Evaluating Software Documentation Quality

by

Henry Kai Wei Tang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Software libraries provide reusable code that allow developers to include needed functionality without committing time and effort to develop the functionality themselves. To benefit from the code reuse, developers first compare multiple libraries that offer the needed functionality and spend time learning how to use the library. This is typically done through a combination of trying the library, as well as reading the library's documentation. However, "bad quality" documentation is not only ineffective in aiding developers learn a library, but may even be detrimental if there is incorrect information. Thus, it would be useful if developers already had an idea about the documentation quality of a library before they invest time in exploring it.

In this thesis, we investigate a metric-based evaluation of software documentation quality. We start by exploring the literature to confirm the importance of documentation when learning a software library before finding descriptions of documentation aspects, as well as previous attempts to evaluate documentation quality. We summarize the different quality aspects of documentation, then narrow our focus to six different documentation aspects: Code Examples, Completeness, Ease of Use, Readability, and Up-to-date. From these documentation quality aspects, we utilize the Goal-Question-Metric [60] paradigm to determine the metrics that can be used to measure them. We validate these initial metrics through an interview study of three industry professionals, where we integrate the feedback back into our metrics. After incorporating the interview feedback, we create a summary tool presenting each

of our metrics for a given library and conduct a survey with 25 developers on the usefulness of our tool and metrics. Participants found our documentation quality summary useful with a median score of four on a five-point Likert scale [36] rating system. Participants especially found the metrics relating to text readability and code example readability as useful documentation quality metrics. We also discuss opportunities and future applications of our work.

Preface

This thesis is an original work by Henry Tang. The research project, of which this thesis is a part, received ethics approval from the University of Alberta Research Ethics Board, Project Name "Task-based Code Recommender Systems", No. Pro00074107, December 12, 2018 - April 19, 2023.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software libraries provide reusable code that allows developers to include needed functionality without having to commit time and effort to develop the behaviour themselves. To benefit from such code reuse, developers usually first compare multiple libraries that offer the needed functionality [41], [54], [64] and then spend time to learn the selected library for efficient integration into their code [52]. Fortunately, most libraries include documentation that helps in library selection [20]–[22], [34], [41], [53], [61], [72] as well as in learning how to use a library. However, if the documentation is "bad quality" then it is not only ineffective in assisting developers to use the library, but it may cause further confusion, e.g., if the information is incorrect or out of date. Therefore, one challenge of using software libraries is the unknown amount of time that developers must spend in order to properly learn multiple libraries to be able to compare and select the right library for their needs. Libraries with "better" documentation quality will take less time for a developer to learn, which raises the question of, "How to evaluate library documentation quality?" This thesis builds upon the research of library comparison by De la Mora and Nadi [41]. De la Mora and Nadi [41] created a tool that compared libraries across different aspects, such as the popularity of a library or its issue response and closing times, which assisted developers in evaluating and comparing software libraries. They also surveyed developers about what other aspects they would like to see, and documentation quality was among the top-requested aspects. However, presenting this aspect first requires a way

to measure or assess documentation quality, which is the focus of this thesis.

Existing efforts for evaluating documentation quality focus on specific areas of documentation that do not necessarily reflect the qualities of documentation that are important to developers in order to learn and use a library. Additionally, some of the existing work propose metrics that are hard to calculate and do not provide tools that would automatically calculate the metrics for developers. For example, Forward [19] developed a measure of documentation called AURA, which measures an **A**rtefact's "**U**sefulness", "**R**eferential decay", and "**A**uthority". Although Forward [19] conducted a survey to find the important parts of documentation that developers found useful, their scope was on every type of documentation (e.g., requirement, specification, design documents, etc.), rather than focusing on the documentation that client developers consider when using software libraries. Aversano et al. [3] measured the "completeness", "up-to-dateness", and "readability" of open source documentation. However, their evaluation of their metrics is based on a single case study on one open source system and they do not provide a tool that automatically calculates their metrics.

In this thesis, our goal is to create an automatic documentation quality evaluation tool to aid developers in comparing and selecting a library to learn and incorporate into their own projects. To evaluate a library's documentation, we first determine what aspects of documentation should be measured, then create metrics for these areas, and finally combine the metrics into a single summary tool. We then conduct both an interview study of industry professionals and a survey attached to the summary tool in order to validate our metrics and usefulness of our summary tool.

Summarized, our research objectives are to:

- Determine which type of documentation is important for learning and using a software library
- Determine what the "aspects" of documentation quality are
- Determine how best to measure these "aspects" and define implementable metrics

- Implement a summary tool utilizing these metrics
- Validate the usefulness of the summary tool

1.1 Thesis Overview and Organization

Figure 1.1 shows the research process we follow and how this thesis is organized. This thesis begins with an overview of related work in the area of software documentation (Chapter 2). Our related work starts by exploring the different types of documentation important to developers using software libraries before identifying documentation quality aspects. Our goal here is to determine which documentation types developers utilize when using software libraries, and to determine "good" and "bad" documentation quality aspects by reviewing and aggregating different features identified in the software documentation literature. After identifying documentation quality aspects, we then quantify these aspects by defining six metrics using the Goal-Question-Metric (GQM) paradigm [60]. The documentation quality aspects we measure in this thesis are completeness, readability, code examples, ease of use, and up-to-dateness.

We then implement the metrics for these documentation quality aspects and construct a documentation quality summary overview, shown in Figure 4.1, where we present our developed metrics. To help with the iterative development of the metrics and the summary overview, we validate our initial metrics through interviews with three professional software developers for feedback. The responses from the interviews support the metrics we conceived and the participants agreed that the metrics give an approximation to various aspects of the quality of software documentation. We update our metrics and presentation using the interview feedback. Afterwards, Chapter 5 describes our implementation of the metrics along with the validation of each metric against manually constructed ground truths. We then finalize the documentation quality summary and create a public web-based summary tool that allows developers to evaluate the documentation quality of libraries they are interested in. Figure 4.2 shows our final documentation quality overview summary.

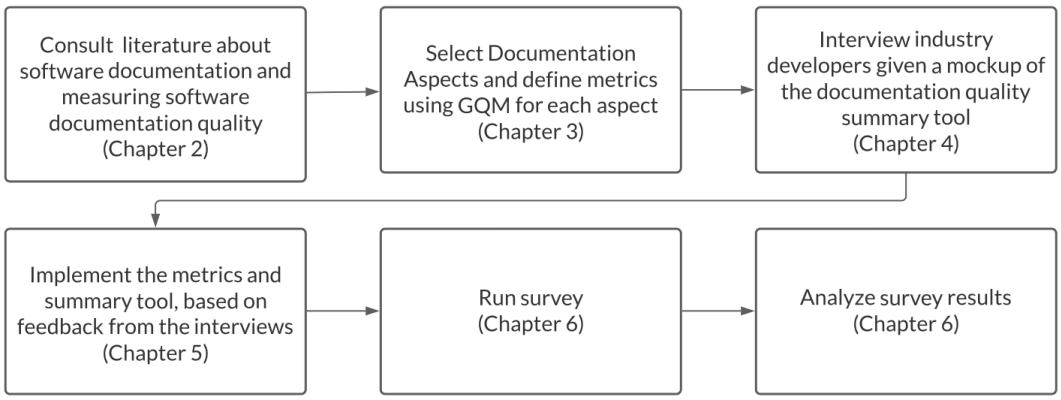

Figure 1.1: Process overview of creating our summary tool

We use this summary tool to conduct a broader evaluation of our metrics and documentation quality summary page. The tool allows a user to provide a link to a GitHub repository and the homepage of the library's documentation, which it then processes to create a summary page with our metrics. The user is then asked to complete a survey that gauges the usefulness of the summary page and its metrics. We currently support processing documentation from Python, Java, and JavaScript libraries.

We received 25 survey responses for 40 analyzed libraries across all three of the supported languages. Overall, participants found our summary to be useful with responses having a median score of four out of five when asking participants about the "usefulness" of our tool. While most respondents found the summary tool to be overall useful, the most highly rated metrics are the "Text readability" and "Code readability" metrics, and the lowest rated metric for participants is the "Documentation/Source code similarity" metric. This implies that library maintainers should focus more on clearly communicating information through the documentation, rather than spending effort to document every method available in the source code.

Finally, we discuss our threats to validity, and challenges and opportunities this work enables. Our work not only provides developers a tool to quickly evaluate a library documentation's quality, but it also assists library maintainers by highlighting the documentation quality aspects they should focus on for their own library's documentation. This work also opens up the development

for creating additional metrics to measure other documentation aspects, as well as extending our current metrics to be more accurate and generalizable, such as supporting an additional programming language.

To summarize, the contributions of this thesis are:

- The definition, automation, and verification of six metrics for evaluating documentation quality, derived from "good" and "bad" traits we extracted from existing literature.
- An implementation of these six metrics for eight libraries in three programming languages (Python, Java, and JavaScript) for an initial validation with industry professionals.
- An interview study with three professional developers that provide intermediate feedback to help us improve our metrics and their presentation.
- A web application [1] that accepts a library link as input and produces an overview summary of a library's documentation quality. To date, the service processed 40 total libraries (22 Python libraries, 5 Java libraries, and 13 JavaScript libraries).
- A survey with 25 developers to evaluate the accuracy and usefulness of the documentation quality summary page and metrics.

[1] https://smr.cs.ualberta.ca/docquality/

Chapter 2

Literature Review

In this chapter, we discuss three directions of related work exploring documentation quality. We use the first two directions (documentation scope and documentation quality aspects) to establish the importance of software documentation and the "good" and "bad" traits of documentation quality. The third direction summarizes previous attempts in evaluating documentation quality. All three of these directions serve to motivate and provide relevant background for our documentation quality metrics.

2.1 Documentation Scope

Software documentation can describe various types of documentation that appear during the software development and maintenance lifecycle [1], [20], ranging from requirements, architectural documents, to even source code comments. Additionally, documentation can have different target audiences, e.g., the system's developers, maintainers, or end users. In our work, we focus on documentation that is meant to help library client developers understand and use a software library. This falls under the *Operation & Maintenance* category defined by Aghajani et al. [1], where the implementation and testing of a software have been completed and the software is made available to end users or passed to subsequent maintainers. In the context of a software library, which is the focus of our work, end users are the developers wishing to use the library (i.e., *client developers*). Thus, we focus on documentation that conveys usage information about the library, including the public Application Programming

Interface (API), installation instructions, and interaction with other software.

Given this scope when evaluating documentation quality, this thesis focuses on the official documentation provided by the library maintainers as an external website because this type of official documentation is typically readily available online and, according to Venigalla and Chimalakonda [68], is what most developers will trust when learning a library. This sentiment is also echoed by Curtis et al. [15], Jazzar [29], and Vargas et al. [34] whose respective work identified documentation as an important part of software development and maintenance, even if in practice documentation suffers too many issues to be useful. Although Javadoc documentation is an external website readily available online, given the choice, we opted to analyze manually created documentation by the library maintainers. Our intuition is if library maintainers provide external documentation, then that official documentation will contain the most up-to-date information that the maintainers wanted to convey to library users. Additionally, dedicated library documentation contains added information, such as installation instructions and interaction with other software. Even though we opted against Javadoc given the option, our tool still supports analyzing Javadoc pages.

We focus on the official documentation sources because an indication of the official documentation's quality reflects the effort of the library maintainers to assist developers in using their library. A rating of a library's documentation quality will assist developers in deciding which library to learn and incorporate into their own project.

2.2 Documentation Quality Aspects

In order to develop metrics that evaluate library documentation quality, we first need to learn and establish which aspects of documentation enhance or detract from its quality. In this section, we discuss what previous researchers have discovered through various surveys and studies. We summarize this information in Table 2.1. This table shows the different kinds of documentation quality aspects, defines each of them, and provides a list of references that

identify or consider this aspect.

Forward and Lethbridge [20] conducted a survey to understand the kinds of documentation used by various roles in a software project, including managers, project leaders, software architects, and software developers. The authors also wanted to understand which attributes of documentation are most important in general, across all roles. The following is the list of attributes in their survey, ordered by importance from highest to lowest based on their participants' responses:

1. Document content
2. Up-to-dateness
3. Availability (of the document)
4. Use of examples
5. Organization (sectionalization of content)
6. Type of document (requirements, specifications, etc.)
7. Use of diagrams
8. Navigation (internal/external links)
9. Structure (arrangement of text/diagrams on the document)
10. Writing style
11. Length of documentation
12. Spelling/grammar
13. Author
14. Influence to use the document
15. Format (PDF, HTML, etc.)

Table 2.1: Summary of Documentation Quality Aspects

Aspect	Definition	References
Accessibility	How difficult is it to find the documentation?	[20], [61], [72]
Appeal	How interesting is it to read?	[65]
Appropriateness comments	Density of source code comments	[3]
Authority	How much authority does the documentation have?	[19], [20], [72]
Code examples	The existence of code examples in the documentation	[20], [21], [52], [53], [67], [70]
Cohesion	How well does the documentation fit together?	[65]
Completeness	Is the information in the documentation complete? Is all the source code documented? Is all the tasks/features documented?	[2], [3], [8], [21], [53], [61], [67], [72]–[74]
Consistency	Is the documentation consistent with itself? Same terminology, same format, etc.	[8], [21], [29], [65], [67], [72]
Consistency (to standard)	Does the documentation conform to a documentation standard defined by an external authority?	[3], [61]
Correctness	Is the information in the documentation accurate?	[2], [21], [67], [72]
Documentation PoV	The documentation should be written in the point of view of the reader.	[4]
Ease of Use	How easy is the documentation able to be used? e.g., navigation, internal and external links	[3], [20], [52], [67], [70]
Effectiveness	Does the documentation make effective use of technical vocabulary?	[65]
Fitness of purpose	Does the documentation fit its intended purpose	[4]
Format	What file format is the documentation? e.g., HTML, PDF, etc.	[20]
Graphical Support	Does the documentation use images?	[3], [20], [21]
Length	The length of the sentences in the documentation.	[3], [20]
Maintainability	How easy is the documentation able to be updated?	[2]
Organization	Is the information in the documentation organized efficiently? e.g., Sections/subsections	[3], [4], [20], [21], [53], [61], [65], [67], [72]
Preciseness	How precise is the documentation?	[21]
Quality	How well written is the documentation?	[65]
Readability	How easy is the documentation read?	[2], [3], [20], [21], [29], [65], [67], [72]
Record rationale	Does the documentation include design decisions?	[4]
Relevance of content	Is the information in the documentation relevant?	[4], [20], [21], [53], [65], [67]
Spelling and Grammar	Spelling and grammar of the documentation	[20], [72]
Support many scenarios	Does the documentation support many scenarios?	[53]
Traceability	What is the extent to which changes in the documentation can be tracked?	[72]
Type	What is the type of documentation? e.g., Requirements, Specifications, Testing, etc.	[20]
Understandability	How understandable is the information in the documentation?	[4], [65], [67]
Up-to-date	How up to date is the documentation relative to the source code?	[2]–[4], [8], [19]–[21], [29], [61], [67], [72]
Usability	To what degree in which users can use the documentation to achieve objectives?	[2]
Usefulness	How useful is the documentation?	[19]

Yellow highlighted rows are the selected aspects

From this list, we see that the most important attribute about documentation is the content, followed closely by its up-to-dateness. This is logical as documentation serves a purpose of informing readers about the software. If this information is not reliable, then that documentation becomes meaningless. The next most important attributes of this list then deal with documentation attributes that are secondary to the information provided in the documentation. For example, making sure users can find the documentation, that there are examples that show how to use or explain parts of the software, and making sure that the documentation is properly organized and users can find the information they need. Each of these attributes are shown in Table 2.1.

Aghajani et al. [1], [2] analyzed 878 artefacts (Stack Overflow threads, issue trackers, pull requests, and mailing lists) that may reveal documentation problems to create a taxonomy of documentation issues. The authors summarized their taxonomy of documentation issues into three overarching categories and multiple subcategories to group the specific documentation issues. They then performed two surveys, where the first survey determines the importance of the different documentation issues in their taxonomy, and the second survey determines the useful documentation types in the various software engineering tasks. The authors' first survey showed the importance of specific documentation issues within each larger category. For example, under the "information content" category, they have a subcategory called "completeness", which details issues surrounding whether the documentation adequately covers the features of the library. Important issues for this subcategory were insufficient installation, deployment and release documents, developer guidelines, and user documentation. We treat their subcategories as documentation aspects and use the important documentation issues, along with the attributes from Forward and Lethbridge [20] above, to help us in selecting which documentation aspects to focus on. We discuss this in detail in Chapter 3. We also note that the results of the authors' second survey show that official documentation sources of a library (e.g., user manual, migration guide, installation guide, etc.) are useful in the operations and maintenance phase of a library, which supports the focus of this thesis.

