SoMEF: A Framework for Capturing Scientific Software Metadata from its Documentation

Allen Mao*, Daniel Garijo†, Shobeir Fakhraei†
*University of California, Berkeley
allenmao@berkeley.edu
†University of Southern California, Information Sciences Institute
{dgarijo, shobeir}@isi.edu

***Abstract*—Scientific software has become a key asset to reproduce and understand the products of scientific research in many disciplines. However, scientific software is becoming increasingly complex and, as a result, researchers need to spend a significant amount of time finding, reading and understanding software documentation to set it up. In this paper we describe SoMEF, a Software Metadata Extraction Framework designed to help highlighting the most important parts of scientific software documentation. SoMEF processes the README files in GitHub repositories to automatically extract which parts of their text refer to the description, installation, invocation, or citation of a software component. Despite its simple features, SoMEF successfully categorizes README excerpts with a minimum 0.92 precision and 0.90 ROC AUC. These results, tested on a corpus of over 70 scientific software repositories, are a promising start towards automatically generating knowledge graphs of scientific software metadata.**

I. Introduction

Within the past few decades, computational science has increasingly become recognized as a fundamental approach to answer scientific questions alongside theory and experimentation [1]. However, the continuous development of new software makes it difficult for scientists to keep track of different method implementations and to evaluate whether a certain piece of software suits their needs [2]. Scientists are required to spend time poring through available software documentation and source code in order to understand the software used in a project [3] and how to properly cite it. This process is time consuming and unappealing to scientists due to the heterogeneity and lack of unified structure in software documentation.

Existing efforts have attempted to simplify this problem by avoiding "wordy, unstructured, introductory descriptions" in favor of a specialized language just for documentation [3]. However, text documentation continues to grow at an exponential rate [4].

In this paper we aim to ease the process of understanding, reusing and attributing scientific software by presenting SoMEF [5], a *Software Metadata Extraction Framework* that automatically extracts relevant software metadata from its documentation. SoMEF takes as input a README file from a GitHub repository and identifies its description, installation instructions, invocation setup and citation. Our approach uses binary classification methods and organizes the results into a structured format that is comprehensible to both humans and machines. In addition, SoMEF will extract additional metadata about a piece of software beyond its documentation by exploiting the GitHub REST API v3[1].

The paper is outlined as follows. We first define and illustrate the main metadata fields we extract from software documentation in Section II. Then, we cover our methodology and results in Section III. Finally, we provide an analysis of SoMEF in the context of related work and conclude the paper discussing related approaches and our planned future work.

II. Extracting Metadata from Software Documentation

We focus on four different categories of scientific software metadata: *description*, *installation instructions*, *invocation*, and *citation*. A *description* tells the reader what the software does, when to use it, and why to use it. *Installation instructions* are the set of steps and dependencies necessary for the for a software component to run. An *invocation* is the series of commands necessary to execute an instance of a software component. A *citation* provides credit to the authors who developed the work. Table I describes an example documentation[2] from these four categories.

These categories, constitute a basic software documentation, which emphasizes the following aspcects of software:

1) *Understandability*: users can easily recognize what a software component does, when to use it, and why to use it. A *description* provides most of this information in a concise manner.
2) *Usability*: users can quickly identify how to use a software component. The *installation instructions* and the *invocation* of the code are a crucial aspect for usability.
3) *Attribution*: users can identify traits or identifiers for the software. These traits are available elements

[1]https://developer.github.com/v3/
[2]README from https://github.com/whimian/pyGeoPressure

TABLE I
DEFINITIONS AND EXAMPLES FOR *description*, *installation*, *invocation*, AND *citation*.

Category	Questions Answered	Example
Description	*(understandability, discovery)*: What does this software do? When or why it?	A Python package for pore pressure prediction using well log data and seismic velocity data.
Installation	*(understandability, usability)*: What setup is necessary before I can use this software?	`pip install pygeopressure`
Invocation	*(understandability, usability)*: How do I use this software? Snippets?	`import pygeopressure as ppp` `survey = ppp.Survey("CUG")` `well = survey.wells['CUG1']`
Citation	Who should get credit for this software?	Yu, (2018). PyGeoPressure: Geopressure Prediction in Python. Journal of Open Source Software, 3(30), 992, https://doi.org/10.21105/joss.00992

of the *description*, but usually a *citation* is provided by authors to indicate other researchers how to credit them for their work.

