands of scientific software descriptions, and GitHub as the main code repository from which readme files are parsed. We use Zenodo because it specifically stores scientific software and has a simple, open API; GitHubstores much of the code available in Zenodo and has an open API as well. Other open repositories were considered but discarded for this version of SOSEN due to their broad scope beyond scientific software (e.g., Software Heritage, n.d.); or lack of explicit link to a code repository (e.g., FigShare, n.d.).

Figure 5 shows a high-level overview of the architecture used to implement our methodology. First, we obtained a list of software entries from Zenodo, an open-access repository of scientific documents. To obtain the software entries (called *records*), we performed a blank search, filtering by software. This returned a list of the 10,000 most recent records. The choice of 10,000 is a limitation imposed by Zenodo, and pagination cannot be used to circumvent this limit. To obtain a larger set of entries, we performed the same search again, with order

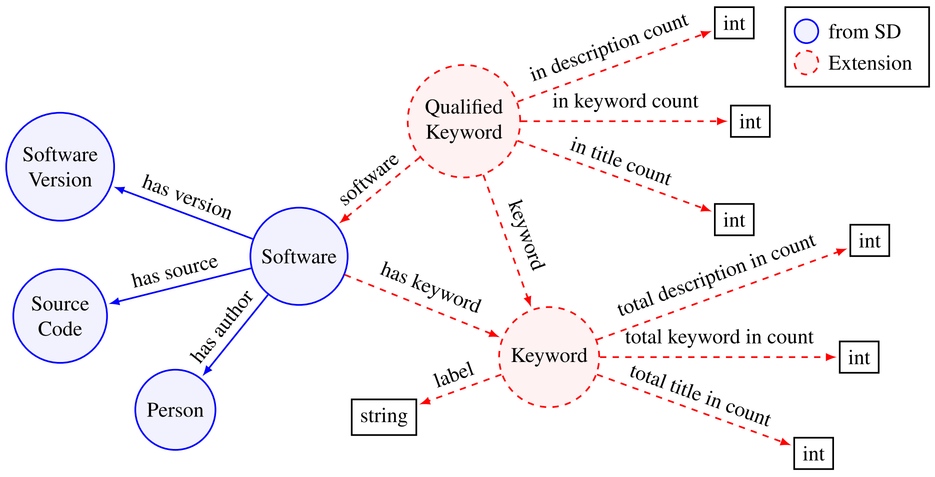


Figure 4. An overview of the data model used in the SOSEN KG.

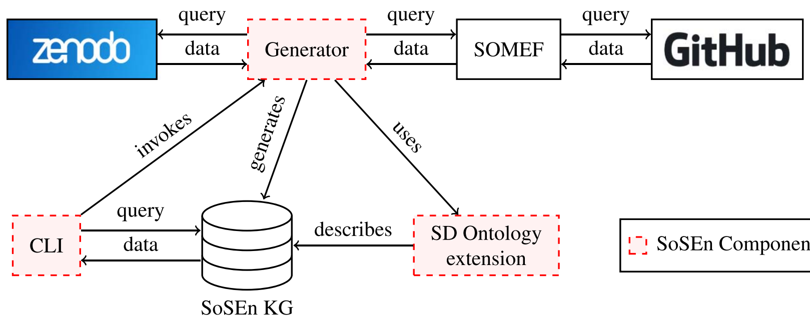


Figure 5. Architecture for generating the SOSEN KG.

reversed, which yielded another 10,000 records. This meant that the 20,000 software entries retrieved were almost half of the total software entries in Zenodo (at the time of writing), which we considered sufficient for demonstrating our methodology.

We then enriched all software entries using SOMEF with the RDF export, enabling supervised classification, header analysis, regular expressions, file exploration and the GitHub API for extracting software metadata categories. We filtered out entries that did not have an associated GitHub link and used SOMEF with the latest commit of each repository. As a result, we extracted metadata categories from 69% of the candidate software records (nearly 13,800).

Next, we automatically extracted keywords from the description and title of each software entry. This was achieved by splitting the title or description into words and removing stop words. Finally, we computed the properties needed to support the representation of TF-IDF scores to retrieve entities efficiently.

As for the structure of the KG itself, we chose a permanent Uniform Resource Identifier (URL) scheme, with the prefix `https://w3id.org/okn/i`. Instances of the Software class have the same name as their corresponding GitHub repositories, which are unique. Table 4 displays examples of other entity URIs using an example from the SOSEN KG.

3.4. SOSEN: Knowledge Graph Assessment

Figure 6 shows a subgraph of one of the entries of the SOSEN KG, highlighting how the information is combined from Zenodo, SOMEF (retrieved from GitHubreadmes), and the keyword enrichment analysis performed as part of our methodology.

Table 5 shows the total number of entities in the SOSEN KG, while Figure 7 shows statistics on the completeness of the main software metadata categories. Metadata categories displayed

Table 4. Example URIs for different classes of entities in the graph. The _ prefix stands for https://w3id.org/okn/i, and is common for all entities

Entity	Example URI
Software	_:Software/dgarijo/Widoco
SoftwareSource	_:SoftwareSource/dgarijo/Widoco
SoftwareVersion	_:SoftwareVersion/dgarijo/Widoco/v1.4.14
Keyword	_:Keyword/ontology
QualifiedKeywordRelationship	_:Software/dgarijo/Widoco/QualifiedKeyword/ontology