Robillard [53] explored the issues developers face when learning APIs. They surveyed 80 participants and found issues relating to the architecture of the API, the experience of the learning developer, and issues with learning resources, such as documentation. We focus on the documentation issues their survey results show, which are: insufficient or inadequate examples, incomplete content, no reference on accomplishing specific tasks with the API, inadequate high-level information about the API, and undesirable presentation formats. These findings align with the results by Aghajani et al. [1], [2] about issues that diminish documentation quality and hinder learning of a library.

Uddin and Robillard [67] also explored the failings of documentation through two surveys. Their first survey collected common documentation problems, which they combined into 10 categories, and their second survey collected each of the problem's frequency and severity. The 10 created categories from 69 respondents of the first survey are:

1. Incompleteness
2. Ambiguity
3. Unexplained (code) examples
4. Obsoleteness
5. Inconsistency
6. Incorrectness
7. Bloat
8. Fragmentation
9. Excess structural information
10. Tangled information

The first six problems relate to the documentation content, while the last four relate to the documentation presentation. From their second survey, Uddin and Robillard [67] found "incompleteness", "incorrectness", "ambiguity", and

"unexplained (code) examples" as the most severe issues from their list. These results coincide with the issues and severities concluded by the previously described studies in this section [1], [2], [53].

Garousi et al. [21] conducted two surveys: the first to understand how a library's documentation (requirements, design, test, process documents, and code comments) is used for both the process of developing and maintaining software, and the second about the attributes that affect documentation quality. Since we are interested in documentation quality aspects, we focus on the second survey. Although Garousi et al. [21] do not focus on the official documentation of a library, the documentation aspects they specified are generalizable to other types of documentation and align with literature already discussed. The results of their survey showed that the most important aspects of documentation quality are readability, relevance of content, and organization. Garousi et al. [21] also defined "lacking" documentation aspects, meaning these aspects were desirable aspects of documentation but were consistently below their expected quality, e.g., documentation is expected to be up to date, but is typically out of date. These "lacking" aspects were up-to-dateness, preciseness, and use of examples.

While there is additional work that also explores the issues and attributes of documentation [4], [52], [65], [70], we do not discuss them in detail here as they include similar documentation quality aspects already discussed above. Instead, we summarize all quality aspects and the references that discuss them in Table 2.1. Table 2.1 orders the different aspects alphabetically and provides the supporting references for each aspect. We use this table to select the documentation aspects to measure, as detailed in Chapter 3.

2.3 (Automatically) Measuring Documentation Quality

In this section, we discuss previous efforts in automatically evaluating documentation quality.

Forward [19] explored the creation of a documentation quality indicator

by combining metrics of an **A**rtefact's **U**sefulness, **R**eferential decay, and **A**uthority, or AURA. The author defined *usefulness* as a measure of a document's recency, frequency, and feedback, which requires manually retrieving user feedback over long-term monitoring for a document. *Referential decay* is a metric that uses a document's last modified date to calculate its relative inconsistency with the source code. Intuitively, an older document is more likely to be inconsistent compared to a newer document. Finally, Forward [19] uses Kleinberg's authoritative ranking technique [33] as a measure of a document's authority, which indicates how important this document is compared to other documents with similar information (e.g., external blogs). Although these metrics are potentially useful, AURA is generalized to all types of documentation, instead of focusing on the documentation that developers use to learn about a library. This means that AURA potentially misses documentation aspects that are important to software developers trying to use a library and captures documentation aspects that may be irrelevant. AURA also requires long-term data collection in order to calculate its metrics.

Aversano et al. [3] focused on evaluating the quality of documentation of open-source software. Their intent was to create a system that can be adopted by the open-source community in order to support high-quality documentation. In their work, they define multiple aspects of documentation quality, such as completeness, alignment (which is up-to-dateness), readability, and ease of use. Many of these aspects were also identified by previous researchers, as shown in Section 2.2. While we do not go through all of their created metrics, some of their formulas include:

1. $Completeness = \frac{Total\ classes\ and\ packages\ described\ in\ doc}{Total\ classes\ and\ packages\ in\ source\ code}$
2. $Up\ to\ date = Yes|No$
3. $Readability = Flesch\ Readability\ Index$

Although Aversano et al. [3] do define automatically measurable metrics for aspects of documentation, they do not evaluate whether these formulas adequately represent documentation quality for their intended metrics. Addi-

tionally, they only propose the metrics but do not implement a tool to automatically calculate these metrics for a given library and leave it to interested developers to calculate these values for themselves.

Other attempts at evaluating parts of software documentation include Zhong and Su's [73] DocRef tool and Lee et al.'s [35] FreshDoc tool, both of which attempt to find discrepancies between a library's documentation and source code. They did this by combining natural language tools with code analysis techniques to find documentation errors and report the inconsistencies; in essence, they check the correctness of the documentation. Wingkvist et al. [71] used the Goal-Question-Metric paradigm (GQM) [60] to derive the idea of using test coverage and clone detection as metrics for documentation. They treat each document as its own "class" and, through clone detection techniques, check for "clones" that represent redundancy in the documentation. Their intuition is that high-quality documentation has a high degree of uniqueness and low degree of similarity between the documents. They also used test coverage to determine how much of the "source code" (the documentation) has been checked. This was done by asking developers to complete a task with a library and then measuring the time taken to find the necessary information. "Test coverage" is then a fraction of documentation read when looking for the relevant information (i.e., documentation coverage). Having both positive test results and high test coverage then gives an indication of "good" quality documentation, while having either individually does not, i.e., positive test results with low coverage means that only a fraction of the documentation is actually relevant. The authors conclude that the combination of clone detection and test coverage is a representation of documentation quality. However, Wingkvist et al.'s [71] approach relies on developer input, as each evaluated library documentation would need developers to complete tasks to measure "test coverage", which is not automatable.

Although all these past efforts have been made to evaluate documentation quality, our work is different in that we provide (1) a comprehensive summary of documentation quality that addresses multiple important documentation aspects, (2) an automatic service that evaluates documentation quality, and

(3) validated metrics through manually constructed ground truths, developer interviews, and survey participants.

Chapter 3

Selecting Metrics to Use for Evaluating Documentation Quality

In this chapter, we discuss how we select the metrics we use to evaluate documentation quality. We first review the documentation aspects combined from the different documentation areas discussed in Chapter 2 and select which aspects to measure. We then discuss the process of creating or selecting the metrics we use to measure each selected documentation aspect and how to visualize them. Note that we do not discuss the concrete implementation of each metric here, but rather in Chapter 5.

3.1 Selecting Documentation Aspects

Given the number of aspects shown in Table 2.1 and the fact that using GQM may result in more than one metric per goal, it is infeasible to create metrics for all possible aspects. Additionally, presenting quantitative data about all these aspects to client developers is likely overwhelming. Instead, we decide to focus on a subset of important aspects by performing the following assessment.

First, we look at the number of supporting literature references of an aspect. Since many of these references base their findings on developer input, aspects mentioned repeatedly in those references are more likely important to developers in assessing documentation. For example, the "Organization" aspect has relatively many supporting references, suggesting that it has more

importance than the "Graphical Support" aspect. However, even though "Organization" has many supporting references, it is difficult to objectively *and automatically* determine whether documentation is properly organized as there is not a standard for what is considered "organized". Therefore, we also consider whether aspects are "potentially measurable" and include that as a criteria for aspect selection. We define *measurable aspects* as aspects with attributes that can be objectively and automatically calculated.

Even though we are now using a combination of number of supporting references and a definition of measurable aspects, we still need to decide whether all measurable aspects are worth measuring. Rather than use an arbitrary threshold for the number of supporting references, we instead utilize the taxonomy of documentation issues created by Aghajani et al. [1], [2] and the attribute list from Forward and Lethbridge [20] as a starting point to determine which aspects to focus on. The documentation issues and attribute list from these two efforts help us narrow our focus on documentation quality aspects that developers find hindering their learning of a library. This corresponds with our initial goal of assisting developers to select libraries that would be easier to learn and incorporate into their own project. Accordingly, we select the following measurable aspects: Completeness, Readability, and Up-to-dateness. From this starting point, we then look at the other documentation aspects from the literature, summarized in Table 2.1, and additionally select the most frequent measurable aspects supported by the literature, which are Code examples and Ease of Use. Thus, our final selection of documentation aspects to measure are: (1) Up-to-date, (2) Completeness, (3) Readability, (4) Code examples, and (5) Ease of Use.

3.2 Deriving Metrics for Selected Documentation Aspects

In this section we describe our process, following GQM [60], in creating the metrics for each selected documentation aspect. The GQM paradigm [60] is an approach that starts from a high-level "goal", where the idea is to ask "ques-

Table 3.1: Summarized GQM for the five aspects we focus on

Goal	Question(s)	Metric
Up-to-date	Is the documentation consistent with the source code?	Ratio of matching public classes/methods in the documentation to public classes/methods in the source code
Completeness	- Are the methods/classes found in the documentation in the source code? - What are the documented library tasks and/or features ?	- Ratio of matching public classes/methods in the documentation to public classes/methods in the source code - List of documented library tasks
Readability	- Is the documentation text readable? - Are the code examples readable?	- Flesch readability score - Code readability metric from previous literature [5], [56], [57]
Code examples	Do all public API methods/classes have code examples?	Ratio of public methods appearing in documentation code examples to public methods in the API
Ease of use	How easy is it to navigate the documentation?	Checklist found from HCI research [11], [43], [69]

tions" about the goal, and ends with "metrics" that answer the "questions" and measure the "goal". For example, one documentation aspect, "Code examples", is about the existence of code examples in the documentation, which we can turn into a "goal", i.e., the goal is to have code examples in the documentation. Then, we move on to asking "questions", where the purpose of the "questions" is to explore and define objects that represent the goal. Therefore, if the goal is having code examples in the documentation, example questions a developer may ask would be: "Are there code examples available?", "How useful are the code examples?", "What are the code examples about?", etc. Finally, "metrics" can be created to measure the answers for these questions, e.g., measure how many code examples exist in the documentation for a method or class of the library. Table 3.1 summarizes our GQM [60] process for the five selected aspects. Note that some metrics can measure multiple goals (aspects), while multiple metrics may be needed to measure one goal (aspect).

3.2.1 Up-to-date and Completeness

We devise two metrics that measure both the"up-to-date" and "completeness" aspects. The first metric measures the ratio of matching public methods between the documentation and source code, and the second metric measures the

ratio of matching public classes between the documentation and source code. Intuitively, a higher ratio for both of these metrics means a higher similarity between the methods and classes in the documentation and the public API methods of the source code. This addresses both the "up-to-dateness" and "completeness" aspect as out of date and/or incomplete documentation would have a low ratio of matching API method signatures and/or a low ratio of matching class definitions.

We also derive a second metric for the "completeness" aspect of documentation as a list of the "documented library tasks". This metric finds all the "library tasks" in a documentation and also whether the task has an accompanying code example. We define a "library task" as *any usage of a library, including its API, installation instructions, and interaction with other software.* This metric is supported by Uddin and Robillard [67] where they mention that one of the documentation issues regarding content (in)completeness is the lack of a description for a topic or task that can be done with the API, i.e., the documentation is incomplete in the sense that developers struggle to figure out how to use the API for a particular task. This sentiment is also supported by another work by Robillard [53], as well as Garousi et al. [21], and Zhi et al. [72].

One example task would be "Return sequence of PoS-tagged words" from a Natural Language Processing (NLP) library. To NLP developers, Parts of Speech (PoS) tagging is a basic and important task when programming in the NLP domain. PoS tagging allows grammatical analysis on text corpora that would provide insight on the grammatical patterns of that data. If a prospective NLP library did not have this task described in the documentation, then it would be harder for a developer to implement this needed task, or even know if this task was supported by the library. Thus, knowing documented supported tasks is important.

3.2.2 Readability

We split the readability of the documentation into two metrics: the first being the readability of the documentation text and the second being the readability

of the code examples. Even if the readability of the text is high, if an accompanied code example is unreadable, then it would be difficult to match the textual information with the concrete implementation. Additionally, a code example would be confusing if the intent and purpose of a code example is not properly described by a readable textual description. Thus, good documentation quality will have both readable text and readable code examples.

When deciding how to measure text readability, we again consult the literature [3], [19], [25], [58] and find that the Flesch readability metrics are commonly used. For the purpose of this thesis, we chose to use the Flesch reading ease formula [17] as it is designed solely to rate the readability of a text, as opposed to the Flesch-Kincaid grade level formula [16] designed to assign a United States grade level. Although other readability metrics are available, we found several issues with using them. For example, although the Coh-Metrix [24] or L2 Syntactic Complexity Analyzer (L2SCA) [38] are arguably better measures of readability, both of these metrics require external information outside of the text itself to properly use the metric. The Coh-Metrix requires input such as a Latent Semantic Analysis [18] (LSA) space, which is a representation of words and their text meaning. This LSA [18] space needs to be created, which requires expert opinion for accurate annotations. The L2SCA [38] metric also requires expert opinion as it requires a mapping of "production units" (which require their own definition) and syntactic structures, which need to be created. As such, we select the Flesch reading ease metric, as it relies solely on the text itself and no other external information requiring expert opinion.

When consulting the literature on code readability, we found a set of features proposed by Buse and Weimer [6] which can be used to evaluate code readability. Some of these features include line length (number of characters), number of identifiers, identifier length, number of parentheses, and number of periods. Following Buse and Weimer's [6] research, Scalabrino et al. [56], [57] further refined these features by analyzing the correlation between these factors and code readability. They then trained a machine learning classifier that predicts the readability of Java code. We utilize the results from Scalabrino

et al. [56], [57] and use their Java code classifier to measure code readability.

3.2.3 Code Examples

Code examples are also an important aspect of documentation. According to a survey conducted by Forward and Lethbridge [20], the use of examples within documentation was one of the most important "attributes" that contributed to the overall "effectiveness" of documentation. Garousi et al. [21] also conducted their own survey and found the use of examples to be one of the most impactful "attributes" when evaluating the overall quality of a document. This sentiment of the importance of code examples is echoed with the other references in Table 2.1. In order to achieve the goal of gauging the code examples in the documentation, the question we asked for GQM was: "Do all public API methods/classes have code examples?" With the aspect being framed by this question, we design the metric for this aspect to be a ratio of how many public methods and classes in the documentation have code examples to all public methods and classes available through a library's API. The higher these ratios, the higher the likelihood that there is an example to help guide developers to use the library's API.

3.2.4 Ease of Use

Lastly, the final documentation aspect we focus on is the ease of use aspect. This documentation aspect focuses on how "easy" is it to use the documentation in terms of navigating and finding information for a task. To answer the question of "how easy is the documentation to navigate?", we turn to Human-Computer interaction (HCI) research. This field of research focuses on the design of computer technology and how humans interact with the interface of computer design. In the context of this thesis, it would be how humans interact with the online website hosting the official documentation. As such, we create a checklist aggregated from HCI research [11], [43], [69] surrounding website design. These guidelines include checking for relevant links to relevant web pages, providing a table of contents, providing a search function within the documentation itself, and providing a list of links to all other pages on the

Table 3.2: Summary of documentation aspects and derived metrics

Aspect	Metric(s)
Up-to-date	- Ratio of matching public methods between documentation and source code - Ratio of matching public classes between documentation and source code
Completeness	- Ratio of matching public methods between documentation and source code - Ratio of matching public classes between documentation and source code - Documented library task list
Readability	- Flesch reading ease
Code Examples	- Ratio of public methods and classes having code examples in the documentation over all public methods and classes in the source code
Ease of Use	- HCI check list - Quick start check (described in Section 4.6)

documentation's home page. The more a documentation matches the guidelines of HCI research, the "easier" it is to navigate the documentation to find information.