Software documentation often contains headings that identify sections of text as one of the four categories we aim to recognize (i.e., *description*, *installation*, *invocation*, and *citation*). For example, "Overview", "Installation", "Usage", or "Citation" are common headings designed to help the reader locate pertinent sections. However, there is no standardized practice to structure documentation in README files. One possible approach to locate information pertinent to each category would be to separate different sections by section headers, i.e. pattern matching [4]. However, this method is not robust because different authors structure, word, and describe their documentation differently. This paper explores an information retrieval approach that exploits Machine Learning techniques to extract each of these four categories, i.e. *description*, *installation*, *invocation*, and *citation* from software documentation. This approach consists of a binary classifier for each category that predicts whether or not a text excerpt belongs to a certain documentation category.

A. Corpus

With more than 96 million repositories in 2018, GitHub is the largest host of source code today [6]. For this reason, this paper focuses on the software metadata of GitHub repositories. GitHub's popularity has contributed to the diversity of its projects, which range from web development tools like Facebook's React [3] to machine learning libraries like Google's Tensorflow.[4] When a user selects a repository on GitHub, GitHub presents two displays: a list of files in the root directory and a rendered version of the default README. We therefore focus by default on READMEs as the source of documentation, as a README is usually a repository's main source of documentation.

1) Selection Criteria: In total, the corpus consists of the README documentation from 74 GitHub repositories and manual annotations (see Section II-A2). GitHub's enormous number of repositories signifies diversity in documentation maturity, software purpose, and programming language. Since the overall goal of this project is to improve scientific transparency, scientific software contributed the most to the selection of repositories, although other well-documented software, e.g. for web development, made their way into the corpus too. *Awesome*[5] curations of repositories for scientific domains or tools such as Geoscience, GeoJSON, Open Climate Science, Geospatial, etc. provided links to relevant scientific projects whose quality of documentation served as the metric for inclusion.

While software code does not often manifest directly in documentation, programming language variability does influence the syntax of installation and invocation commands (e.g. for package managers such as *pip* or *npm*. As a result, repositories covered a wide variety of programming languages. Table II shows the total number of bytes that each programming language took up across all repositories in the corpus.

TABLE II
TOTAL BYTES OF LANGUAGES IN COLLECTED REPOSITORIES

Language	Bytes	Percent
C++	77692823	32.66%
Python	74125620	31.16%
Jupyter Notebook	55637798	23.39%
JavaScript	11187134	4.70%
HTML	6633188	2.79%
Lasso	2377884	1.00%
Go	1670128	0.70%
Cuda	1377119	0.58%
C	1191690	0.50%
Other	6010474	2.53%
Total	237903858	100%

2) Annotation Specifications: (GitHub) Markdown format syntax can provide hints to the function of an excerpt. For example, three back ticks (```) identify

[3]https://github.com/facebook/react
[4]https://github.com/tensorflow/tensorflow

[5]https://awesome.re/

a block of code text. Sometimes, a researcher appends the programming language name to the back ticks to produce correct syntax highlighting. These code texts often display the installation or invocation instructions for a software.

While such properties may help a human identifying the information they need from documentation, we cannot rely on them because a researcher may choose to not follow this practice. Similarly, researchers may provide descriptive instructions instead of specific commands, e.g. for GUI applications. Consequently, the corpus consists of plain text **rendered** Markdown. For convenience, we used a newline delimiter which splits the corpus into paragraph excerpts. Table III displays the number of excerpts per category and the average excerpt length.

TABLE III
NUMBER OF PARAGRAPH EXCERPTS COLLECTED

Category	Count	Mean Excerpt Length (words)
Description	275	27.95 ± 28.46
Installation	719	9.24 ± 11.39
Invocation	1092	7.74 ± 9.88
Citation	252	8.20 ± 7.40

3) Balancing the Corpus: For each binary classifier, we transformed the corpus to label the category to be predicted as **True** and all other categories as **False**. This transformation resulted in an unbalanced corpus. Therefore, we sampled negative excerpts such that corpus is composed of 50% positive samples and 50% negative samples. In addition to the three other categories, the negative class contained random samples of control sentences from the Treebank [7] corpus as a background to make the system more robust, e.g. to ensure that the classifiers do not devolve into a code vs natural text classifier. All four categories of negative text contribute equally to the 50% negative excerpts. Table IV illustrates a breakdown of the *description* corpus as reference for all other corpora. An approximate ratio is given in the third column because rounding changes the ratios slightly.