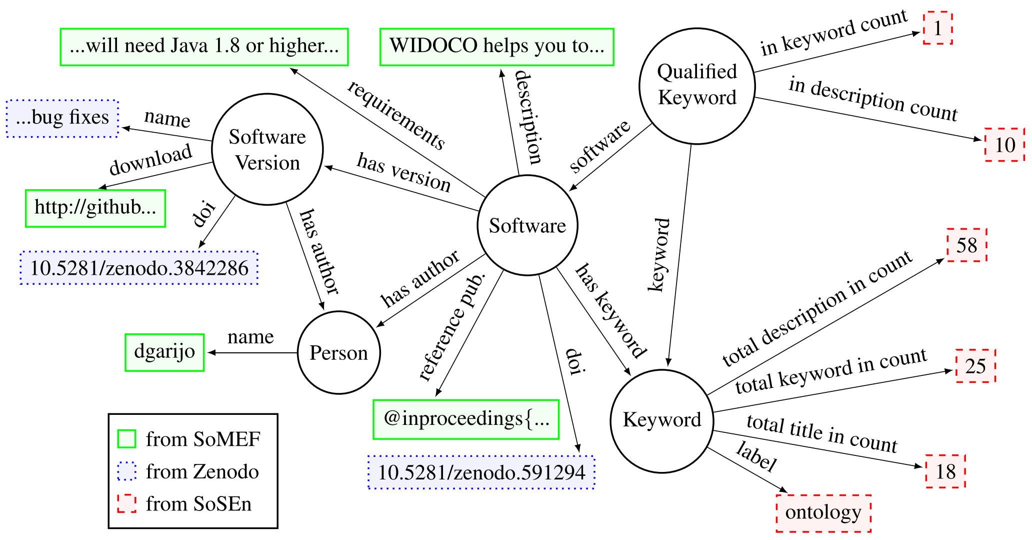


Figure 6. A subgraph of a software entry in the SOSEN KG, showing the different information sources.

to the right of "license" in Figure 7 (with the exception of "doi") come only from the GitHub API, and therefore some entries are incomplete because the authors who created them did not add enough information. Those categories to the left of "license" were extracted using classifiers, regular expressions, header analysis, or the GitHub API (e.g., documentation is complemented by pointing to the source readme file, hence the high completion rate in the KG), and are prompt to precision and recall errors. Some metadata categories, such as the detection of auxiliary files (Docker, Notebooks) were not yet supported at the time of the creation of the SOSEN KG and therefore are not included in the figure. Categories that are shared by all repositories by default (i.e., source code, issue tracker, programming language, owner) are not included in the figure for simplicity.

Notably, less than a quarter of the software entities have user-defined keywords. This hinders their findability, although a search based on the keywords in the titles of these repositories would likely reach them (most software entities have titles).

Further, we see that more than half of the entries have both a license and a version. This statistic is important, as having a license is necessary to reuse the code, and a software entity having multiple versions suggests that it has been maintained, which may be an indicator of its quality (almost half of the entries have multiple versions with independent releases of code).

Table 5. The number of entities of a given class in the SOSEN KG

Entity class	Count
Software	13,763
SourceCode	13,763
SoftwareVersion	50,795
Keyword	88,304
Person	11,858
Total triples	3,927,004

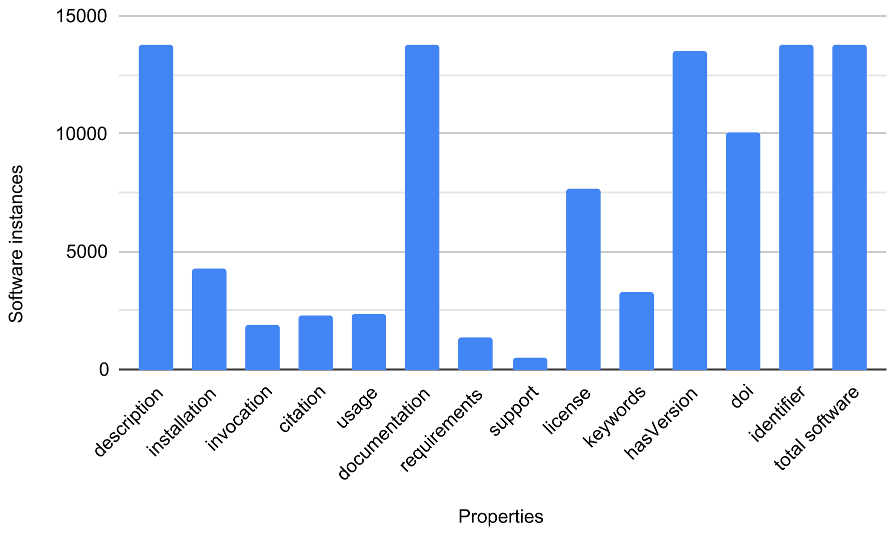


Figure 7. Coverage of the main properties used in the SOSEN KG. The total number of software instances is included for reference.

We see that the categories detected without the GitHub API are relatively sparse. This may be due to user omission (i.e., authors not adding sufficient detail to their readme), SOMEF error (i.e., the classifiers missing a metadata category), or the property being mentioned in documentation external to the readme. A lack of these categories means that the user will, in many cases, have to revert to manually browsing the code repository for relevant information.

Figure 8 shows the distribution of the top 15 programming languages in the SOSEN KG (out of 267), with Python as the most commonly used. Note that a repository may contain files in more than one programming language (e.g., Python, Jupyter notebooks, and shell scripts to help starting up the project), and hence the number of times programming languages appear may be higher than the number of software instances in the KG.

All of these statistics were generated using SPARQL queries against the SOSEN KG, which can be accessed under a Zenodo DOI (Kelley & Garijo, 2020).