3.2.5 Metric Summary

We present Figure 3.2 as a summarization of the different metrics we derive for each of the selected documentation aspects described in detail in the above sections. The "Quick start" check is an additional check we incorporate when measuring the navigability of a documentation, which was added from feedback given during the interview study discussed in Chapter 4.

Chapter 4

Initial Validation of Selected Metrics through Developer Interviews

After defining metrics that provide information about documentation quality, we need to validate whether our metrics provide a measure for the intended aspect, whether the metric is useful to developers when evaluating a library's documentation, and whether our summary presents them meaningfully. More formally, we want to answer:

1. Are our metrics useful in evaluating documentation quality?
2. Do our metrics accurately reflect the documentation aspects they intend to measure?
3. Is our current presentation of each metric easy to read and understand?

Answering these three questions gives us intermediate feedback before fully implementing the metrics and documentation overview summary and performing large-scale evaluation. To answer these three questions, we resort to external validation and conduct interviews with industry developers, asking about the metrics we defined and their presentation.

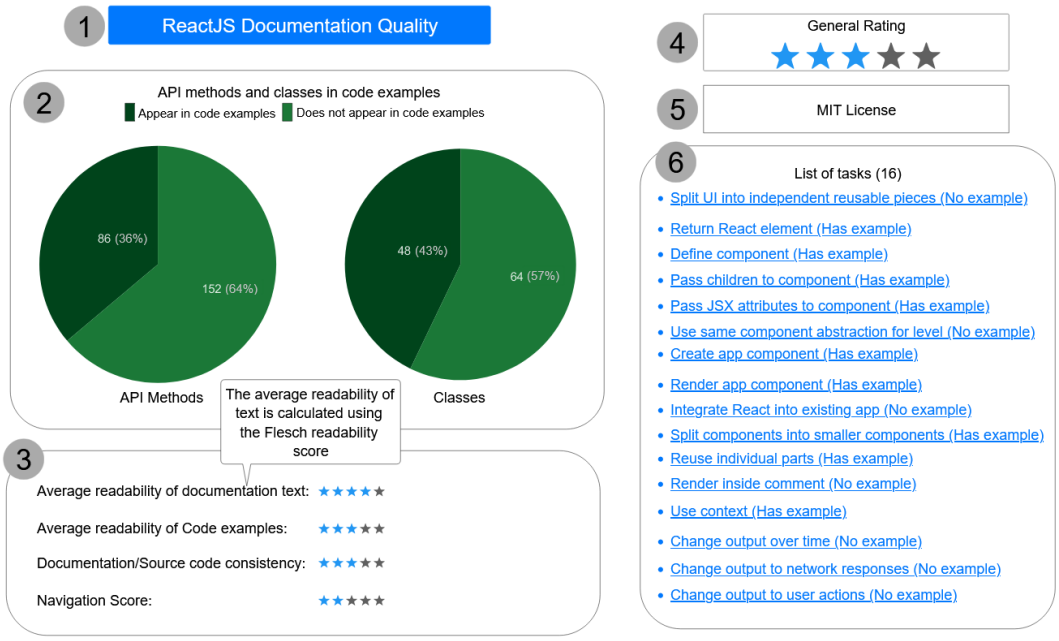

Figure 4.1: Initial mockup of documentation quality overview. We validate this mockup in our interview study.

4.1 Mockup of Documentation Quality Overview Visualization

Our end goal is to create a tool to automatically evaluate documentation quality and provide a visual summary to client developers. This visual summary would present the values of the metrics we calculate in an easy-to-read format. However, before building this visual summary, we want to get early feedback from developers to ensure that we create a visualization that is clear to them and that presents useful information, in addition to validating our metrics we derived in Chapter 3. Thus, we first create a mockup for the metrics we discussed but without all of them being fully implemented yet. We show this mockup as Figure 4.1 and use it in the interview study to receive feedback from developers before fully implementing the metrics and documentation quality overview summary (described below). The thesis author approximated the values of the metrics for the mockup by manually calculating the metrics for one page of the ReactJS documentation [13].

Our initial summary tool has six different components, labelled from ① to

⑥. Although components ① and ⑤ are not any of our metrics, we include them to provide background information about the library, that being its name and license. Component ② is a visualization of the individual metrics of measuring the number of public API methods and public API classes that have a code example provided in the documentation. Component ③ displays multiple metrics. The "Average readability of x" refers to the rating of either the documentation text readability or the code example readability. We create the star ratings for these two metrics by treating the returned values from the respective metrics as percentages out of 100, as both the Flesch reading ease [17], and Scalabrino et al. [56], [57] measurements return a value from 0 to 100. Component ③ also includes the two ratios of matching methods/classes in the documentation with the source code combined into one metric labelled "Documentation/Source code consistency". We create the star ratings for this metric by normalizing the ratio over 100, allowing us to treat the value as a percentage. Therefore, each star represents 20% consistency between the documentation and source code. We do not use the term "up-to-date" in our visualization because according to Zhong and Su [73], Lee et al. [35], and Zhou et al. [75], "up-to-date" implies that textual descriptions and examples (if any) are also "up-to-date" in the documentation, which we do not measure. The last metric presented in Component ③ is the "Navigation Score", where we create the star rating based on the number of features the documentation includes from the checklist we created from HCI research. The navigation score star ratings are not based on a percentage of the checklist as, according to the World Wide Web Consortium's definition of a navigable website [69], having at least two features constitutes a navigable website. Therefore, documentation having more than two navigable features equates to a five-star rating, while having exactly two features equates to a three-star rating, one feature equates to a one-star rating, and no features having a zero-star rating. Component ⑥ is the "List of tasks" which details what tasks can be done with the library, as extracted from the documentation. It also provides information on whether each task has an accompanying code example in the documentation between parentheses. Finally, component ④ is a "general rating" of the documentation

Table 4.1: Demographic information of interview participants

Participant	Job Title	Role	Years of Experience
P1	Jr. R&D Engineer	Software Engineering, Software Development, Dev Ops	4
P2	Jr. Developer	Research and development, Front-end web development	2
P3	Sr. R&D Engineer	System Administration, Back-end coding, Databases	12

quality, where the star rating is an equal-weighted combination of the previous metrics.

4.2 Interview Study Setup

This section details the interview setup, including the participant recruitment and the creation of interview questions.

4.2.1 Participant Recruitment

Our participants are professional software developers working in an industrial research company, using mainly JavaScript, whose goal is to create real products out of research prototypes and to perform applied research in software. After receiving ethics clearance from our university, we interview three developers of varying seniority and experience from this company. Table 4.1 describes the demographics of the three participants.

4.2.2 Interview Setup

We conduct the interviews online through Zoom meetings. We follow a semi-structured interview setup [37] where we use pre-determined guiding questions but allow the discussion to deviate and explore tangent directions. However, we try to ensure that interviewees answer our guiding questions such that we can validate our chosen documentation aspects and metrics.

We structure the interview and guiding questions as follows. We first ask participants about their background and role in the company. We then ask them open-ended questions about what they perceive as positive and negative

features in library documentation (i.e., what do they like seeing in documentation and what do they dislike seeing). The responses from this part help us understand what developers use when evaluating documentation. It also provides insights on whether our summary addresses these features and if we are generally measuring relevant aspects. The list of open-ended questions we ask are:

1. What are things you look for when viewing documentation for the first time?
2. What do you want to see in documentation?
3. What are things that lower documentation quality?
4. What would be useful to know about the documentation of a library before exploring it deeper?

Finally, we show participants the documentation quality mockup, Figure 4.1, and ask for their feedback on the provided metrics, as well as how they are presented. For each component we present in the mockup, we ask developers whether (1) the metric adequately represents its intended aspect and (2) whether that aspect is beneficial in evaluating documentation quality. When a participant does not understand the goal or intention of a metric, we take note and explain it to them before asking them again to answer these two questions. Each interview lasted approximately 30 minutes and we recorded the audio, with participant consent and ethics clearance, for transcription purposes and review.

4.3 Interview Analysis Methods

In this section, we describe the methods we use to analyze the interview responses. We start by transcribing each interview before analyzing the responses through an open coding mechanism [14], [63]. For each transcript, we extract responses from each question and group related responses before analyzing each group for the documentation aspect or metric they address.

4.3.1 Open Coding

After extracting all responses from the transcripts, we perform open coding to group similar responses through a series of coding stages [7], [55]. First, the author of this thesis reviews each of the interview transcripts and extracts all the responses. This follows the same method as Campbell et al. [7] to define the "units of meaning" which is described to represent one item that can be analyzed for coding data. Second, the same author performs open coding in order to group responses based on the content of the response. Third, the author assigns a label to each group and creates "higher-level" categories to group multiple labels. We refer to these "higher-level" groups as "themes" and the "lower-level" labels as "codes". After the thesis author creates the themes and codes, the thesis author and another research collaborator independently annotate each of the responses with both a theme and a code and we measure inter-rater agreement. The inter-rater agreement between the two authors, measured by the Cohen's Kappa-score [10], is 0.66 for themes and 0.58 for codes, which is a substantial agreement for themes and moderate agreement for codes [10]. These themes and codes help us summarize the support and criticisms of the current metrics, the mockup presentation, and in identifying missing metrics/documentation aspects that developers find useful when evaluating documentation quality. We discuss our results below.

4.4 Interview Results: General Documentation Questions

Here we discuss the results of our interview, following the open coding of the responses from the interviewees. Our analysis will follow the flow of the interviews, question by question.

4.4.1 Open-Ended Questions

We first discuss the responses to our open-ended questions about software documentation.

What are things you look for when viewing documentation for the first time? According to all our participants, we find that libraries are explored when there is a specific task that needs to be solved. Additionally, having a general high-level idea of the library, or what tasks the library supports, is beneficial when deciding whether or not to use a library. This coincides with our metric of finding a list of documented library tasks the library supports. Participants P2 and P3 also notice the general cleanliness of the documentation's user interface, e.g., if the documentation looks clean and aesthetically pleasing, they will base their expectations from that first glance.

What do you want to see in documentation? All three participants want to know how to use a library without much reading. Therefore, all three participants agree that positive features in documentation include the presence of a "quick-start" guide that includes installation or importing instructions and code examples. The benefit of code examples help the developer understand how the library should be used without much reading, and the benefit of quick-start guides and installation examples serve to aid the developer to start working with the library. Quick-start guides also provide basic information without too much navigation into the documentation.

What are things that lower documentation quality? P1 and P3 mention that negative features of documentation include having long and complicated explanations, either in textual descriptions or code examples. Concrete examples of this are large blocks of text and long code examples. Intuitively, the long physical length of documentation would make information difficult to find in addition to being difficult to read and comprehend. Generally, all participants agree that when information is hard to find or outdated, it causes negative feelings and a loss of trust toward the library.

What would be useful to know about the documentation of a library before exploring it deeper? We ask this question to determine if there is any useful information for a developer to know about the documentation before starting to read it. However, most of the responses by the interviewees focus on information regarding the library itself, rather than about its documentation. For example, P1 and P2 both mention knowing a library's popularity, or use in

other projects, would be beneficial to know about a library before investing time and effort into it. Additionally, P3 answers that the license of a library is beneficial, as a conflicting license means no effort needs to be spent on this library. The decision of whether to use a library or not can simply be based on the license alone. Although not following our goal for this interview, this information is still valuable in learning what is beneficial to developers when evaluating a library, and other works already have such metrics for library comparison [26], [39]–[41].

4.5 Interview Results: Mockup Feedback

We now discuss the feedback regarding our mockup (Figure 4.1) of a documentation quality overview summary. We again follow the interview flow of questions, starting with general feedback about the mockup, before discussing the feedback for each of the metrics in the same order as labelled in Figure 4.1.

4.5.1 General Mockup Feedback

Generally, all participants find that our mockup is clean and appreciate the current information present as an overall report on a library's documentation. That being said, there are several suggested areas of improvement. One item of feedback is to include more contrasting colours for clarity of reading the numbers on the pie charts (component ②). Additionally, some of the space could draw more attention to what is more important; e.g., component ②, though important, takes too much space on the page. Instead, that space could have been used to make the "List of Tasks" (component ⑥) more prevalent (which all participants find important). As such, participants suggest changing the structure of the different components, i.e., placing more important information (e.g., list of tasks, general rating) on the left side and moving the metrics to the right side. The current labels of the different metrics are also suggested to be more succinct, while at the same time clearly convey what the metric is trying to measure, e.g., "Average readability of documentation text" is described to be *"...almost like a sentence"* by P1, when it could be more succinct

as "Readability of text". Participants also express interest in additional information, such as the current maintainer of the library, providing the logo of the library (if available), and having a high-level summary of the library.

4.5.2 Examples of API Methods/Classes

This feedback relates to component ② of our mockup and aims to determine whether developers view code examples of the library's API methods and classes as important. The answer to this question is a unanimous "yes", with P3 responding, *"'...hopefully I would like to have 100 percent every time..."*. However, in terms of the presentation, participants find that the pie charts are unnecessary given that there are only two numbers presented for each metric. As such, a suggestion is to remove the graph and add a single line for each metric within component ③.

4.5.3 Documentation & Code Example Readability

This feedback relates to component ③ and ④ of our mockup. All participants respond that readability of text and code examples are important. The readability of the documentation text is important because developers do not want to waste time or effort understanding complex explanations. They prefer short, concise sentences that convey the intent of what is being described so developers can focus more on the code examples provided to learn how to use that API for their own use. Additionally, the readability of code examples is also important. Since developers gravitate toward code examples to quickly understand how to use the library, its clarity and understandability is necessary for developers to feel comfortable with the library enough to experiment with it. To facilitate this, P2 and P3 express the desire of having runnable code examples within the documentation itself, e.g., a CodePen [9] example. This allows developers to experiment and explore a library without committing the time and effort of installing or importing the library into their own project. All participants also convey their satisfaction with the way we present the information in the mockup, P2 stating *"You give me one important information, because average readability of documentation is something I value a*

lot. ...[Y]ou gave me that is four out of five and you gave me information on how you calculate it. So it's perfect."

4.5.4 Source Code/Documentation Consistency

As a reminder, we use the term "consistency" instead of "up-to-date" because our metric only matches method signatures and class definitions between the source code and documentation. It does not determine if other information about the methods/classes, such as code examples and textual descriptions, are up-to-date. All participants agree that consistency is important because documentation builds "trust" between the developer and the library. When the documentation proves itself to be incorrect, that trust is eroded and developers are more inclined to ignore the documentation and focus on analyzing the source code, as P3 states, *"Let's say [I start with] 100% trust the documentation is up-to-date and correct, and if I ever find out that it is not the case, even for one method, then I tend to kind of ignore it and start digging into the code much more."* This supports our current metric of rating the consistency between the documentation and source code.

4.5.5 Navigation Score

Our intent with "Navigation Score" is to capture how easy it is to navigate the documentation to find wanted information. While the participants express interest in this metric, they were unable to think of a way to improve our current plan of the implementation of the metric. Instead, one interesting idea presented by participant P1 is to incorporate documentation within the IDE itself accompanying code suggestions. This ignores needing to navigate the documentation entirely by instead providing documentation along with the suggested method within the IDE.

4.5.6 Library Tasks

For this component, the participants respond that knowing the tasks a library supports is important. However, the current presentation can be improved,

e.g., there are too many tasks in the list and it is difficult to navigate. One suggestion is to group or categorize the tasks in order to display it in a more concise manner where a developer can see what tasks a library addresses at a glance. One important feedback to note is, although knowing the tasks of a library is important, it is equally important to know how to perform that task. For example, P3 states *"Code examples...I think is the thing that I love to see the most"*, and this is because *"...descriptions I tend to get kind of confused...code examples if you run with these parameters, you get that."* In other words, code examples provide a clearer understanding of the intentions of a library and how it accomplishes a particular task compared to textual descriptions. As such, code examples go hand in hand with knowing what tasks a library supports. This feedback supports our decision to additionally provide information about whether a task has an accompanied code example in the "Documented Library Task" list. However, we change this indicator to be more visual (e.g., with a green checkmark and red cross) as opposed to a textual indicator based on feedback.