TABLE IV
DESCRIPTION CORPUS BREAKDOWN

Truth Value	Category	Apprx. Ratio	Count
True	Description	0.5	275
False	Installation	0.125	68
	Invocation	0.125	68
	Citation	0.125	68
	Treebank	0.125	68
	Total	1.0	547

B. Data Preparation

Stemming algorithms are useful because they prevent a feature matrix from becoming too sparse. However, while word stemming is useful for natural language, command line inputs and computer language lexicons are very precise and do not exhibit variations found in human languages. To maintain this distinction, we used the scikit-learn [8] *TfidfVectorizer* to compute a TF-IDF matrix of unigram features without any stemming, stop words, or stemming to perform an initial analysis.

C. Classifiers

We used of the following classifiers from the Scikit-learn package [8].

1) Logistic Regression: liblinear solver and balanced class weights because of small corpus size and imbalances that may arise from undersampling. We selected a Logistic Regression rather than a Support Vector Machine in order to obtain probabilities for better insights into model performance.

2) Multinomial Naive Bayes (MNB): additive smoothing parameter of $\alpha = 1$ as default fail-safe probability.

We focused on these two classifiers in this study because they provided a good introductory analysis of our approach. In particular, we were curious to see how these two approaches would contrast each other since Naive Bayes is a generative model while Logistic Regression is a discriminatory model.

III. RESULTS AND DISCUSSION

We performed a stratified 5-fold cross validation to evaluate the performance of each candidate model. We used the same random seed selection of samples in order to balance and and split the corpus. Tables V and VI show summary metrics of our results, with average and standard deviation of the *Receiver Operating Characteristic's area under the curve* (ROC AUC) for each fold; and the average precision of each fold's for the Logistic Regression and MNB classifiers, respectively.

For a given text category, the Logistic Regression and Multinomial Naive Bayes (MNB) classifiers yielded similar results. Both classifiers identified citations with

TABLE V
LOGISTIC REGRESSION CROSS VALIDATION SUMMARY

Category	Mean ROC AUC	Mean Average Precision
Description	0.90 ± 0.03	0.92 ± 0.04
Installation	0.96 ± 0.01	0.97 ± 0.01
Invocation	0.94 ± 0.02	0.96 ± 0.01
Citation	0.97 ± 0.01	0.98 ± 0.01

TABLE VI
MULTINOMIAL NAIVE BAYES CROSS VALIDATION SUMMARY

Category	Mean ROC AUC	Mean Average Precision
Description	0.91 ± 0.04	0.93 ± 0.05
Installation	0.96 ± 0.01	0.97 ± 0.01
Invocation	0.95 ± 0.02	0.97 ± 0.01
Citation	0.97 ± 0.01	0.98 ± 0.01

the highest precision and AUC, and descriptions with the lowest precision and AUC. This consensus validates our approach to identify sections of documentation based on their vocabularies.

Despite features so simple as unigrams without stemming or lemmatization, the performance metrics from V and VI demonstrate promising results because all classifiers had low rates of Type I errors (hence average precision of 0.93+) and had an average 0.91+ minimum probability of ranking a positive sample higher than a negative sample.

A. Example Output

In order to illustrate our approach, the following excerpt exposes the results of our classifiers on the first ten lines of the README of the `PyGeoPressure` repository.

```
[{'description.sk':
    ↪ 0.4497832746871151, 'excerpt':
    ↪ '<!-- # pyGeoPressure -->'},
 {'description.sk':
     ↪ 0.39746669619634734,
  'excerpt': '<img src="docs/img/
      ↪ pygeopressure-logo.png" alt="
      ↪ Logo" '
          'height="240">'},
 {'description.sk':
     ↪ 0.7378379663396829,
  'excerpt': 'A Python package for
      ↪ pore pressure prediction using
      ↪  well log '
          'data and seismic velocity
             ↪ data.'},
 {'description.sk':
     ↪ 0.4734045518090183, 'excerpt':
     ↪ 'Cite pyGeoPressure as:'},
 {'description.sk':
     ↪ 0.4380679608688805,
  'excerpt': 'Yu, (2018).
      ↪ PyGeoPressure: Geopressure
      ↪ Prediction in Python. '
  ...}
...]
```

The result is returned as a JSON file which contains the contents of the README separated by paragraphs (newlines), along with their classification score. For example, the excerpt "A python package for pore prediction ..." was classified as being close to be a software description. While our default threshold remains at 0.5, we continue to analyze how different categories merit different thresholds.