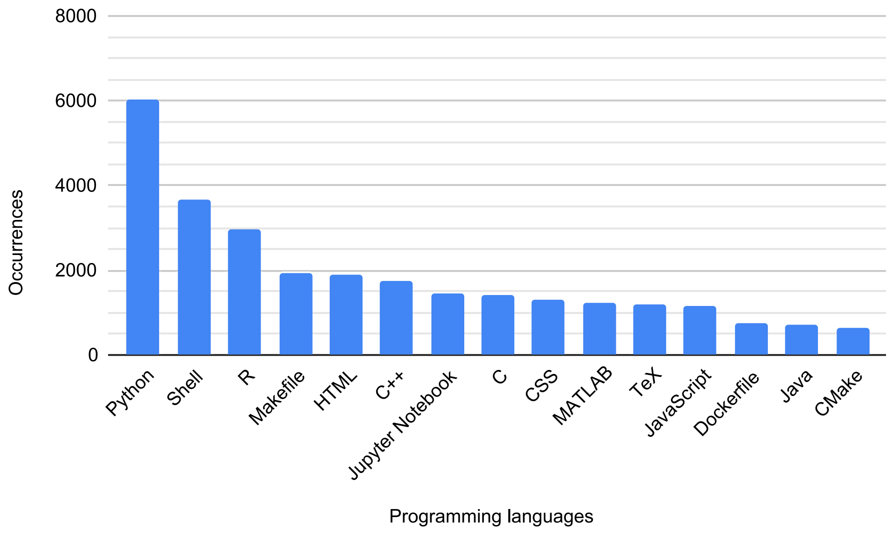
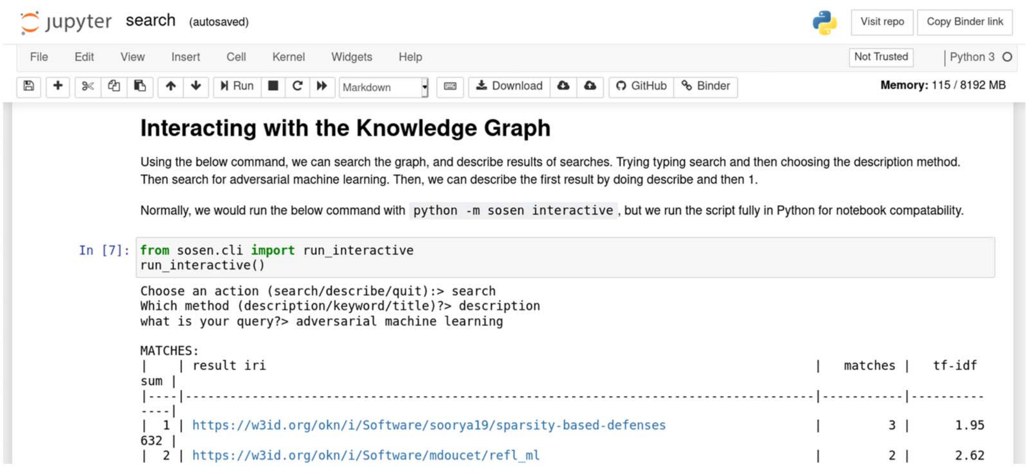


Figure 8. Distribution of the top 15 programming languages in the SOSEN KG.



Interacting with the Knowledge Graph

Using the below command, we can search the graph, and describe results of searches. Trying typing search and then choosing the description method. Then search for adversarial machine learning. Then, we can describe the first result by doing describe and then 1.

Normally, we would run the below command with `python -m sosen interactive`, but we run the script fully in Python for notebook compatability.

```
In [7]: from sosen.cli import run_interactive
        run_interactive()

        Choose an action (search/describe/quit):> search
        Which method (description/keyword/title)?> description
        what is your query?> adversarial machine learning

        MATCHES:
        |    | result iri                                                                          |   matches |   tf-idf
        sum |
        |----|-------------------------------------------------------------------------------------|-----------|----------
        ----|
        |  1 | https://w3id.org/okn/i/Software/soorya19/sparsity-based-defenses                    |         3 |      1.95
        632 |
        |  2 | https://w3id.org/okn/i/Software/mdoucet/refl_ml                                     |         2 |      2.62
```

Figure 9. Snapshot of the Jupyter notebook we developed to search and compare software metadata entries in the SOSEN KG.

3.5. SOSEN CLI: A Framework for Using the SOSEN KG

We created a command line interface (CLI) Python framework (Kelley & Garijo, 2021) to ease search and comparison of scientific software in the SOSEN KG. The framework can be used through Jupyter Notebooks as shown in Figure 9. First, we implemented a TF-IDF based keyword search using SPARQL (see Appendix A). This functionality is exposed through the `search` method of the SOSEN CLI. Users enter a query, which is broken into keywords, splitting at the space character. Then, users can choose between three methods for keyword search: user-defined, title, or description keywords. An example result for the search "knowledge graph construction" is shown in Table 6.

We also implemented a method to describe and compare software. The search method returns result URIs, which can be passed into the `describe` method to give a short summary of the target software. If multiple URIs are passed to the describe method, they are compared side by side. The results are sorted so that, for a given metadata category, values that are in common show up first. An example can be seen in Table 7, where we describe the top two results for the search "knowledge graph construction" from Table 6. We are able to compare relevant information between the two software packages and see that both use similar languages and have open source licenses. The metadata shown for the side-by-side comparison uses as reference some of the most demanded fields (Hucka & Graham, 2018) by scientists when searching software. However, the number of metadata categories has been reduced on purpose to avoid overwhelming users.