4.5.7 Free-Form Answer and Tangent Discussions

Finally, the participants are able to share any insights regarding documentation and the mockup not previously addressed. Participant P1 acknowledges the importance of good naming of API methods and classes, mentioning that well-made, descriptive, and concise names for API methods and classes removes the need for much of the documentation. A descriptive method name allows a developer to understand the intent of the method and, accompanied by a code example, to understand how to use the method, removing the need for any textual documentation. Participant P2 highlights the importance of the idea of "consistency" between the documentation and the source code, suggesting that additional measurements can be included into the "consistency" metric. Keeping the trust between the developer and library requires correct documentation, however, and aside from bringing up its importance, no concrete metrics were suggested to facilitate measuring consistency beyond what was currently presented. Participant P3 had no additional comments.

4.6 Summary of Interview Insights and Concrete Changes

The interviews provide us intermediate feedback on whether "we are on the right track" and allow us to address any obvious issues before conducting more large-scale validation with the survey. We summarize the feedback and changes we considered based on it.

Are our metrics useful in evaluating documentation quality? From the interview responses, our metrics are useful in evaluating documentation quality. All of the responses convey that each metric we present in the summary is useful, with many responses expressing the benefit of that metric.

Do our metrics accurately reflect the documentation aspects they intend to measure? All interviewees find our metric calculations to be reasonable if not intuitive, and have no issue with the way our metrics are calculated. As a reminder, during the free-form answer discussion (Section 4.5.7), participant P2 expresses the desire for the incorporation of additional measurements into the "consistency" metric, but has no suggestions for any concrete metrics that could be added.

Is our current presentation of each metric easy to read and understand? Unfortunately, our presentation of the metrics is what most of the negative feedback is about. Regardless of the usefulness and accuracy of our metrics, our presentation of the metrics is confusing and requires some guidance to understand. The most common issue is the confusion of how to read certain metrics. We summarize our concrete changes with regards to our presentation below.

Figure 4.2 shows our updated visual summary after incorporating the interview feedback and is our full working version of the summary tool.

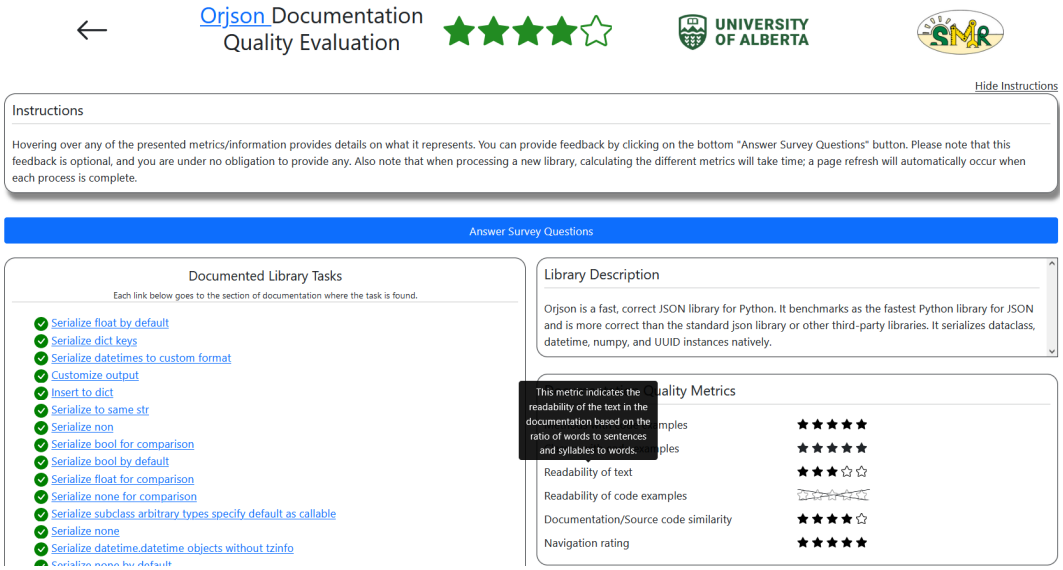

Figure 4.2: A screenshot of the final documentation summary tool

4.6.1 Content Changes

Since all three participants agree that a "quick-start" or installation guide is a positive feature of documentation, one of the additions to our tool is the check for such a guide, which we incorporate into the "Navigation Score" metric (described in Section 5.4). Additionally, we add a summary description of the library to add high-level information about the library. We also remove the license since, although important, it is more related to the library itself, whereas we want to focus on the library's documentation and the license can be incorporated into a library comparison overview, such as the one created by De la Mora and Nadi [41].

4.6.2 Presentation Changes

Based on the presentation feedback by the participants, we change the pie charts for "API methods and classes in code examples" to a single line each in Figure 4.2. We also change the labels to "Methods with examples" and "Classes with examples," respectively. These metrics still present the percentages of how many methods and classes have code examples in the documentation. However, they now take less room and do not overshadow the other information present. We create the star ratings for these two metrics by again

normalizing the ratio to percentages out of 100, where each star represents 20% of methods or classes with a code example in the documentation. Note that in Figure 4.2, they are covered by the tooltip explaining the "Readability of text" metric. We also change the structure of the summary by moving the "List of Tasks" (changed to "Documented Library Tasks") to the left side and the box containing the different metrics to the right side of the page. This change is made because of the participants' opinions escalating the importance of the task list in comparison to the other information. We also sort the "Documented Library Tasks" list by the frequency of the extracted task, alphabetically, and if the task has a code example. The textual description of whether a task has a code example is changed to a visual indicator with a legend. The general rating is kept to the right side of the summary page; however, we move it closer to the name of the library. We rename the readability metrics "Average readability of x" into: "Readability of text" and "Readability of code" for each readability rating, respectively. We change the label for "Documentation/Source code consistency" to "Documentation/Source code similarity", as we find that the initial responses of the summary tool state that the word "consistency" is confusing. We change the label "Navigation Score" to "Navigation rating", as the word "score" has a connotation of being absolute, while a "rating" is more relative (in this case, to other library documentation).

Chapter 5

Implementation of Documentation Quality Summary

After receiving feedback from the interviews, we proceed to implement the final versions of the different metrics, as well as implement the automatic calculation of metrics and creating the summary page for any given library.

5.1 Documented Library Task List & Linked Examples

In this section, we explain how we extract the "Documented Library Tasks" list in Figure 4.2 which corresponds to the "completeness" documentation aspect. This component presents extracted tasks from documentation and links code examples to those tasks.

We extract tasks using an existing technique by Treude et al. [66] and link code examples by using heuristics that analyze the HTML structure of the documentation. In other words, there are two stages to extracting the information in this component. The first is to extract tasks from the documentation, designated as the *task extraction* phase, and the second is to find and link code examples to those tasks, designated as the *task linking* phase.

5.1.1 Task Extraction Implementation

To extract tasks from the documentation, we utilize the task extraction tool created by Treude et al. [66], called TaskNav (short for TaskNavigator). TaskNav assists developers in navigating documentation where it provides a list of tasks, extracted from documentation, in a search engine. For example, if a developer wants to know the supported tasks surrounding PoS tagging by an NLP library , they would search "tag" (or some form of the word) in TaskNav, and it would return a list of tasks, such as "Run part of speech tagging" or "Remove XML tags". The intuition behind TaskNav is based on analyzing the grammatical structure of a sentence to see if there is a potential task phrase. It does this by extracting phrases from a sentence that include a programming verb and follow a predefined task definition. A list of *programming verbs* was created by Treude et al. [66] through their extensive testing, and we provide that list both on our artifact page [47], as well as in Appendix B.1. They define a *task definition* as: *"...[a] verb involved in a dependency with an object or with a prepositional phrase (or both)..."*. Using the previous example, the words "Run" and "Remove" are approved programming verbs in the verb list, and their respective sentence structures follow the task definition.

For each page of a documentation, we follow Treude et al. [66]'s preprocessing procedure where we remove all HTML tags from the page, except for HTML header tags (`h[1-6]`), and replace inline `<code>` tags with `<tt>` to allow inline code to be treated as nouns during the task extraction process. Note that this replacement is only for the task extraction phase, while the task linking phase (described next) uses the original HTML.

After preprocessing a documentation page, we feed it into TaskNav [66][1], one paragraph at a time, to extract tasks. TaskNav treats each sentence of a paragraph independently and uses the Stanford NLP toolkit [48] to find the grammatical dependencies in each sentence to extract the tasks. Let us take the following paragraph shown as "p-1" in Listing 5.1 as an example: *"If you want to run a series of TokensRegex rules before entity building, you can also*

[1]Note that the authors have kindly provided us with a executable version of TaskNav.

specify a set of TokensRegex rules. A TokensRegexAnnotator sub-annotator will be called. It has the name ner.additional.tokensregex." [2] TaskNavigator splits this paragraph into three sentences and extracts the following tasks from them:

- run series before entity building
- run series of TokensRegex rules
- specify set of TokensRegex rules
- call TokensRegexAnnotator sub-annotator

Here, we can see that there are three tasks extracted from the first sentence, because the words "run", and "specify" are considered verbs, and there is a dependency from these words to the objects "entity building", and "TokensRegex rules". The second sentence has the verb "call" and the object "TokensRegexAnnotator sub-annotator", which leads to one extracted task, and the third sentence does not have any extracted tasks.

5.1.2 Code Example Linking Implementation

After extracting tasks from each paragraph in each page, we then link code examples found on the same page to those paragraphs. We use a heuristic to help us link code examples to tasks. The heuristic only considers paragraphs that are (1) above the code example and (2) do not have a header in between the paragraph and code example. Overall, for each code example denoted by the `<pre>` tag, we traverse the HTML DOM tree to find the nearest paragraph following the heuristic and link it to the closest paragraph that has extracted tasks. Listing 5.1 shows four paragraphs (p0-p3), one code block (pre-0), and one header (h-0). In this example, we find the code block pre-0 and attempt to link this code example to a paragraph. According to our heuristics, we ignore paragraphs p-0 and p-3. Since p-0 has header h-0 between it and the code example and paragraph p-3 is below the example, this leaves paragraphs

[2] https://stanfordnlp.github.io/CoreNLP/ner.html

p-1 and p-2 as candidate paragraphs. Of these two paragraphs, we now check whether either of them contain any extracted tasks. As shown in Section 5.1.1, paragraph p-1 has four extracted tasks, while paragraph p-2 does not have any extracted tasks. Therefore, we link code example pre-0 to paragraph p-1. If both paragraphs p-1 and p-2 have code examples, then we would have selected the closest paragraph to the example, i.e., paragraph p-2. After we find the paragraph with extracted tasks nearest to the code example, we link and store the extracted tasks of that paragraph with that code example for later presentation in the "Documented Library Tasks" component of the summary page.

5.1.3 Task Extraction and Linking Verification

In order to verify the correctness of our task extraction and linking process, we create a ground truth consisting of eight web pages from four different domains and three different programming languages. We use the orjson [28] and JSON-java [62] library GitHub pages, the Stanford NLP "Named Entity Recognition" [42] and "'Command Line Usage" [12] pages, the NLTK "parse" [45] and "tag" [46] pages, the jQuery "get" [32] page, and the React "Component and Props" [13] page. We first describe the ground truth creation and verification for the task extraction, then detail the ground truth creation and verification for the task linking. We provide the list of rules and guidelines used to make the ground truths in Appendix A.

Task Extraction

To create the task extraction ground truth, the thesis author and another research collaborator manually extract 354 tasks by looking at each paragraph for each documentation page and finding tasks related to the usage of the library. As a reminder, we define a "library task" as *any usage of the library, including its API, installation instructions, and interaction with other software.* For example, in this paragraph extracted from orjson [28]: *"orjson supports CPython 3.7, 3.8, 3.9, and 3.10. It distributes x86_64/amd64 and aarch64/armv8 wheels for Linux and macOS. It distributes x86_64/amd64*

Listing 5.1: Example paragraph and example from StanfordNLP NER documentation

```
<p> {% p-0 %}
  By default no additional rules are run, so leaving
      ↪ ner.additional.regexner.mapping blank will cause this
      ↪ phase to not be run at all.
</p>
<h3 ...> {% h-0 %}
  Additional TokensRegex Rules
</h3>
<p> {% p-1 %}
  If you want to run a series of TokensRegex rules before entity
      ↪  building, you can also specify a set of TokensRegex
      ↪ rules. A TokensRegexAnnotator sub-annotator will be
      ↪ called. It has the name ner.additional.tokensregex.
</p>
<p> {% p-2 %}
  Example command:
</p>
<div ...>
  <div ...>
    <pre ...> {% pre-0 %}
      java edu.stanford.nlp.pipeline.Stanford...
    </pre>
  </div>
</div>
<p> {% p-3 %}
  You can learn more ...
</p>
```

wheels for Windows. orjson does not support PyPy. Releases follow semantic versioning and serializing a new object type without an opt-in flag is considered a breaking change." [3], we would extract the library task:

- Serialize a new object

After the two annotators extract the library tasks, they discuss any discrepancies and resolve agreements. Since tasks are extracted from the paragraph and are not predefined categories/labels, we cannot use Cohen's Kappa [10] to measure agreements. Instead, we provide a percentage of the agreements

[3] https://github.com/ijl/orjson

between the two annotators where this ratio is the number of same extracted library tasks on a document page, e.g., an agreement of 60% means the authors extracted the same task(s) for 60% of the paragraphs on a document page. The percentage agreement of library tasks over the eight ground truth pages ranges from 21% to 83%, with an average agreement of 69%. Note that the agreement percentage here is whether or not the two annotators extract the exact same tasks from a paragraph. This means if annotator 1 extracts three tasks from a paragraph and annotator 2 extracts four tasks from the same paragraph, we mark that paragraph as a disagreement, even though all but one task is different. The reason we choose to gauge agreement percentage based off paragraphs and not the tasks themselves is because we can be certain of the number of paragraphs as our denominator, while the number of tasks is subjective and harder for future researchers to replicate.

We then automatically extract tasks using our described technique in Section 5.1.1, which extracts 592 tasks from the eight different pages containing 1,681 paragraphs (based on the HTML `<p>` tag). To verify the extracted tasks, we compare the automatically extracted tasks to ground truth tasks. However, the manually extracted tasks in the ground truth and the automatically extracted tasks of the technique do not always match exactly. For example, sometimes words are added or replaced, or even a different form of the word is used, e.g., "Run trained CRF's on each sentence" vs "Run series of trained CRF's". In order to combat this issue, we use fuzzy matching with a similarity threshold to determine if the tasks are equivalent. If this threshold is not met, then we mark the extracted task as incorrect. Using this ground truth, we calculate the precision as the number of matched extracted tasks with the ground truth over the number of extracted tasks and find that the precision of our task extraction technique for library tasks ranges from 3% to 20%, with an average precision of 13%. The recall of our task extraction is the ratio of matching extracted tasks to the number of ground truth tasks and ranges from 4% to 79% with an average recall of 47%.

We find multiple reasons that contribute to the false positives and false negatives of the extraction approach. In terms of false positives, an automati-

cally extracted task by TaskNav may not be a "library task", meaning that the extracted task does not meaningfully denote a task supported by the library, and only follows the grammatical task definition with an allowed "programming verb". For false negatives, some expected tasks in the ground truth did not have the needed verb in the allowed programming verb list. For example, from the "Named Entity Recognition" page of the CoreNLP documentation [4], we manually extract the task "Give tokens more specific tags". However, the general verb list from Treude et al. [66] (Appendix B.1) does not contain the verb "give", as this verb is not a common general programming verb, and if included, would cause too many false positives. In other cases, the program found a programming verb, but could not extract a task following the grammatical task definition by Treude et al. [66]. For example, another task we extract from the "Named Entity Recognition" page [5] is "tag numeric and time related sequences", from the sentence: *"Next a series of rule based systems are run to recognize and tag numeric sequences and time related sequences."*. However, the program will extract "tag time related sequences", because the word "numeric" in this sentence is considered an adjective and not part of the object. Even though the verb "tag" is part of the programming verb list, our ground truth tasks may not match the sentence structure that TaskNav uses. Additionally, false negatives can occur due to not reaching our threshold of fuzzy matching. We choose 95% as the threshold as lower thresholds result in many false positives.