IV. Extracting Other Relevant Software Metadata

A manual inspection of software documentation may allow a users to identify what a software component does, how to install or use it, and how to cite it. But there are several other important attributes of software metadata that are useful in the distribution and organization of software. Examples of such attributes include:

1) *owner of the software* for attribution and accountability
2) *license of the software*, which notifies researchers on the availability and usage restrictions of a piece of software
3) *number of forks*, which represent how the community perceives the usefulness of a software component
4) *programming language* to help users identify software that match their requirements
5) *list of releases* to help distinguish different versions of a software component and how often it gets maintained.

In SoMEF, we developed a component for fetching metadata from GitHub repositories by using the GitHub APIs.[6] This kind of metadata also helps complementing any gaps left by our binary classifiers, as in some cases it includes short descriptions that can be used to improve our results. We describe further details on our approach below.

A. Inputs and Outputs

SoMEF takes a GitHub repository URL as a command line argument and returns a JSON file with the following fields:

1) description
2) programming languages
3) license (including name and URL)
4) repository name
5) repository owner
6) list of releases each with author name, description, release name, tag name, and URLs to the HTML, link to tarball, release, and link to zipball
7) topics

We illustrate SoMEF with the Tensorflow repository[7] in the following excerpt:

```
{'description': 'An Open Source
    ↪ Machine Learning Framework for
    ↪ Everyone',
 'forks_url': 'https://api.github.com/
     ↪ repos/tensorflow/tensorflow/
     ↪ forks',
 'languages': ['C++',
              'Python',
              'HTML',
              ...
               ],
```

[6]https://github.com/KnowledgeCaptureAndDiscovery/SM2KG/blob/master/helper_scripts/fetchgithubmetadata.py
[7]https://github.com/tensorflow/tensorflow

```
 'license': {'name': 'Apache License
    ↪ 2.0',
           'url': 'https://api.github.
              ↪ com/licenses/apache
              ↪ -2.0'},
 'name': 'tensorflow',
 'owner': 'tensorflow',
 'readme_url': 'https://github.com/
    ↪ tensorflow/tensorflow/blob/
    ↪ master/README.md',
 'releases': [{'author_name': '
    ↪ goldiegadde',
             'body': '# Release 1.15.0-
               ↪ rc3\r\n'
                ...
             }, ... ],
'topics': ['tensorflow',
          'machine-learning',
          'python',
          'deep-learning',
          'deep-neural-networks',
          'neural-network',
          'ml',
          'distributed']}
```

V. Related Work

In this section, we first summarize other work that analyzes or extracts information from unstructured documentation. We use these studies as a point of comparison to distinguish SoMEF's approach. Then, we discuss how SoMEF ties into software organization in software registries.

A. Characteristics of a Good README

SoMEF rests on the assumption that the README is informative and has the information that addresses *understandability*, *usability*, and *attribution*. However, SoMEF is not very useful if the README is deficient in quality. Previous work [9] demonstrates an analyzer that measures a README's quality based on the most starred repositories of a language. While SoMEF and [9] use similar classification models, they have different ground truths because SoMEF looks for four specific software metadata categories while [9] is based on an average of most starred GitHub repositories.

B. Software Documentation to Development Tasks

In [10] the authors analyze a documentation corpus and organize unstructured documentation into tasks such as "add widget" or "add widget to page". While its approach is more complicated than SoMEF, the scope of its problem is also different. SoMEF is more concerned with smaller pieces of documentation, e.g. READMEs, and searches for installation and invocation commands, e.g. "`pip install pygeopressure`" or "`import pygeopressure as ppp`", respectively, rather than development tasks. Furthermore, SoMEF also seeks to extract description and citation information to make software more understandable, and usable.

C. Software Metadata Registries

Domain-specific disciplines have developed software metadata registries with metadata descriptions of their commonly used software, complementing code repositories which only store software code. Examples of software metadata registries include the Community Surface Dynamics Modeling System (CSDMS) for Earth surface processes [11], the Astrophysics Source Code Library (ASCL) in astrophysics [12]; and OntoSoft [13] and OKG-Soft [14] in geosciences.

Software registries are very useful points of entry for researchers to search and understand complex software models. However, the software metadata entries in these registries are usually curated by hand by experts who have to skim through unstructured information of variable quality. SoMEF is complementary to these efforts, as it may be used to automatically suggest new candidate entries in a more efficient and structured manner.