Table 6. The result of searching "knowledge graph construction" using the description keyword method. The search has been limited to the first five results

	Result URI	Matches	TF-IDF sum
1	https://w3id.org/okn/i/Software/SDM-TIB/SDM-RDFizer	3	2.29
2	https://w3id.org/okn/i/Software/usc-isi-i2/kgtk	2	3.38
3	https://w3id.org/okn/i/Software/SystemsGenetics/KINC	2	2.81
4	https://w3id.org/okn/i/Software/TBFY/knowledge-graph	2	1.69
5	https://w3id.org/okn/i/Software/pykeen/pykeen	2	1.45

Table 7. Example subset of the comparison functionality. These are the top two results for the search "knowledge graph construction," using the description keywords method. By default, the SOSEN CLI will output the table to the terminal, but it can be configured to output LaTeX markup (as shown in the example)

Name	**SDM-RDFizer**	**kgtk**
author	SDM-TIB	usc-isi-i2
description	An Efficient RML-Compliant Engine for Knowledge Graph Construction	Knowledge Graph Toolkit
languages	Dockerfile	Dockerfile
" "	Python	Python
" "		Makefile
" "		Shell
license	https://api.github.com/licenses/apache-2.0	https://api.github.com/licenses/mit

4. DISCUSSION

Our work aims to address important challenges for software findability, comparison, and understanding that are performed mostly in a manual manner today. In this section we discuss some of the assumptions and limitations of our approach, which may inform new research challenges and lines of future work.

4.1. Software Metadata Availability

While readme files are highly informative for setting up and describing software, they may contain typos, be incomplete, or be nonexistent. Using other sources for documentation, such as manuals, reports, and publications, may help retrieving additional insight into how to use a particular scientific software component. For example, publications may contain additional insight on the assumptions and restrictions of a software component. Repositories sometimes contain input and output samples that may help understand how to prepare and transform data for a particular software component, or how to combine it with other software. In this work we have started capturing auxiliary files of scientific software, but additional work is needed to describe all these *ad hoc* resources in the right context.

During our analysis, we have prioritized extracting precise descriptions of software metadata fields by different methods (supervised classification, header analysis, regular expressions, file exploration or the GitHub API). Some of these methods may extract the same fields, leading to similar, redundant statements. A postprocessing step would help curating redundancies in the graph.

At the same time, the work proposed here may be used to inform users on how well their repositories are described, enforcing better practices on software description for authors to help dissemination and findability of their software.

4.2. Updating and Extending the SOSEN KG

SOSEN was designed with extensibility in mind. We believe that many of the design choices of the project (such as using KGs) make SOSEN extensible, both for adding new data from existing sources and for incorporating new data sources, as we detail below.

We are continually evolving SOMEF, and as a result, the SOSEN KG may need to be updated with new data from existing sources. We have a pipeline for recreating the KG, and plan to update the SOSEN KG after each major SOMEF release. This is a process that occurs in bulk, and it is not designed to be incremental at the moment.

Updating the SOSEN KG with data from other registries is relatively easy, but needs additional work to find the right correspondence between entries in different catalogs. This has been left out of the scope of this publication. Therefore, at the moment, our methodology has an assumption of having an explicit link between the target metadata registry and the code repository to integrate.

4.3. Finding Scientific Software

The SOSEN framework makes good first steps towards improving scientific software findability. As shown in Figure 7, we are able to retrieve a significant number of keywords from the descriptions extracted by SOMEF, integrating them together in an enriched KG. In addition, we extract metadata categories that may inform the search (e.g., enabling specifying a license, programming language or software requirements). Current limitations of our approach include that our search algorithm uses exact keyword matching, which behaves poorly to spelling errors and ambiguities; and that the KG entities are not dereferenceable (i.e., KG entities do not resolve in a browser). Using scientific software text embeddings and fuzzy search (such as that supported by text search engines) are promising solutions to address the first limitation. Using a LinkedData Frontend may address the second limitation.

The SOSEN KG is not large in size, and therefore many scientific software packages are currently missing. However, the scope of this work is to demonstrate our methodology with a working KG of enriched entries from readme files.

4.4. Software Understanding and Comparison

Our work for automated metadata extraction and comparison extracts categories that are usually hard to find in other metadata registries without manual curation. The SOSEN CLI exposes this information easily, without requiring users to be SPARQL experts to exploit the contents of the SOSEN KG. The metadata fields exposed in the SOSEN CLI have not directly been validated with a user evaluation, but they are a subset of the metadata categories identified by community surveys with more than 60 answers from scientists of different disciplines (Hucka & Graham, 2018). In addition, two software packages can be put side by side, allowing users to assess the limitations of each and make an informed decision. Further work is needed to explore other meaningful ways to compare software, for example, by exploring their code, calculating analytics (e.g., how well documented or maintained the code is), coverage of tests, code comments, or exploring support files (notebooks, Dockerfiles, etc.).

5. RELATED WORK

5.1. Scientific Software Metadata Extraction from Text and Code

While an extensive amount of work exists to extract entities and events from text, few approaches have paid attention to scientific software documentation. The Artificial Intelligence Knowledge Graph (AIKG) (Dessì, Osborne et al., 2020) and the Open Research Knowledge Graph (ORKG) (Jaradeh, Oelen et al., 2019) both leverage deep learning techniques to extract mentions to methods and, in some cases, tools used in scientific publications. However, their focus is on research papers, and hence they do not handle external code repositories or readme files, which are the focus of our work. The OpenAIRE Research Graph (Manghi, 2020) is an ongoing effort to create a KG of open science artefacts, including scientific software. However, OpenAIRE focuses on the integration of public repositories at scale (e.g., by linking duplicate entities); while our approach extracts software-specific metadata.