To address the reasons for the false positives and negatives that contribute to the low precision and recall, we modify the verb list by Treude et al. [66] and include an exclusion list of verbs. By changing the original verb list, we limit the amount of false negatives that occur because a domain-specific verb is missing from the allowed verb list, and by utilizing an exclusion verb list we can filter verbs that raise too many false positives. The thesis author creates these lists through manual experimentation and reading of the documentation pages for the six libraries used in creating the ground truth. The number of

[4]https://stanfordnlp.github.io/CoreNLP/ner.html#fine-grained-ner
[5]https://stanfordnlp.github.io/CoreNLP/ner.html#numeric-sequences-and-sutime

documentation pages over the six libraries total to 148 documentation pages. To modify the original verb list (Appendix B.1) to the other verb lists (Appendices B.2, B.3, B.4), we compare the missing extracted tasks of the program to the ground truth tasks of libraries in the same domain and add commonly found verbs that are missing in the original verb list. For example, the additions we make to the NLP verb list B.2 are: supports, serialize, deserialize, decode, parse, validate, and specialize. To create the exclusion list for the NLP domain B.2.1, we again analyze a comparison between the program extracted tasks and the ground truth, and find tasks that were found by the program that are not tasks in the ground truth. After we apply the different verb and exclusion lists to the eight pages respectively, we raise our precision to an average of 40% and recall to an average of 62%.

Task Linking

Similarly, we create a second ground truth for "task linking", where we manually check whether a task has a code example or not. The thesis author and research collaborator again separately go through the same eight pages. For each code example in a document, they select which paragraph (if any) to link with the code example. This resulted in 227 linked code examples and paragraphs. Since this is a more objective task, we obtain near-perfect agreement on most of the documentation pages, with the only guiding instruction being that the paragraph must describe the code example in some way. The lowest agreement page was 64%, with the highest agreement being 100%.

Using this ground truth, we evaluate the performance of our task linking technique on these documentation pages. We define the precision of our task linking technique as the ratio of correct automatically linked code examples to paragraphs over all linked code examples, where our technique ranges from 0% to 82% with an average linking precision of 43%. We define the recall as the ratio of correct automatically linked code examples to the ground truth code example/paragraph links, where the recall ranges from 0% to 75% with an average recall of 33%.

One reason for false positives is that our automated technique links the

example to the closest paragraph **that contains tasks and is located above it**. If the descriptive paragraph is underneath the code example or the correct descriptive paragraph did not have extracted tasks, then our technique will instead link the example to the closest paragraph with tasks, which would be an incorrect description. False negatives occur when the program finds a code example, but is unable to link it with any paragraph. This is because none of the paragraphs between the code example and the nearest above (HTML tag) header have extracted tasks. Since our heuristic does not consider paragraphs above the closest header to an example or below the example, such a code example would not be linked to any paragraph.

Another issue with our task linking evaluation stems from manually creating the ground truth. Since the ground truth links paragraphs and code examples regardless of whether the paragraphs had extracted tasks, there is a chance our program would be unable to link code examples to the correct ground truth paragraph because that paragraph does not have any extracted tasks. This means that evaluating the task linking on the ground truth as is would be double penalizing the program, once for its performance on the task extraction, and again for being unable to correctly link code examples. Therefore, to isolate the task linking evaluation, we use a subset of the ground truth where all linked paragraphs have extracted tasks. This results in 107 linked code examples and paragraphs. Using this subset, the precision for the task linking ranges from 43% to 82% with an average precision of 55%, and the recall ranges from 12% to 75% with an average recall of 42%.

5.2 Documentation and Source Code Linking

We now discuss the three metrics that require comparing the documented APIs with the APIs offered in the actual source code: "Methods with examples", "Classes with examples", and "Documentation/Source code consistency".

Listing 5.2: Code example from StanfordNLP Simple API documentation[6]

```
Sentence sent = new Sentence("your text should go here");
sent.algorithms().headOfSpan(new Span(0, 2)); // Should return 1
```

5.2.1 Methods and Classes with Code Examples

To calculate "Methods with examples" and "Classes with examples" metrics, we need to correctly detect which of the library methods appear in code examples in the documentation. Thus, we begin by extracting all code examples designated by the `<pre>` HTML tag. We then use a heuristic to filter code examples without a method call by checking for at least one open round parenthesis ("`(`"). This is a simple heuristic used to support incomplete code examples and to support as many programming languages as possible. As a reminder, we currently aim to support Python, Java, and JavaScript libraries. If the code example just stated the method (e.g., in a comment), rather than show how to use the method, it is not useful as an example. Listing 5.2 shows a code example from the Stanford NLP [48] documentation [6]. Using our above heuristic, we extract the following method calls:

1. Sentence("your text should go here")
2. sent.algorithms()
3. headOfSpan(new Span(0, 2))
4. Span(0, 2)

After extracting the method calls from code examples in the documentation, we need to determine which of these method calls are library API methods. Thus, we next extract public methods from the library's source code. For each library, we clone its repository from GitHub and find its source code by finding either a "src" directory or a directory with the same name as the library; e.g., NLTK [44] stores its source code under the "nltk" directory in its GitHub repo [7]. Note that in the tool implementation, if we are unable to find

[6]https://stanfordnlp.github.io/CoreNLP/simple.html
[7]https://github.com/nltk/nltk

the source files of a directory, we let the user know on the summary page. For each source file in the repository, we use an Abstract Syntax Tree (AST) to represent each source code file. We then traverse the AST to find and extract all public methods and their relevant information, including the fully qualified method signature to determine its containing class. We use this information to create a dictionary of public method signatures and then match the method calls extracted from the documentation to the extracted source code method signatures.

We match methods from the source code to methods in the code examples by first matching the method name. If there are multiple function definitions with the same name, e.g., same method in different classes, then we compare the number of arguments in the example method call with the number of parameters in each of the candidate method signatures. If the number of arguments match the number of parameters, we count it as a match; otherwise, we do not. In the event there are multiple function definitions with the same name and parameters, we do not match the example method call with any of them. We choose not to match the method call with any of the potential definitions to err on the side of precision, rather than randomly select a definition to link. When we find a match, we consider that this documentation snippet mentions this API and thus has an example.

Note that when matching method calls and signatures, both Python and JavaScript have their own challenges. Python has "optional parameters" that modify the number of arguments a method accepts, meaning there is an upper and lower bound to the number of parameters for a Python method. JavaScript, on the other hand, does not check for the correct number of arguments as missing arguments are inherently treated as "undefined" and do not stop the execution of the method. To account for these differences, we modify the parameter count logic to use a range for Python and to solely check for matching method names for JavaScript.

For the class level, we consider that a class has code examples in the documentation if there is at least one method from this class (including constructors) that is in use in a code example in the documentation according to the

above matching steps.

Validating Method and Class Linking

To verify the correctness of our method/class linking process, we create a ground truth using the same libraries from section 5.1.3 with the exclusion of React [50] and jQuery [31], and the addition of Requests [51], QUnit [49], and jBinary [30]. We replace these libraries as the React and jQuery GitHub repositories do not store their source code in a "src" directory, or a directory sharing its name. This makes it difficult to verify as the generalized approach described above for this metric would need to make exceptions for these two libraries (in the form of custom file traversal algorithms).

Recall that our metric computes the proportion of source code library API methods that have examples. Thus, the correctness of this metric depends on the ability of our technique to detect library API methods in the documentation code examples. Accordingly, to create the ground truth for this metric, we first extract all code examples from the documentation. For each code example, we manually collect the list of library method calls in that example. We double check that this is indeed a library method call by comparing it to the source code of the library. Thus, our final ground truth for each library consists of the list of documentation code examples and their corresponding library methods. Given a code example, ideally, our automated extraction technique would correctly extract all methods in the ground truth and nothing beyond that.

We use the above ground truth to calculate the precision and recall of our method linking technique. We define the precision of our technique as the number of correct method calls the technique extracts divided by the total number of method calls the technique extracts for a given library. We define the recall of our technique as the proportion of ground truth methods the technique is able to correctly detect. We calculate recall and precision per documentation code example. To report overall numbers for a given library, we calculate the median recall and precision over all snippets in that library. Across the seven libraries we evaluate, the precision of our technique has a

median of 32% and the recall has a median of 40%.

A false positive in this context is when our technique says a code example contains a library method m, but m is neither part of the code nor a library method. One reason for false positives is if a third-party method call shares the same name as a method in the library. In this scenario, the program will match the extracted method call with a library method definition; however, the code example does not actually use that library method. The last issue that may cause false positives is incorrect method call extraction from the documentation. Since our technique uses regular expressions to find method calls and its arguments, there is the possibility our program finds an incorrect number of arguments which match a library method definition.

We define false negatives as a failing of the program to find expected library method calls in a code example. One reason for this is if the program fails to match the extracted method from the code example to the corresponding function definition extracted from the source code. This could be the case, similar to above, if the library is updated and the method is removed, or the number of parameters is changed in the source code but not the documentation. Another reason that contributes to this issue is the possibility that our program fails to properly parse the example. This means that the program fails to create an AST of the code example, where one reason would be incomplete code examples. The issue of incomplete code examples is one area of improvement we discuss in Chapter 8.2.

5.2.2 Documentation/Source Code Similarity

We now describe how we calculate the "Documentation/Source code similarity" metric, which finds how many methods and classes in the documentation are in the library source code. This differs from the previous section as we do not look for code examples; instead, we match the signatures of the classes/methods in the documentation **text** with the signatures of the classes/methods in the source code. As a reminder, this metric measures the "completeness" and "up-to-date" aspect, i.e., we want to measure how much of the signatures in the documentation are consistent with the source code

API signatures.

We first extract any inline code mentions from the documentation by parsing the HTML for `<code>` and `<dt>` HTML tags. The `<dt>` HTML tag stands for "description term" and is the other common HTML tag aside from the `<code>` tag that we found to designate inline code in online documentation. Once we extract the candidate signatures from the documentation, we then use the same previously used heuristic to check for at least one open parenthesis in extracted HTML elements. We use the heuristic as an initial check to ensure that the found HTML element has arguments or parameters that we can extract and use to match against the created source code dictionary. We extract the parameters and the corresponding class from the candidate signatures by using regular expressions. We then try to match the candidate signatures by its method name to the source code dictionary before also attempting to match the corresponding class and parameter identifiers. At the end, we calculate the metric as the percentage of documentation method calls that are part of the library.

Validating Documentation/Source code similarity

The validation for the "Documentation/Source code similarity" metric is similar to the validation described earlier in Section 5.2.1. However, instead of the technique detecting library API methods in documentation code examples, our technique detects library API methods in the inline code snippets of documentation text. To create the ground truth for this metric, we first extract all inline code from the documentation text. For each piece of inline code, we manually find the matching library method call(s). We perform the same check of determining whether the library method calls are indeed part of the library by comparing it to the source code. Thus, our final ground truth for each library consists of the list of inline code snippets and their corresponding library methods. Given a code snippet, ideally, our automated extraction technique would correctly extract all methods in the ground truth.

We use this ground truth to calculate the precision and recall of our method linking technique for inline code snippets. We define the precision of our

technique as the number of correct method calls extracted by the technique divided by the total number of method calls extracted by the technique. We define the recall of our technique as the proportion of ground truth methods the technique was able to correctly detect. We calculate recall and precision per documentation code snippet. To report overall numbers for a given library, we calculate the median recall and precision over all snippets in that library. Across the same seven libraries we evaluate on, the precision of our technique ranges from 18% to 39%, with a median precision of 28%. The recall ranges from 15% to 24%, with a median recall of 20%.

When we analyze the reasons for the low precision and recall, we find multiple issues that caused false positives and negatives. We define false positives as our technique saying a code snippet contains a library method m, but m is not part of the code snippet and not a library method. One issue that contributes to false positives is the initial heuristic of using parentheses to find method calls. For example, the documentation may use parentheses as a way to convey information, e.g., *": - Response.__nonzero__ (false if bad HTTP Status)"*. The program would extract this code reference and find one parameter, i.e., "false if bad HTTP Status", then match it to the method found in the source code with the same name and number of parameters. However, this reference is not a correct signature of the method, but rather information around its return value. We define false negatives as the program failing to find expected method calls in an inline code snippet with source code method definitions. One reason for the program being unable to match extracted documentation methods is a difference between the method definitions in the source code and the usage of the documentation method. For example, the jBinary [30] library does not conform to the patterns of the other eight initial libraries. In this instance, jBinary utilizes JavaScript's "proto property" to define its methods within its source code. However, the jBinary documentation demonstrates how to use the library by referencing the methods directly. This means the usage in the documentation differs from the definition in the source code.

5.3 Readability of Text and Code

In this section, we explain the metrics for calculating the "readability" of documentation text and code examples. This corresponds with the "Readability of text" and "Readability of code" metrics in Figure 4.2 and relates to the "Readability" aspect of documentation.

5.3.1 Readability of Text

We calculate the readability of text using the Flesch reading ease metric [17]. This metric was developed by Flesch in order to assess the difficulty of a text based on its readability; the higher the score, the easier the text is to read and understand. Although the Flesch reading ease metric [17] was originally developed for improving the readability of newspapers for the Associated Press in the United States of America, it is still one of the most accurate and commonly used metrics when evaluating the readability of text. As mentioned in Chapter 3.2.2, the Flesch reading ease metric [17] does not rely on any external information aside from the text itself and it generates a score between 1 and 100, where the higher the score means the higher its readability, i.e., the easier it is to read. The formula for the metric is shown in Equation 5.1. The simplicity and accuracy of the Flesch reading ease metric [17] makes it one of the most traditionally used readability formulas, even in other attempts of rating the readability of software documentation [3], [19], [25], [58].

We incorporate the Flesch reading ease metric [17] into our summary tool by taking the average score calculated on each page of the documentation. We do this by first extracting all paragraphs, designated by the HTML `<p>` tag, from each documentation page of a library and combine them into one text, then we apply the formula in Equation 5.1 on each page. To extract the words and sentences from the combined text, we use a tokenizer while we extract syllables using a regular expression. The regular expression uses the heuristic of the number syllables being equal to the number of vowels (or vowel pairs) in a word, e.g., "onomatopoeia" has six vowels/vowel pairs, and has six syllables ("o", "o", "a", "o", 'oe', "ia").

$$Score = 206.835 - 1.015(\frac{totalwords}{totalsentences}) - 84.6(\frac{totalsyllables}{totalwords}) \quad (5.1)$$

5.3.2 Readability of Code

We calculate the readability of code using the implementation provided by Scalabrino et al. [56] in their own work. As mentioned in Chapter 3.2.2, Scalabrino et al. [56] proposed a set of updated features based on Buse and Weimer's [6] code readability model and applied machine learning to create a classifier to use on Java documentation. In their work, they found positive and negative correlations, and measured predictive power, between their proposed features and the readability of code. For example, they found that code line length, based on the number of characters, was a negatively correlated strong predictor of code readability, which means the smaller the length of the code line, the more readable it is. Using their findings, they created a machine learning classifier that predicts the readability of Java code using their features. For the purpose of this thesis, we use Scalabrino et al. [56]'s implementation and Java classifier to rate the readability of Java code. However, we are unable to utilize their work to support the other two languages of Python and JavaScript without needing to reimplement their work and train new classifiers. We discuss the extension of these features to classifiers supporting other languages in Chapter 8.2.

5.4 Navigability

5.4.1 HCI checklist

To objectively evaluate the navigability of software documentation, we leverage Human-Computer Interaction (HCI) research. Since most official documentation pages are online webpages, utilizing HCI-research-backed guidelines provides us with a research-supported checklist of features to evaluate online library documentation navigability. Using guidelines from the Web Content Accessibility Guidelines (WCAG) 2 [27] developed by the World Wide Web

Consortium (W3C), we implement an adaptation of their navigability checklist for technical documentation.

We focus on their guidelines surrounding the multiple ways to reach a webpage within the documentation itself, i.e., we focus on searching for the following recommended navigation tools:

- G125: Providing links to navigate to related Web pages
- G64: Providing a Table of Contents
- G63: Providing a site map
- G161: Providing a search function to help users find content
- G126: Providing a list of links to all other Web pages
- G185: Linking to all of the pages on the site from the home page

According to the WCAG [27], a documentation is considered navigable if at least two of these techniques are used. For each of these items, we use a combination of HTML parsing and heuristics to determine whether a documentation implements any of them. To determine if developers implement built-in search functionality (item G161), we create a heuristic to look for input elements with an attribute of the name *"placeholder"*, or form elements, that also contain a *"search"* class. To check for a table of contents (item G64), we parse the HTML to look for HTML `<div>` tags that have a class of "navigation" or an ID of "sidebar". If either of those checks fail, we instead look for all ordered lists (HTML `<ol>` tag) and unordered lists (HTML `<ul>` tag) on the homepage and check if the list contents contain links (HTML `<href>` tag) to the other pages of the documentation, i.e., the links share the same domain as the home page of the documentation. We check for the different links to other pages of documentation by extracting all HTML `<a href>` tags on the page (not restricted to being in `<ol>` and `<ul>` elements), then matching domains between the link and the documentation home page. Finally, we create a heuristic to check for "quick-start" and/or installation guides by checking the

contents of ordered and unordered lists for such a link. This heuristic is added based on the feedback of our interview participants in Chapter 4.6.1. We create each heuristic by manually investigating the combination of the nine libraries used in Chapter 5.1.3 and Chapter 5.2.1 to find the different HTML structures of each item. We then run the heuristics through six additional libraries for a total of 15 libraries our heuristics worked correctly on. Of the 15 libraries, our checklist runs successfully on all libraries for each heuristic that was applicable.