VI. Conclusions and Future Work

In this paper, we have described a novel approach that employs linguistic features to extract useful software metadata, i.e. *description*, *installation*, *invocation*, and *citation*, from software documentation. We have implemented our approach with a baseline model that extracts unigram features without stemming, lemmatization, or stop words and obtains a minimum average 0.92 precision and 0.90 ROC AUC. We supplement this model with a framework that fetches additional software metadata. The baseline results are promising and yield many possibilities for future work, which we further describe below:

- **Corpus expansion**: Since the 74 GitHub repositories from the corpus are too few in number to represent GitHub, let alone software as a whole, we hope expand the corpus by soliciting the help of crowdsourcing platforms such as *Amazon Mechanical Turk*[8]. However, as identification of README components requires human interpretation, this step demands a guideline to standardize results and filter out statistically erroneous interpretations and annotator bias. We also hope that with this large a corpus, we will be able to explore state of the art deep learning techniques to capture semantic elements within software documentation.
- **Text separation**: Currently, the model trains and tests on samples of whole paragraphs, i.e. sections surrounded by newline characters, because newlines provided a convenient method to automatically split

[8]https://www.mturk.com/

text. However, this method leads to two potential problems: one entire paragraph may contain multiple classes sentence after sentence and some classes, especially citation, consist of entire text blocks instead of individual sentences. We hope to refine the corpus to better reflect these realities.

- **Exploring other linguistic features**: The programming language that built a software can influence the instructions to install or invoke the software. For example, installation of python software can require prerequisite packages and the `python` command runs these software. We hope to explore how feature vectors that encode language-conscious information may improve detect or distinguish installation or invocation commands.
- **Towards knowledge graphs of scientific software metadata**: Eventually, we hope to take advantage of this research to automate the effort that scientists spend on setting up scientific software. This paper establishes the first step, identifying metadata within a documentation, and we hope it will help us teach a computer to set up software by itself without human intervention. Before we get there, however, we want to take the organized structures, i.e. the JSONs, that SoMEF creates and integrate them into Knowledge Graphs (KGs). KGs provide a structured way to provide relationships between different entities, thus making software more accessible and easier to find.

Acknowledgments

This work was performed as part of a Research Experience for Undergraduates (REU) program, supported by National Science Foundation (NSF) grant #1659886. We would also like to thank Rosna Thomas for her help towards building the scientific software metadata corpus; and Yolanda Gil for her feedback.

References

[1] Prakash Prabhu, Hanjun Kim, Taewook Oh, Thomas B Jablin, Nick P Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, David Walker, Yun Zhang, et al. A survey of the practice of computational science. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[2] Douglass E Post and Lawrence G Votta. Computational science demands a new paradigm. *Physics today*, 58(1):35–41, 2005.

[3] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, 2011.

[4] Gabriele Bavota. Mining unstructured data in software repositories: Current and future trends. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 1–12. IEEE, 2016.

[5] Allen Mao, Rosna Thomas, and Daniel Garijo. KnowledgeCaptureAndDiscovery/SM2KG: SoMEF 0.0.1: First release of the framework, October 2019.

[6] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th working conference on mining software repositories*, pages 384–387. ACM, 2014.

[7] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank: annotating predicate argument structure. In *Proceedings of the workshop on Human Language Technology*, pages 114–119. Association for Computational Linguistics, 1994.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Besir Kurtulmus. How to write a good github readme using data science, Oct 2017.

[10] Christoph Treude, Martin P Robillard, and Barthélémy Dagenais. Extracting development tasks to navigate software documentation. *IEEE Transactions on Software Engineering*, 41(6):565–581, 2014.

[11] Scott D. Peckham, Eric W.H. Hutton, and Boyana Norris. A component-based approach to integrated modeling in the geosciences: The design of csdms. *Computers Geosciences*, 53:3 – 12, 2013. Modeling for Environmental Change.

[12] Lior Shamir, John F. Wallin, Alice Allen, Bruce Berriman, Peter Teuben, Robert J. Nemiroff, Jessica Mink, Robert J. Hanisch, and Kimberly DuPrie. Practices in source code sharing in astrophysics, 2013.

[13] Yolanda Gil, Varun Ratnakar, and Daniel Garijo. Ontosoft: Capturing scientific software metadata. In *Proceedings of the 11th International Conference on Knowledge Capture*, page 32. ACM, 2015.

[14] Daniel Garijo, Maximiliano Osorio, Deborah Khider, and Yolanda Ratnakar, Varun Gil. OKG-Soft: An Open Knowledge Graph with Machine Readable Scientific Software Metadata. In *Proceedings of the 15th International Conference on Knowledge Capture*, page 32. ACM, 2019.