Other areas of related work perform static code analysis (Ilyas & Elkhalifa, 2016) for different purposes, ranging from code quality to cybersecurity. Among these efforts, some techniques can be used to extract metadata. For example, libraries such as PyCG (Salis, Sotiropoulos et al., 2021) or pydeps (n.d.) can be used to extract the requirements and dependencies in a software project. These techniques are usually oriented towards a single programming language, but may complement the metadata extraction categories we perform with our work.

Other approaches mine code repositories and popular web forums such as Stack Overflow to create KGs for question answering (Abdelaziz, Dolby et al., 2020), retrieve code similar to a given function (Mover, Sankaranarayanan et al., 2018), autocomplete code snippets (Luan, Yang, et al., 2019), or help finding software to perform a particular functionality described in natural language (CodeSearchNet) (Husain, Wu et al., 2019). The scope of these approaches is different from ours, which is focused on automatically describing and linking software metadata. However, these initiatives define useful directions to expand and combine with our work (e.g., finding similar software).

Perhaps the approach that most resembles our work (besides our initial work in Mao et al. (2019), where we introduced an initial version of our framework) is AIMMX (Tsay, Braz et al., 2020), a recent AI model metadata extractor from code repositories that captures their data dependencies, machine learning framework (e.g., TensorFlow), and references. AIMMX also labels the main purpose of a machine learning code repository (medical domain, video learning, etc.). Instead, SOMEF extracts up to 23 metadata fields that range from software setup and auxiliary files to how to obtain support from the community, and can be applied to any type of scientific software.

5.2. Scientific Software Code Repositories and Metadata Registries

Code repositories such as GitHub (n.d.), GitLab (n.d.) and BitBucket (n.d.) are perhaps the most widely used by the scientific community to store, test, integrate, and disseminate scientific software code. However, these repositories do not hold much machine-readable software metadata besides license, programming language, creator, and keywords. Similarly, when releasing code, scientists may use platforms such as FigShare (n.d.) and Zenodo (CERN & OpenAIRE, 2013), as they provide DOIs stating how to cite particular code; code archival services such as Software Heritage (n.d.) or package repositories such as Pypi (n.d.) and Maven Central (Maven Central Repository Search, n.d.), which focus on disseminating an executable version of the code. In all these cases, metadata is often optional, and must be added manually by researchers.

Software metadata registries provide metadata descriptions of scientific software, complementing code repositories, and are usually curated by hand by domain experts. For example, the Community Surface Dynamics Modeling System (CSDMS) (Peckham, Hutton, & Norris, 2013) contains hundreds of codes for models for Earth surface processes; the Astrophysics Source Code Library (ASCL) contains unambiguous code descriptions in Astrophysics (Shamir et al., 2013); and OntoSoft (Gil et al., 2015), describes scientific software for Geosciences. These registries usually contain high-quality software metadata entries, but curating them by hand requires significant expertise. Our techniques may be used to automatically fill in entries, easing the work from curators and users.

Finally, Wikidata (Vrandečić & Krötzsch, 2014), a general-purpose KG which contains part of the information in Wikipedia in machine-readable manner, also stores high-level software metadata descriptions. Wikidata relies on manual curation as well, but has a strong, lively community of contributors and editors, making it an ideal candidate to integrate with our work and link to external entities (researchers, licenses, frameworks, etc.).

5.3. Scientific Software Metadata Comparison

Creating surveys to review existing work is a time-consuming task. For this reason, researchers have started leveraging KGs to create comparisons of related work. For example, the Open Research Knowledge Graph (Jaradeh et al., 2019) uses the content extracted from scientific publications to create interactive surveys to compare existing publications,but does not support scientific software.

Other platforms, such as OpenML (Vanschoren, van Rijn et al., 2013) and Papers with code (n.d.) take a more practical approach, providing comparison benchmarks on how well different machine learning methods perform for a particular task. This comparison excludes most software metadata, but is very informative to showcase the efficiency of a given method for a given task.

Finally, software registries such as our previous work in OntoSoft (Gil et al., 2015) and OKG-Soft (Garijo et al., 2019) provide the means to compare different scientific software entries using a UI. In contrast, the presented work takes a lightweight approach which does not require a UI to access and query the KG, making it easier to maintain (but becoming less visually attractive for users).

6. CONCLUSIONS AND FUTURE WORK

Given the volume of publications made available every year, it is becoming increasingly important to understand and reuse existing scientific software. Scientific software should become a first-class citizen in scholarly research, and the scientific community is starting to recognize its value (Smith, Katz, & Niemeyer, 2016). In this work we have introduced SOMEF, a framework for automating scientific software metadata extraction that is capable of extracting up to 23 software metadata categories; and a methodology to convert its results into connected KGs of scientific software metadata. We have demonstrated our methodology by building the SOSEN KG, a KG with over 10,000 enriched entries from Zenodo and GitHub; and a framework to help the exploration and comparison of these entries. Both SOMEF and SOSEN are actively maintained open source software, and available under an open license (Kelley & Garijo, 2021; Mao et al., 2020).

Our work uncovers exciting lines of future work. First, we are working towards addressing the current limitations of our software metadata extraction framework (e.g., by removing redundant extractions, improving robustness to typos in headers, and augmenting the training corpus). Second, we are exploring new metadata categories to facilitate software reuse and understanding, such as software package dependencies (different from the installation requirements); named entities that may be used to qualify relationships (e.g., installation instructions in Unix); and improving the capture of the functionality of a software component.

Third, we aim to improve the annotation of auxiliary files, not only recognizing them but also qualifying their relationship with the software component being described. For instance, identifying whether a notebook is an example, a preparation step, or needed for setting up a software component; or extracting additional software details from the reference publication.