Chapter 6

Survey Evaluation of Our Documentation Quality Summary

To evaluate our documentation quality summary, we run a survey with software developers. In this section we describe the survey setup, participant recruitment, and analysis of results. The goal of this survey is to evaluate our summary tool and gauge its effectiveness in providing developers an idea of the documentation quality for a library. To determine this, we ask developers to rate the usefulness of the different metrics in the context of evaluating documentation quality. Additionally, for users that are familiar with a library, we ask whether our summary evaluation matches their expectations of that library's documentation quality. Figure 6.2 is a screenshot of our survey, which is shown to the right of the documentation summary.

6.1 Survey Setup

To evaluate whether the metrics we select are useful to developers in evaluating a library's documentation quality, we create a website to display the metrics, as well as a survey attached to that website to retrieve feedback from users. Our research questions are as follows:

- RQ1. How useful is our documentation summary for assessing the documentation quality of a library?

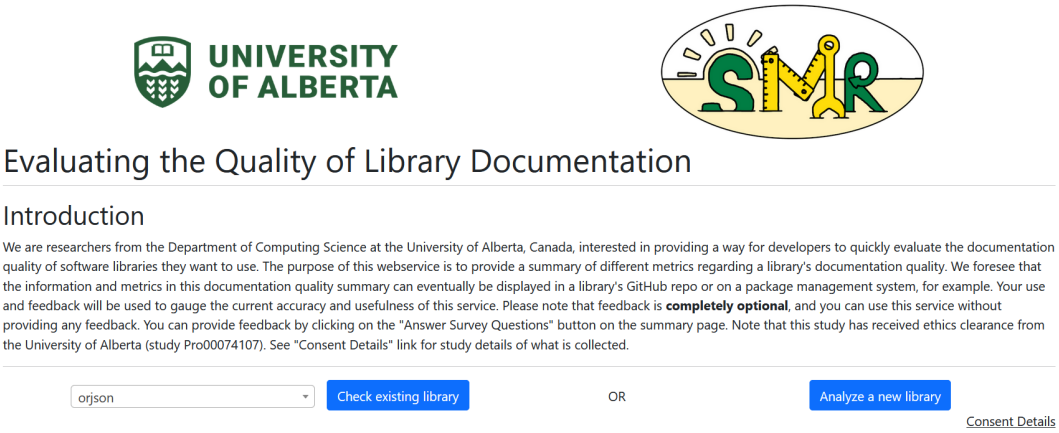

Figure 6.1: The landing page of our summary tool

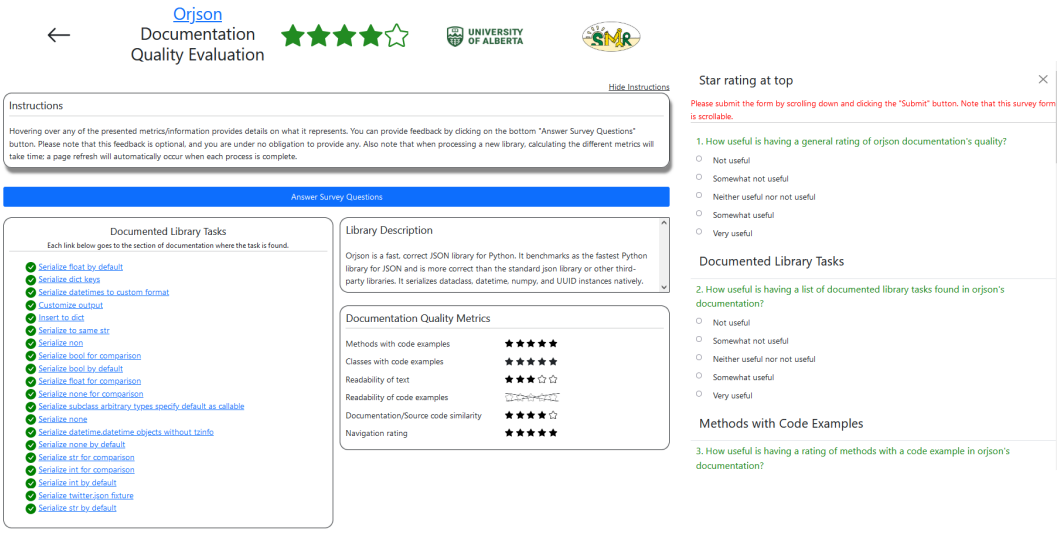

Figure 6.2: A screenshot of our survey within the summary tool

- RQ2. Does our summary match users' experience with the library?
- RQ3. Where do users want to see this summary information?

Before describing how we answer our research questions, we first describe the flow a user would take when using our summary tool and answering the attached survey. The flow of the survey begins on the landing page, which is shown in Figure 6.1. From here, the user can choose to either select a library with existing summary information or analyze a new library. If the user decides to analyze a new library, they must provide the following information:

- The library name

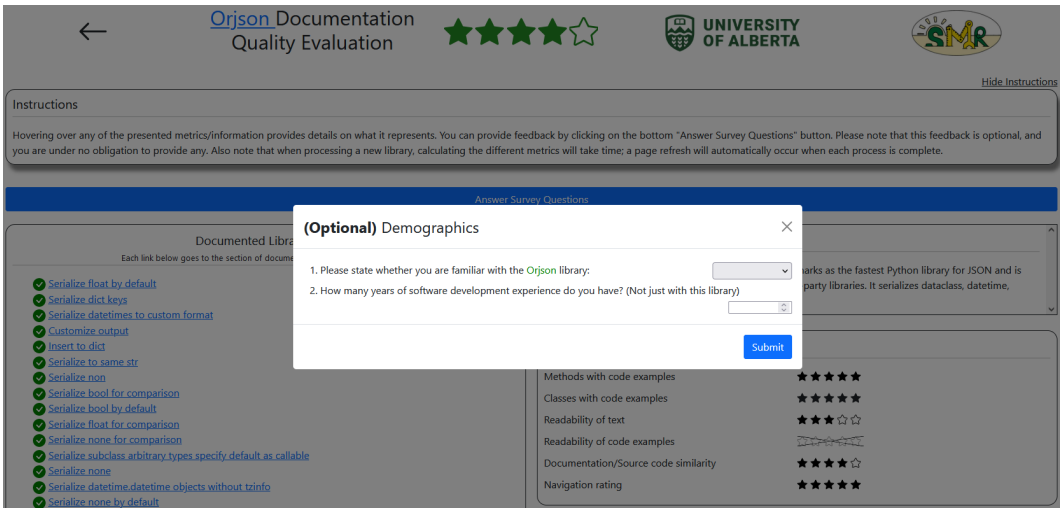

Figure 6.3: A screenshot of the demographic question

- The library's main programming language
- The official documentation URL for the library
- (Optional) The GitHub repo URL

Note that the user provides the documentation URL when analyzing a new library.

We populate the initial list of available libraries with the nine libraries we previously used in the different metric verifications: orjson [28], JSON-Java [62], CoreNLP [48], NLTK [44], jQuery [31], React [50], jBinary [30], QUnit [49], and Requests [51]. After selecting either of those two options, we then ask for the user's **optional** demographic information, shown in Figure 6.3. Regardless of whether the user chooses to provide their demographic information, we then present the summary information about the documentation quality of a library, shown in Figure 4.2. If the user decides to participate in the survey, they can access the survey by clicking on the blue "Answer Survey Questions" button.

Figure 6.2 shows a screenshot of the start of the survey questions. The survey has one question for each of the metrics we present in the summary, with the "General Feedback" portion of the survey asking users if the summary is generally useful, as well as where they want to see such a summary, and a

final free-text answer for additional feedback. The number of survey questions total to 14 and the full list of survey questions is provided in Appendix C.

To answer RQ1, we ask about the usefulness of the individual metrics as a proxy. We use a Likert scale [36] rating for each question of a metric, i.e., each question takes the form: "How useful is having a `<metric name>` in `<library name>` documentation?" For example, one question we ask on the CoreNLP [48] summary page is: "How useful is having a *rating of methods with a code example* in *CoreNLP's* documentation?", with responses being a selection of: "Not useful", "Somewhat not useful", "Neither useful nor not useful", "Somewhat useful", and "Very useful". We provide the full list of questions in Appendix C.

If the user expresses that they are familiar with the library, we ask an additional survey question, along with the other above questions, in order to answer RQ2. This question is: "To what extent do you agree or disagree with this statement: *The documentation quality metrics represented in this summary are consistent with my experience working with* `<library name>` *(e.g., the metrics indicate low documentation quality and your experience is that this library is poorly documented, or vice versa).*", which also has Likert scale [36] responses from "Strongly disagree" to "Strongly agree".

Finally, to answer RQ3, we ask users to respond with where they would like to see our documentation quality summary integrated. We provide a drop down list of four options: "None", "README file badge", "Package manager", and "Other", where selecting the "Other" option allows the user to input their own suggestion.

6.2 Participant Recruitment

We recruit participants for the survey following the Snowball sampling [23] and Convenience sampling [59] methods using social media apps, such as Facebook and Twitter, and personal connections. This resulted in a large spread of developer experience, ranging from one to two years of experience, to 10+ years, with the median value being six years of experience. Since our target demo-

graphic is general software developers, both of these sampling methods work to achieve our goals. As the purpose of this survey is to determine whether the summary tool is useful, asking developers who are close at hand (Convenience sampling [59]) allows us to quickly receive survey responses, while asking them to share the survey with their connections (Snowball sampling [23]) allows us to grow the survey responses among more software developers.

6.3 Survey Response Analysis

While gathering survey responses, we do not ask participants for their name or other identifying information. Instead, we differentiate the different responses through the use of a "session key" generated by Django, in combination with the library the participant is viewing when answering the survey questions. If a user views multiple libraries and submits multiples survey responses using the same browser instance, then the session key will remain the same across all responses with the different library names differentiating the responses. By tracking the session keys in combination with the library, we find at least 168 distinct accesses to our tool and 133 distinct session keys. Unfortunately, due to a bug in the implementation of the survey, early survey responses only saved demographic information and failed to save responses for the ratings of individual metrics. We do not know how many of these early participants filled in the survey. After fixing the bug, we received 25 survey responses.

Along with the survey responses, we also received two endorsements on Twitter in the form of comments expressing their interest and excitement about the summary documentation tool. We present a distribution of all years of experience for both the 168 participants who accessed the service and the distribution of years of experience of the 25 survey responses in Figure 6.4. The rest of our analysis will be based on 25 responses from 21 unique session keys, where the participants filled in at least one of the survey questions. Of these 25 responses, 24 rated every metric, while 1 response only rated the general usefulness of the tool. These 25 responses were made on 12 different libraries. We present the number of responses per library in Figure 6.5. From

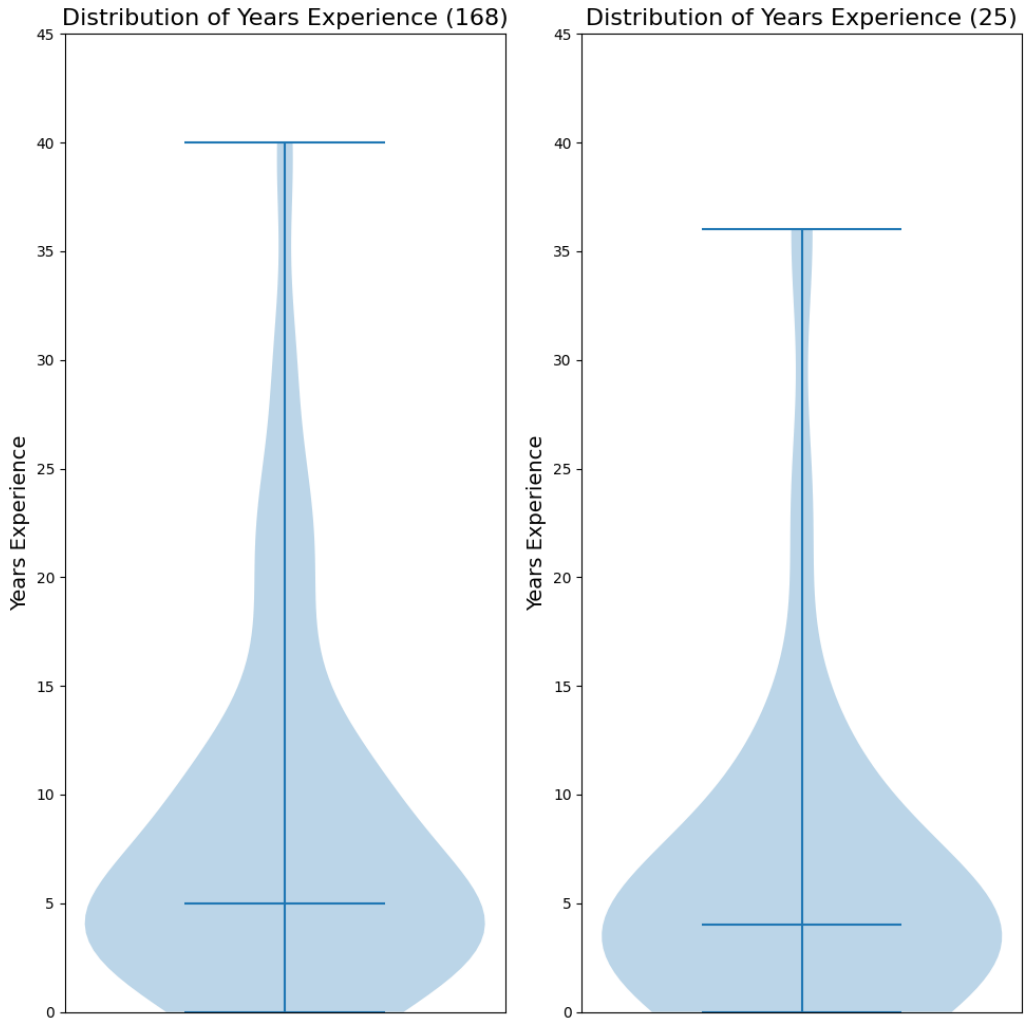

Figure 6.4: Distribution of years of experience of the tool accessors and survey participants

Figure 6.5, we can see that while most libraries have only one response, a few libraries such as “numpy” and “React” are more popular for participants to see a summary of the documentation quality. Of these libraries, it also appears that Python and JavaScript libraries are more popular, as there are only two responses for Java libraries: one for “Mockito” and one for “soot”.