To package all these files together, we plan to leverage the RO-Crate specification (Sefton, Ó Carragáin et al., 2021; Ó Carragáin, Goble et al., 2019), capturing the context in which all these files are used together when incorporating them into the SOSEN KG.

Finally, we plan to expand the SOSEN KG with additional data sources (e.g., by including all Zenodo software entries and FigShare software entries with readme files); and integrating our KG with public KGs such as Wikidata (Vrandečić & Krötzsch, 2014), which have a strong community of users that can help curate and refine the software metadata extraction errors. In particular, Wikidata contains a vast collection of scholarly articles, which we plan to explore to align to those entries in SOSEN KG with reference publications.

ACKNOWLEDGMENTS

We would like to thank Yolanda Gil, Allen Mao, Shobeir Fakhraei, HaripriyaDharmala, Jiaying Wang, Rosna Thomas, Thomas Eblen, Vedant Diwanji and Joseph Lee for their contributions.

AUTHOR CONTRIBUTIONS

Aidan Kelley: Investigation, Software, Writing—original draft, Writing—review & editing. Daniel Garijo: Investigation, Software, Supervision, Writing—original draft, Writing—review & editing.

COMPETING INTERESTS

The authors have no competing interests.

FUNDING INFORMATION

This work was performed as part of the Research Experience for Undergraduates (REU) program, supported by National Science Foundation (NSF) grant #1659886. This work was also funded by the Defense Advanced Research Projects Agency with award W911NF-18-1-0027, the National Science Foundation with award ICER-1440323 and the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with Universidad Politécnica de Madrid in the line Support for R&D projects for Beatriz Galindo researchers, in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

DATA AVAILABILITY

Datasets related to this article can be found at https://doi.org/10.6084/m9.figshare.14916684.v1: The SOSEN-KG (Turtle format), hosted at FigShare. https://doi.org/10.5281/zenodo.4574207: SOMEF 0.4.0 software and training corpus, hosted at Zenodo. https://zenodo.org/record/4574224: SOSEN-KG repository and examples, hosted at Zenodo.

REFERENCES

Abdelaziz, I., Dolby, J., McCusker, J. P., & Srinivas, K. (2020). Graph4code: A machine interpretable knowledge graph for code. *arXiv preprint*, arXiv:2002.09440.

Albrecht, J., Alves, A. A. Jr., Amadio, G., Andronico, G., Anh-Ky, N., ... Yazgan, E. (2019). A roadmap for HEP software and computing R&D for the 2020s. *Computing and Software for Big Science*, *3*(1), 7. https://doi.org/10.1007/s41781-018-0018-8

Awesome. (n.d.). https://awesome.re/ (accessed February 25, 2021).

Binder. (n.d.). https://mybinder.org/ (accessed February 27, 2021).

BitBucket (n.d.). https://bitbucket.org/ (accessed February 25, 2021).

Bonatti, P. A., Decker, S., Polleres, A., & Presutti, V. (2019). Knowledge graphs: New directions for knowledge representation on the semantic web (Dagstuhl Seminar 18371). In *Dagstuhl reports* (*Vol. 8*).

Carothers, G., & Prud'hommeaux, E. (2014). *RDF 1.1 turtle* (W3C Recommendation). W3C. https://www.w3.org/TR/2014/REC-turtle-20140225/

CERN & OpenAIRE. (2013). *Zenodo*. CERN. Retrieved from https://www.zenodo.org/. https://doi.org/10.25495/7gxk-rd71

Champin, P.-A., Longley, D., & Kellogg, G. (2020). *JSON-ld 1.1* (W3C Recommendation). W3C. https://www.w3.org/TR/2020/REC-json-ld11-20200716/

Dessì, D., Osborne, F., Recupero, D. R., Buscaldi, D., Motta, E., & Sack, H. (2020). AI-KG: An automatically generated knowledge graph of artificial intelligence. In *International Semantic Web Conference* (pp. 127–143). https://doi.org/10.1007/978-3-030-62466-8_9

DockerHub. (n.d.). https://hub.docker.com/ (accessed February 25, 2021).

FAIR4RS. (n.d.). *FAIR for research software*. https://www.rd-alliance.org/groups/fair-research-software-fair4rs-wg (accessed February 25, 2021).

FigShare. (n.d.). https://figshare.com/ (accessed February 27, 2021).

Garijo, D., Osorio, M., Khider, D., Ratnakar, V., & Gil, Y. (2019). OKG-Soft: An open knowledge graph with machine readable scientific software metadata. *15th International Conference on eScience (eScience)* (pp. 349–358). IEEE. https://doi.org/10.1109/eScience.2019.00046

Gil, Y., Ratnakar, V., & Garijo, D. (2015). Ontosoft: Capturing scientific software metadata. *Proceedings of the 8th International Conference on Knowledge Capture* (p. 32). https://doi.org/10.1145/2815833.2816955

GitHub. (n.d.). https://github.com/ (accessed February 25, 2021).

GitLab. (n.d.). https://gitlab.com/ (accessed February 25, 2021).

Gousios, G., Vasilescu, B., Serebrenik, A., & Zaidman, A. (2014). Lean GHTorrent: GitHub data on demand. *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 384–387). https://doi.org/10.1145/2597073.2597126

Guha, R. V., Brickley, D., & Macbeth, S. (2016). Schema.org: Evolution of structured data on the web. *Communications of the ACM*, *59*(2), 44–51. https://doi.org/10.1145/2844544

Guides, G. (n.d.). *Documenting your project in GitHub*. https://guides.github.com/features/wikis/ (accessed February 27, 2021).