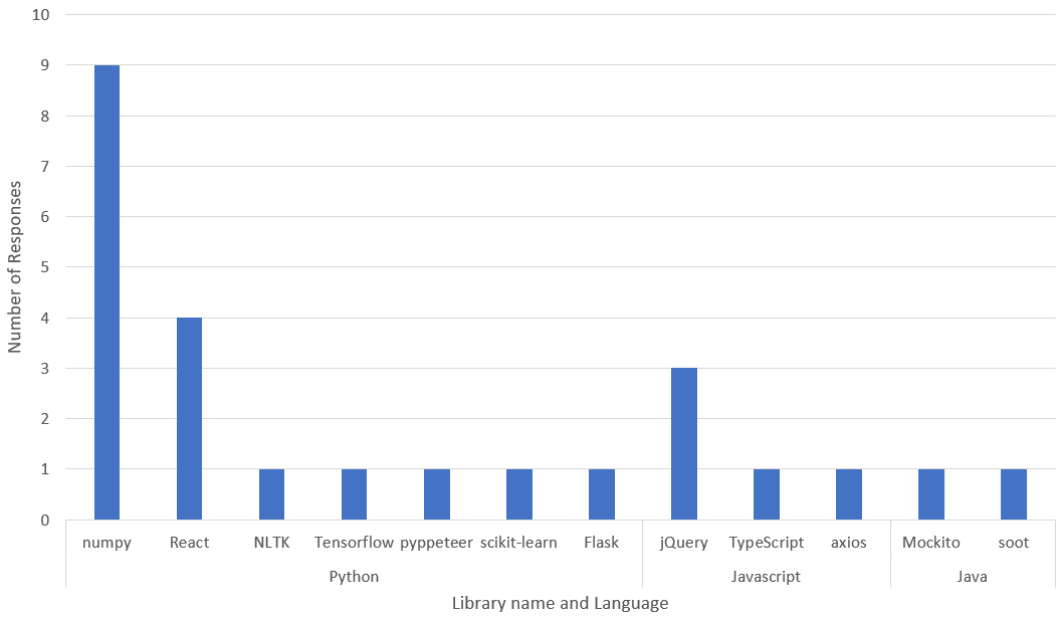

Figure 6.5: The number of responses for each library

6.3.1 RQ1: Usefulness of Our Documentation Quality Summary

To answer our first research question, we analyze the distribution of the 24 responses on each metric, which is presented in Figure 6.6. From the median score of each metric, we see that most participants find most of the provided metrics to be useful. The "General Rating", "Text Readability", and "Navigability" metrics in particular seem to be the most generally useful of our metrics as they have no rating of one, only ratings from two to five, meaning that no participant found any of these metrics as "Not useful". The median scores for each of these metrics is four, meaning most participants found these metrics to be "Somewhat useful". While "Code Readability" does have some ratings of one, its distribution, along with the "Text Readability" and the "Navigability" ratings, suggests that it is one of the highest-rated metrics, signifying that participants view these metrics as the most important metrics of the summary. When viewing the other swells of the violin plot, most metrics have the majority of the responses around a rating of four or five, with the exception of the "Documentation/Source code similarity" metric, which has an even distribution across the different rating values. The "Documenta-

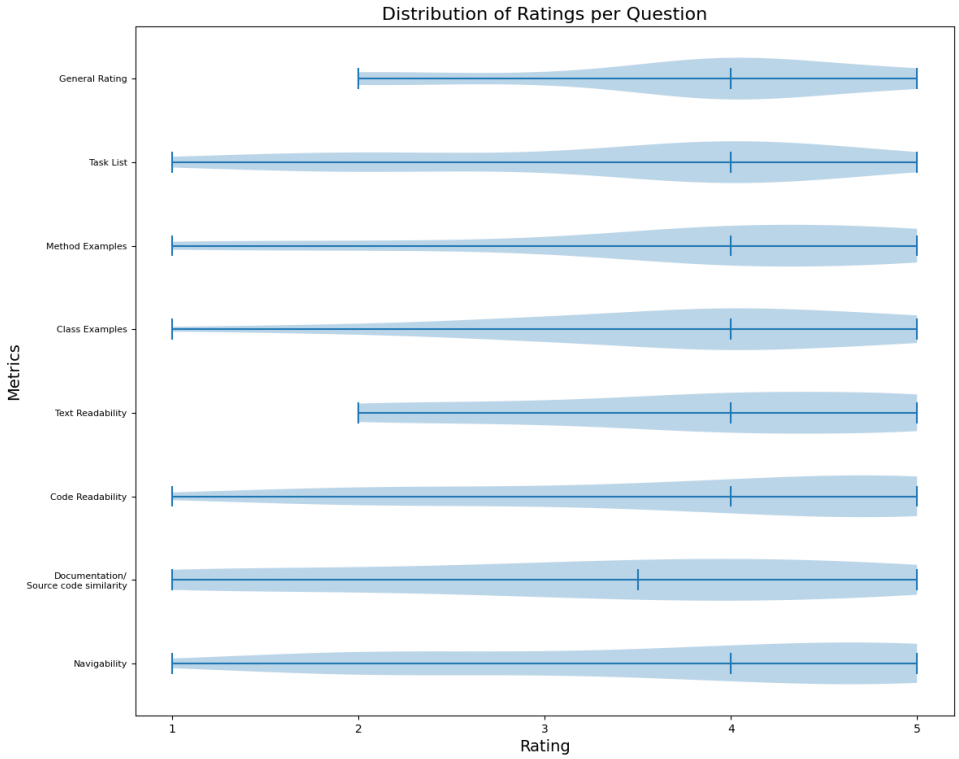

Figure 6.6: The distribution of ratings for each metric in our documentation quality summary

tion/Source code similarity" metric also has the lowest median rating of three and a half. This implies that while our other metrics are "Somewhat useful" to developers when evaluating a library documentation's quality, the "Documentation/Source code similarity" metric has more neutral sentiments, and is generally more controversial in terms of its usefulness. The "Usefulness" metric question asks participants to rate the general usefulness of the survey tool. For this metric, all 25 responses provided a rating. From Figure 6.7, we see that the median rating is four and the violin plot swell is also largest around four, implying our participants agree our tool is "Somewhat useful".

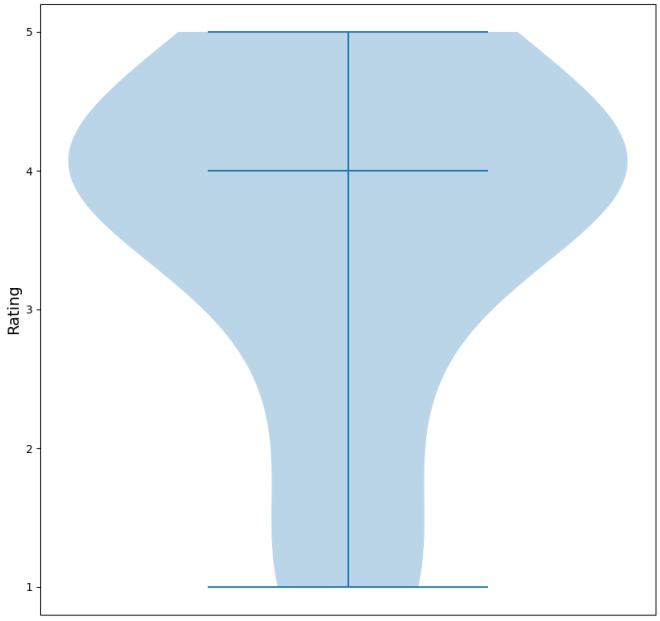

Figure 6.7: The distribution of ratings of the general usefulness of our documentation quality summary

6.3.2 RQ2: Matching User Expectation of Summary

For our second research question, we first retrieve the responses of participants that are familiar with the library they were responding about. This results in us analyzing 21 of the 25 responses. We plot the distribution of ratings asking participants whether the summary matches their expectations of the documentation quality. We present this information in Figure 6.8. As we can see from this graph, the median responses is a rating of three, meaning that most participants "Neither agree nor disagree" that our summary matches their expectations of the library documentation's quality for each of our presented metrics.

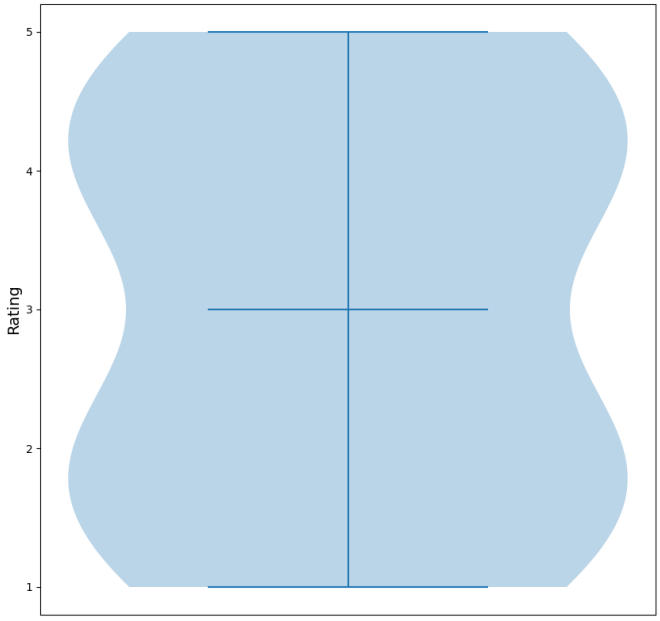

Figure 6.8: The distribution of ratings of whether the documentation quality summary matches user expectations for 21 participants familiar with the library they were viewing

6.3.3 RQ3: Integration of Documentation Quality Summary

Finally, to answer RQ3, we ask participants where they would like to see library documentation quality summaries. As a reminder, this question provides four options of "None", "README file badge", "Package manager", and "Other", where "Other" allows users to input their own suggestion. Figure 6.9 shows the responses given by the participants and we see that most participants would like a documentation quality summary in a README file badge, while others would prefer the summary as part of a package manager. One "Other" response expresses interest in seeing our summary tool in "Both readme and package manager page", which we simplify in Figure 6.9 as "Both". The integration of this documentation quality summary into README file badges and package managers allows developers to see high-level information about

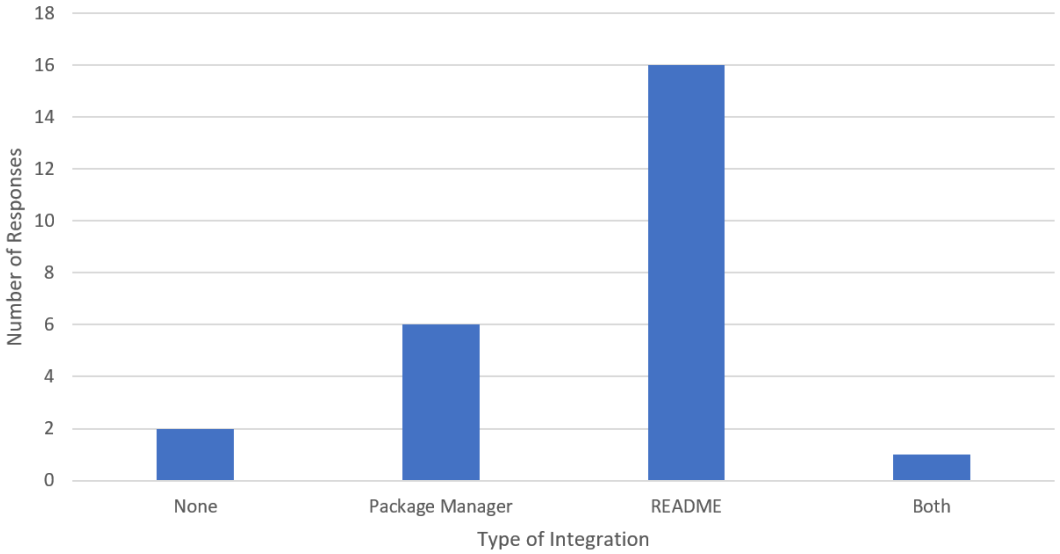

Figure 6.9: The responses for where users would like to see this summary

the documentation quality at a glance, which we discuss in further detail in Chapter 8.1.

6.3.4 General Comments

Of the 25 responses we received, 10 participants responded with general feedback. We provide the list of general comments in Appendix D. While most participants agree with the usefulness of each of our metrics, most general feedback comments are regarding confusion with the calculations behind the different metrics. For example, one participant's response is that *"My experience with React's documentation has been overall pleasant. While I agree with the navigation rating, I disagree with the documentation quality metric for "Readability of text" and "Readability of code examples" - I think the text and code examples are very easy to understand and easy to read."*. This suggests that other more sophisticated measures of readability should be used, or an explanation of the calculations for the readability metrics should be provided. Another feedback comment expressing confusion with the metrics is *"I find it difficult to understand how the metrics are actually populated: what data are they based on? How were they calculated? Based on what data? Which version of jQuery? jQuery is a very bizarre library. I just find it difficult to*

draw a conclusion from jQuery's documentation to these metrics. Since most the functionality is exposed on the $(selector) object. How are code examples extracted?" We can also address this comment by explaining the calculations behind the different metrics. Although we already provide information about the calculations via tooltips when hovering over the metrics, adding further detail within the tooltips will be cumbersome to read and may cover other information in the summary. Instead, one idea would be a separate page containing the details of the specific calculations for a library for those interested to check. Other feedback comments include suggestions with other areas of the documentation we can try to measure, such as this response: *"Since jQuery is a popular package documenting some security practices/well-known issues might be helpful too. I am not sure how feasible this is though."* We discuss potential improvements to our documentation quality summary in Chapter 8.

Chapter 7

Threats to Validity

In this section, we discuss the limitations and threats to the validity of our results.

7.1 Construct Validity

Since we rely on manual validation to create the ground truths for each of the different metrics, there is a risk that the tasks extracted, or code snippet links (to documentation tasks, or source code methods) in the ground truth, are subjective. When creating the ground truth for extracted tasks, the thesis author and another research collaborator separately annotate the dataset for extracted tasks from documentation paragraphs. We combat the subjectivity of the task extraction by defining what a "task" is to help aid the research collaborators in objectively determining what is and what is not a task. As a reminder, this definition is: *any usage of the library, including its API, installation instructions, and interaction with other software*, which we first define in Section 3.2.1. For any disagreements that occurred between the collaborators, they are discussed and resolved, erring on the side of precision, meaning extracted tasks are confirmed only if it is unquestionable. The linking of code examples to documentation paragraphs is a more objective assignment, where the only (informal) "rule" is that the paragraph must describe the code example in some way. The two research collaborators again annotate the dataset separately, and we find the agreement between the collaborators to be in near-perfect agreement, meaning the construct validity threat of this

ground truth is low. For any disagreements present, however, we follow the same method of resolving them and erring on the side of precision. Finally, the assignment of code example/snippet linking to source code methods is the most objective, given that methods in the documentation must link to only one source code method. As such, the thesis author is able to create the ground truth themselves without the need of a second opinion on the links.

While there is subjectivity that arises from manual annotations, we combat the subjectivity by stating definitions, rules, and guidelines where needed to obtain objective decisions when creating the different ground truths. For the ground truth assignments that require additional objectivity, we utilize a second researcher to annotate the datasets, with disagreements resolved with precision as a priority.

Another threat to construct validity is the issue of survey participants having a "perceived value" of our summary and evaluation tool when answering our survey questions. Given that there is nothing for our survey participants to compare against, they are likely to rate our summary tool positively. We mitigate this threat by asking participants not only about the usefulness of the summary tool as a whole, but for each individual metric as well. By providing participants tooltips that explain each metric and having participants provide opinions on each metric, it provides us with a more robust view of how each participant views our summary tool. One opportunity that arises from this issue would be to run a user study asking participants to compare parts of our summary tool against other existing documentation quality evaluation metrics for the respective part; however, many of those tools will require updating or implementation. For example, the user study can ask participants to compare our summary tool's evaluation of a documentation's up-to-dateness, completeness, and readability against an implementation of Aversano et al. [3]'s defined metrics for these documentation aspects.

7.2 Internal Validity

In Chapter 4, we discuss the initial validation of our selected metrics through an interview study. Although interview responses from only three industry professionals are present, the purpose of the interview is to receive initial feedback on the selected metrics and presentation of the summary, not a comprehensive evaluation of our tool. As such, we incorporate the feedback we received into our final implementation of the summary tool and metrics, especially given the experience of the participants, which spans multiple years of experience and roles.

Another threat to internal validity is the way we identify survey responses. As described in Chapter 6.3, we differentiate responses through a combination of a Django-generated session keys and the library the user was viewing. As such, it is difficult to determine if each response was made by an individual person, or if the same user submitted multiple responses on different sessions accessing the website (closing and reopening the browser will do this). If it is the case that the same person rates different libraries, their ratings on the different metrics are valid; however, the questions of the general usefulness of the tool and where they would like to see the documentation quality summary integrated may be skewed. Additionally, the user may rate the same library twice, skewing the results for all the questions. However, we mitigate this threat by filtering the responses before analysis, which we describe in Section 6.3. We also take note of the session key, as a session key remains the same while a user is using the same instance of their browser while viewing the tool, even if they view a different library, and we show in Section 6.3 that there are 19 unique session keys. This provides an adequate level of mitigation in determining the validity of the survey responses themselves and, consequently, the reliability of the survey responses.