Hucka, M., & Graham, M. J. (2018). Software search is not a science, even among scientists: A survey of how scientists and engineers find software. *Journal of Systems and Software*, *141*, 171–191. https://doi.org/10.1016/j.jss.2018.03.047

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019).Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint*, arXiv:1909.09436.

Ilyas, B., & Elkhalifa, I. (2016). *Static code analysis: A systematic literature review and an industrial survey* (Master's thesis). https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Abth-12871

Jaradeh, M. Y., Oelen, A., Farfar, K. E., Prinz, M., D'Souza, J., … Auer, S. (2019). Open research knowledge graph: Next generation infrastructure for semantic scholarly knowledge. *Proceedings of the 10th International Conference on Knowledge Capture* (pp. 243–246). https://doi.org/10.1145/3360901.3364435

Jones, M. B., Boettiger, C., Mayes, A. C., Smith, A., Slaughter, P., … Goble, C. (2017). *CodeMeta: an exchange schema for software metadata*. KNB Data Repository. https://doi.org/10.5063/SCHEMA/CODEMETA-2.0

Kelley, A., & Garijo, D. (2020). SoSEN-KG: Knowledge graph dump for SoSEN: Software Search Engine. *Zenodo*. https://doi.org/10.5281/ZENODO.3956451

Kelley, A., & Garijo, D. (2021). SOSEN-CLI first release. *Zenodo*. https://doi.org/10.5281/ZENODO.4574224

Lamprecht, A.-L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., … Capella-Gutierrez, S. (2020). Towards FAIR principles for research software. *Data Science*, *3*(1), 37–59. https://doi.org/10.3233/DS-190026

LIGO-VIRGO. (n.d.). *Software for gravitational wave data*. https://www.gw-openscience.org/software/ (accessed February 25, 2021).

Luan, S., Yang, D., Barnaby, C., Sen, K., & Chandra, S. (2019). Aroma: Code recommendation via structural code search. *Proceedings of ACM Programming Languages*, *3* (OOPSLA). https://doi.org/10.1145/3360578

Manghi, P. (2020).OpenAIRE research graph for research. *Zenodo*. https://doi.org/10.5281/zenodo.3903646

Mao, A., Garijo, D., & Fakhraei, S. (2019). SoMEF: A framework for capturing scientific software metadata from its documentation. *IEEE International Conference on Big Data* (pp. 3032–3037). https://doi.org/10.1109/BigData47090.2019.9006447

Mao, A., vmdiwanji, Garijo, D., Kelley, A., Dharmala, H., … jiaywan. (2020). KnowledgeCaptureAndDiscovery/somef: SOMEF 0.4.0. *Zenodo*. https://doi.org/10.5281/zenodo.4574207

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., … Schasberger, B. (1994). The Penn treebank: Annotating predicate argument structure. In *HUMAN LANGUAGE TECHNOLOGY: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8–11, 1994*. https://doi.org/10.3115/1075812.1075835

Maven Central Repository Search. (n.d.). https://search.maven.org/ (accessed February 27, 2021).

Miller, E., & Manola, F. (2004). *RDF primer* (W3C Recommendation). W3C. https://www.w3.org/TR/2004/REC-rdf-primer-20040210/

Miller, G. A. (1995).Wordnet: A lexical database for English. *Communications of the ACM*, *38*(11), 39–41. https://doi.org/10.1145/219717.219748

Mover, S., Sankaranarayanan, S., Olsen, R. B. P, & Chang, B.-Y. E. (2018). Mining framework usage graphs from app corpora. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering* (pp. 277–289). https://doi.org/10.1109/SANER.2018.8330216

Ó Carragáin, E., Goble, C., Sefton, P., & Soiland-Reyes, S. (2019). A lightweight approach to research object data packaging. *Zenodo*. https://doi.org/10.5281/zenodo.3250687

OpenAIRE. (n.d.). https://www.openaire.eu/mission-and-vision (accessed February 25, 2021).

Papers with code. (n.d.). https://paperswithcode.com/ (accessed February 27, 2021).

Peckham, S. D., Hutton, E. W., & Norris, B. (2013). A component-based approach to integrated modeling in the geosciences: The design of CSDMS. *Computers & Geosciences*, *53*, 3–12. https://doi.org/10.1016/j.cageo.2012.04.002

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., … Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Prlić, A., & Lapp, H. (2012). The PLOS computational biology software section. *PLOS Computational Biology*, *8*(11), e1002799. https://doi.org/10.1371/journal.pcbi.1002799

pydeps: Python module dependency visualization. (n.d.). https://github.com/thebjorn/pydeps

Pypi: The python package index. (n.d.). https://pypi.org/ (accessed February 27, 2021).

Rector, A., & Noy, N. (2006). *Defining N-ary relations on the semantic web* (W3C Note). W3C. https://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/

Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D., & Mitropoulos, D. (2021). PyCG: Practical call graph generation in Python. *2021*

IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1646–1657). https://doi.org/10.1109/ICSE43902.2021.00146

Scientific Paper of the Future Initiative. (n.d.). https://scientificpaperofthefuture.org/spf.html (accessed February 25, 2021).

Sefton, P., Ó Carragáin, E., Soiland-Reyes, S., Corcho, O., Garijo, D., … Portier, M. (2021). *RO-Crate Metadata Specification 1.1* (Tech. Rep.). https://doi.org/10.5281/ZENODO.3406497

Shamir, L., Wallin, J. F., Allen, A., Berriman, B., Teuben, P., … DuPrie, K. (2013). Practices in source code sharing in astrophysics. *Astronomy and Computing*, *1*, 54–58. https://doi.org/10.1016/j.ascom.2013.04.001

Smith, A. M., Katz, D. S., & Niemeyer, K. E. (2016).Software citation principles. *PeerJ Computer Science*, *2*, e86. https://doi.org/10.7717/peerj-cs.86

Software Carpentry. (n.d.). https://software-carpentry.org/about (accessed February 25, 2021).

Software Heritage. (n.d.). https://www.softwareheritage.org/ (accessed February 25, 2021).

Software Sustainability Institute. (n.d.). https://software.ac.uk/ (accessed February 25, 2021).

Tsay, J., Braz, A., Hirzel, M., Shinnar, A., & Mummert, T. (2020). AIMMX: Artificial Intelligence Model Metadata Extractor. *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 81–92). https://doi.org/10.1145/3379597.3387448

USGS. (n.d.). *US Geological Survey: Software for Water Resources Applications*. https://www.usgs.gov/mission-areas/water-resources/software (accessed February 25, 2021).

Vanschoren, J., van Rijn, J. N., Bischl, B., & Torgo, L. (2013). OpenML: Networked science in machine learning. *SIGKDD Explorations*, *15*(2), 49–60. https://doi.org/10.1145/2641190.2641198

Vrandečić, D., & Krötzsch, M. (2014). Wikidata: A free collaborative knowledge base. *Communications of the ACM*, *57*(10), 78–85. https://doi.org/10.1145/2629489

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., … Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, *3*, 160018. https://doi.org/10.1038/sdata.2016.18, PubMed: 26978244

APPENDIX A: IMPLEMENTING TF-IDF SEARCH USING A SPARQL QUERY

Figure A1 shows the SPARQL query that would be generated if users searched for the keywords “knowledge” and “graph”, using the description keyword method. To search for different keywords, we would modify the newline-separated list of keywords in the query. To use a different search method, the properties with the word “description” in them would be swapped out for their equivalent property names for the user-defined or title keywords.

The query works by first getting the global document count. Then, for each keyword in the list, it attempts to link that keyword string to a Keyword entity in the graph. If no match for the keyword is found in the graph, no document uses this keyword, and thus it is ignored. Additionally, the query retrieves the total number of documents that the keyword appears in. This, together with the document count, is used to compute the inverse document frequency (IDF).

Next, we have the Keyword entities together with their respective IDFs. The query now matches Keyword entities to Software entities that use this keyword in their description. We do this by looking for a QualifiedKeyword object that points to both specified keyword and software. The existence of the QualifiedKeyword object means that there is an edge from the software to the keyword, because the QualifiedKeyword exists to describe that edge. However, this does not mean that the keyword exists in the description; it could be in the title or user-defined list. Using the `sosen:inDescriptionCount` property, which tells us the number of times this keyword appears in the description of the Software entity, we then additionally get the `sosen:descriptionKeywordCount` property, which stores the total number of keywords in the software description. With these two properties, we can compute the term frequency (TF) of the keyword, which we multiply by its IDF to get the TF-IDF score.

With the keywords that match and their TF-IDF scores, the query computes the total number of keywords that matched and the sum of the TF-IDF scores of all matching keywords. The results are then sorted in descending order primarily by the number of keyword matches, with the TF-IDF score sums as a secondary key, used to break ties.

The biggest benefit of writing a keyword search as a SPARQL query is that it may be combined with other SPARQL queries. For example, we can do a keyword search but add filters, specifying that the software had to have a release in the last 6 months, use Python as a language, or have an open-source license.

```
PREFIX sosen: <https://w3id.org/okn/o/sosen#>
PREFIX sd: <https://w3id.org/okn/o/sd#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX math: <http://www.w3.org/2005/xpath-functions/math#>
SELECT ?software (COUNT(?keyword) as ?matches)
(SUM(?tf_idf) as ?sum_tf_idf)
WHERE {{
    SELECT ?keyword (math:log(?total/?doc_count) as ?idf)
    WHERE {{
        SELECT ?total WHERE {
          ?global a sosen:Global .
          ?global sosen:totalSoftwareCount ?total .}}
      ?keyword a sosen:Keyword .
      VALUES ?label {
        "knowledge" "graph"} .
      ?keyword rdfs:label ?label .
      ?keyword sosen:totalDescriptionInCount ?doc_count . }}
  ?qualified_kw a sosen:QualifiedKeyword .
  ?qualified_kw sosen:keyword ?keyword .
  ?qualified_kw sosen:inDescriptionCount ?kw_count .
  FILTER(?kw_count > 0) .
  ?qualified_kw sosen:software ?software .
  ?software sosen:descriptionKeywordCount ?total_kw_count .
  BIND((?kw_count/?total_kw_count) * ?idf as ?tf_idf)}
GROUP BY ?software
ORDER BY DESC(?matches) DESC(?sum_tf_idf)
LIMIT 10
```

Figure A1. A SPARQL query showing TF-IDF based keyword matching for the values `knowledge` and `graph`.

APPENDIX B: HEADER ANALYSIS EVALUATION DETAILS

Table B1 provides additional information on the precision and recall results obtained in the header analysis evaluation.

Table B1. Detailed header evaluation results

Category	Total	Correct	Incorrect	Missed	Precision	Recall	F-Measure
Description	20	13	5	7	0.72	0.65	0.68
Installation	82	72	16	10	0.82	0.88	0.85
Invocation	21	20	3	1	0.87	0.95	0.91
Citation	36	33	7	3	0.82	0.92	0.87
Usage	75	55	32	20	0.63	0.73	0.68
Documentation	18	18	2	0	0.90	1.00	0.95
Requirements	31	29	2	2	0.93	0.93	0.93
Support	9	6	8	3	0.43	0.67	0.52
License	30	30	0	0	1.00	1.00	1.00