Finally, we address the internal validity issue of the created domain lists listed in Appendix B. The thesis author creates these lists through manual experimentation and reading of the six different libraries used in Chapter 5.1.3. This brings into question the validity of the generalizability of the patterns

extracted from the different documentation sources. However, we mitigate this threat through the quantity of documentation pages in use when creating each domain list, which totals 148 different documentation pages. Although not an exhaustive analysis of all libraries in the respective domains, it is enough to create an initial domain list for the different domains that increase the precision of task extraction by an average of 40% across the different tested libraries.

7.3 External Validity

A threat to our external validity is that we analyze eight documentation pages over seven libraries and three languages when creating the ground truth for the "Documented Task List & Linked Examples" metric (Chapter 5.1.3), and 242 documentation pages when creating the ground truth for "Documentation and Source Code Linking" metrics (Chapter 5.2.1). Even though this sample is not exhaustive of all software libraries, it helps us evaluate our technique. Additionally, the metrics we developed against that ground truth received positive feedback from the results of the survey described in Chapter 6. During our survey analysis, we show that most participants find our summary tool to be generally useful, and each of the metrics being useful when evaluating documentation quality.

Another threat to our external validity is that we have responses from only 25 developers. We believe that the small number of responses is due to a technical bug we had, which resulted in survey responses not being caught. While the bug is fixed, we do not know how many of the early survey responses are lost. However, of the 25 responses we did receive, the responses cover a wide range of experienced developers over 12 different libraries in each of the three supported languages. From Figure 6.4, we see that the distribution of years of development experience of the 25 responses is similar to the distribution of years of experience of the 168 responses. While not a statistically significant sample of all software developers, the 25 responses we received still indicates our tool and its metrics to be useful.

Chapter 8

Discussion

In this section, we discuss possible future applications that can utilize our documentation quality summary work, as well as challenges encountered and opportunities for improvement.

8.1 Applications

Our documentation quality summary can integrate with other tools evaluating open source or company libraries. For example, our documentation quality summarization could integrate with the previous work done by De la Mora and Nadi [41]. In their work, they created a library comparison website, which we provide a visualization for in Figure 8.1.

From Figure 8.1, we can see that their work presents various metrics surrounding libraries as a whole, e.g., a metric indicating the popularity of the library, or the average issue response time. Our documentation quality summarization could be another metric that provides developers more information about the documentation quality of a library. De la Mora and Nadi [41] conducted a survey asking developers about their own library comparison tool and also asked developers about other additional features they wanted included in the comparison. Of the 61 responses they received, documentation quality was one of the most demanded additional features. One integration of our documentation quality summary would be using our "General Rating" as another row to the original metrics of their tool, while also allowing a way to view our full summary of the documentation quality (e.g., by clicking on the star

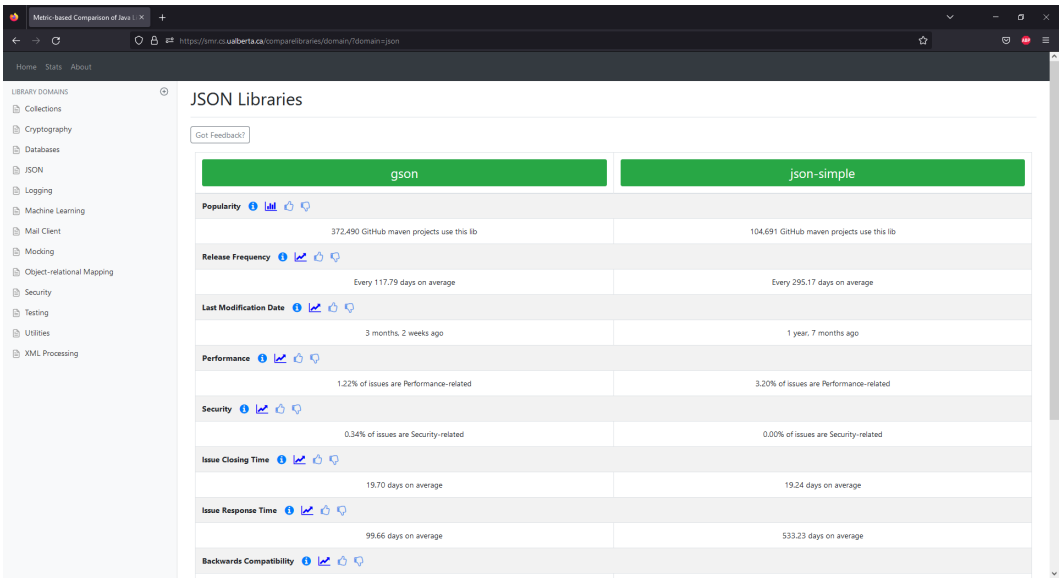

Figure 8.1: A screenshot of the Library Comparison tool by De la Mora and Nadi [41]

rating).

Other applications of our work could be integration into other formats that provide quick evaluation of a library. From our analysis in Section 6.3, we find that 16 respondents like the idea of integrating documentation quality summaries into README files, possibly as a badge, and 6 respondents would like to integrate it within package managers. README badges are highlights to a repository that convey important information to prospective developers. For example, a popular README badge is a dynamic badge that calculates the code coverage of a repository. A documentation quality badge could also be a useful dynamic badge that developers of a library can display in their README file.

Package managers are tools that assist developers to install, upgrade, and track the packages (or libraries) that are in use for a project. Some package managers already present a high-level evaluation of a package to developers. Node Package Manager (NPM) displays a small bar chart for each package that conveys a package's "popularity", "quality", and "maintenance", which is calculated through metrics implemented by npms.io [1]. Our documentation

[1]https://npms.io/about

quality summary could integrate into NPM and other package managers that want to provide a high-level evaluation of a library's documentation quality.

8.2 Challenges and Opportunities

In this section, we discuss challenges and future opportunities of improvement.

8.2.1 Challenges

Extraction Accuracy One challenge is accurately extracting "useful" tasks from documentation. Since we utilize the TaskNav application created by Treude et al. [66], most libraries are evaluated using their defined "task verb list" [2], which we provide in Appendix B.1. However, as mentioned earlier, we find this list to be limited as a general task list for all software domains. Instead, specific domain verb lists are needed in order to accurately extract the tasks for a domain. Although we create domain verb lists for Json, NLP, and DOM manipulation domains, there are many other software domains that need their own domain verb lists to accurately extract tasks from their respective documentation. The creation of custom domain verb lists for different software domains is tedious, if not infeasible, as these lists need to be manually tailored. There is also no authority over what is and what is not a software domain, which would potentially lead to confusion of how to classify a particular library.

HTML Parsing Another challenge is that our implementation relies heavily on HTML parsing of online library documentation. Since there is no standard online documentation format, much of our implementation depends on finding patterns for the different kinds of documentation we encounter. There is a probability that there is "good" documentation that we evaluate as "bad" because our implementation is not able to match the documentation's HTML with one of our defined patterns. However, our analysis is made over 242 different documentation pages over 9 different libraries in 3 different languages, which gives confidence in the generalizability of our tool. Additionally, as

[2]https://www.cs.mcgill.ca/ swevo/tasknavigator/

shown in Figure 6.5, the survey responses are mainly on other libraries aside from our original set of nine libraries, where most respondents still find our tool useful for those other libraries. To overcome this challenge, one improvement could be the development of an online library documentation standard, which developers can adopt for their own libraries (similar to Javadocs).

8.2.2 Opportunities

Although we received positive feedback from developers on both the usefulness of our metrics and summary tool, there are always opportunities for improvement. In response to the general feedback from the survey, some comments convey confusion with the calculations behind the different metrics. Even though we have tooltips that appear when hovering over the individual metrics, one beneficial idea would be an FAQ section that details the breakdown of each metric and how it is calculated. This would provide clarity to users and would potentially explain the scores for metrics they may disagree with.

Other categories of improvement include the expansion of current metrics for both accuracy and generalizability, and other areas of integration for a documentation quality summary. One opportunity for expansion is creating classifiers for other languages using the baseline from Scalabrino et al. [56]'s research. As mentioned in Chapter 5.3.2, our tool utilizes an implementation provided by Scalabrino et al. [56], along with a machine learning classifier they trained on Java code. The inclusion of classifiers in other languages, such as Python and JavaScript, will increase the usefulness and generalizability of our summary tool, which will in turn support additional languages aside from the three languages covered in this thesis.

Another area of improvement would be an expansion of our existing metrics, or even the addition of other metrics into our summarization of documentation quality. For example, our text readability metric uses the Flesch reading ease metric [17]. However, in Chapter 3.2.2 we also describe other readability metrics that are more accurate, but require external expert opinion on curating datasets particular to the language used in software libraries. One opportunity would be the creation of these datasets in order to utilize

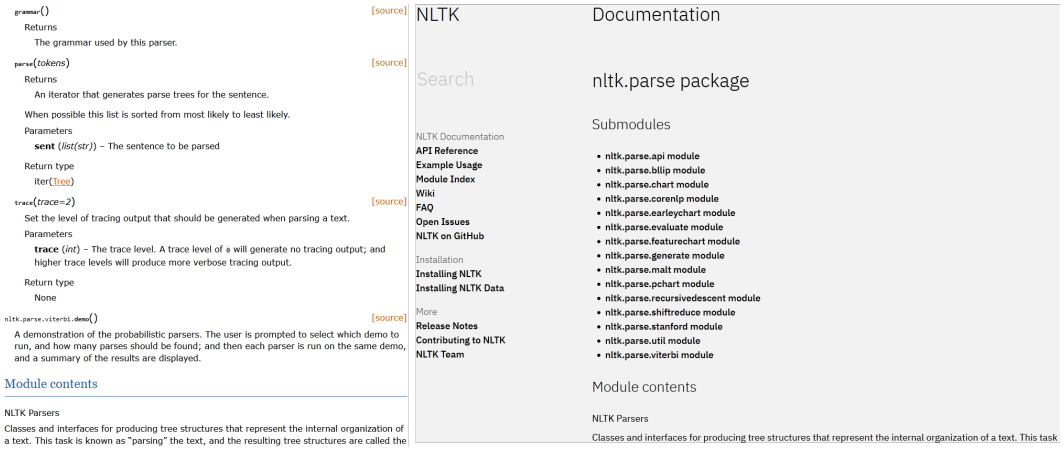

Figure 8.2: Old (left) and new (right) NLTK parse package documentation page

these metrics for library documentation.

Another opportunity for gauging the accuracy of our summary tool, as well as a potential application, is reaching out to the library maintainers with our evaluation of their library's documentation. If library maintainers agree with our evaluation of their documentation, then they could in turn improve their documentation to achieve a higher score. Contrastingly, if library maintainers disagree with our evaluation, we would be able to ask for feedback to improve our metrics to more accurately reflect how the library's documentation is perceived. For example, during the course of this research, the NLTK [44] documentation was updated. We show an example updated page in Figure 8.2. Although we did not reach out to the NLTK maintainers to trigger this change, it would be beneficial to run our evaluation on both the old and new documentation versions and ask NLTK developers whether they agree or disagree with the evaluations. According to our summary tool, the new NLTK documentation has a better readability score compared to the old NLTK documentation. This aligns with an initiative by the NLTK development team to update the NLTK online documentation for greater readability [3].

In terms of other metrics that can supplement our existing summary, we can revisit the table shown in Table 2.1. As a reminder, the chosen documentation aspects we select are highlighted in Table 2.1. Even though we decide to focus

[3]https://github.com/nltk/nltk/pull/2845

on these documentation aspects, other aspects may also be useful to include, such as the "Relevance of Content", and "Consistency" aspects. For example, one metric we can envision for the "Consistency" metric is to compare the words describing the different tasks found in different documentation pages. A metric for the "Relevance of Content" aspect could be a clustering of the documentation pages to see how redundant some pages may be, which may be combined with a previous metric, such as whether the pages in the same cluster have different code examples or task explanations. Pages that are not in a cluster, or with nothing unique to the other pages in a cluster, may be flagged for redundancy.

Chapter 9

Conclusion

In this thesis, we surveyed the literature on the importance of documentation in learning and using software libraries, extracted documentation aspects defined throughout the literature, and created metrics to measure select documentation aspects. We then validated our metrics through an interview study, before automatically implementing it as a documentation quality online summary tool for nine different libraries over three programming languages. We presented our metrics in a summary of a library documentation quality through an online tool available for public use. We evaluated our documentation summary tool through a survey with 25 responses, with the results of our survey concluding that our summary as a whole is useful, with positive responses for all metrics. We then discussed some challenges we faced, as well as opportunities for improvement and extension, including the integration of our documentation summary into other evaluation tools. All of our data and code are available online on our artifact page [47]. We hope that this research along with the discussion we provide encourages further research into this area of evaluating software documentation quality.

References

[1] E. Aghajani, C. Nagy, M. Linares-Vásquez, *et al.*, "Software documentation: The practitioners' perspective," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 2020, pp. 590–601.

[2] E. Aghajani, C. Nagy, O. L. Vega-Márquez, *et al.*, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[3] L. Aversano, D. Guardabascio, and M. Tortorella, "Evaluating the quality of the documentation of open source software.," in *ENASE*, 2017, pp. 308–313.

[4] F. Bachmann, L. Bass, J. Carriere, *et al.*, "Software architecture documentation in practice: Documenting architectural layers," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2000.

[5] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010. DOI: 10.1109/TSE.2009.70.

[6] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010. DOI: 10.1109/TSE.2009.70.

[7] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological methods & research*, vol. 42, no. 3, pp. 294–320, 2013.

[8] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *Journal of Systems and Software*, vol. 82, no. 6, pp. 981–992, 2009, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2008.12.036. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121208002793.

[9] *Codepen feature*. [Online]. Available: https://codepen.io/features/.

[10] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[11] T. Comber, "Building usable web pages: An hci perspective," in *Proceedings of the First Australian World Wide Web Conference*, Norsearch, Ballina, Australia, 1995, pp. 119–124.

[12] *Command line usage*. [Online]. Available: `https://stanfordnlp.github.io/CoreNLP/cmdline.html`.

[13] *Components and props*. [Online]. Available: `https://reactjs.org/docs/components-and-props.html`.

[14] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[15] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[16] R. Flesch, "Flesch-kincaid readability test," *Retrieved October*, vol. 26, no. 3, p. 2007, 2007.

[17] R. Flesch, "A new readability yardstick.," *Journal of applied psychology*, vol. 32, no. 3, p. 221, 1948.

[18] P. W. Foltz, "Latent semantic analysis for text-based research," *Behavior Research Methods, Instruments, & Computers*, vol. 28, no. 2, pp. 197–202, 1996.

[19] A. Forward, *Software documentation: Building and maintaining artefacts of communication*. University of Ottawa (Canada), 2002.

[20] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," ser. DocEng '02, McLean, Virginia, USA: Association for Computing Machinery, 2002, pp. 26–33, ISBN: 1581135947. DOI: `10.1145/585058.585065`. [Online]. Available: `https://doi.org/10.1145/585058.585065`.

[21] G. Garousi, V. Garousi, M. Moussavi, G. Ruhe, and B. Smith, "Evaluating usage and quality of technical software documentation: An empirical study," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13, Porto de Galinhas, Brazil: Association for Computing Machinery, 2013, pp. 24–35, ISBN: 9781450318488. DOI: `10.1145/2460999.2461003`. [Online]. Available: `https://doi.org/10.1145/2460999.2461003`.

[22] G. Garousi, V. Garousi-Yusifoğlu, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith, "Usage and usefulness of technical software documentation: An industrial case study," *Information and Software Technology*, vol. 57, pp. 664–682, 2015, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.08.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491400192X.

[23] L. A. Goodman, "Snowball sampling," *The annals of mathematical statistics*, pp. 148–170, 1961.

[24] A. C. Graesser, D. S. McNamara, M. M. Louwerse, and Z. Cai, "Coh-metrix: Analysis of text on cohesion and language," *Behavior research methods, instruments, & computers*, vol. 36, no. 2, pp. 193–202, 2004.

[25] G. Hargis, "Readability and computer documentation," *ACM J. Comput. Doc.*, vol. 24, no. 3, pp. 122–131, 2000, ISSN: 1527-6805. DOI: 10.1145/344599.344634. [Online]. Available: https://doi.org/10.1145/344599.344634.

[26] A. Hora and M. T. Valente, "Apiwave: Keeping track of api popularity and migration," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 321–323. DOI: 10.1109/ICSM.2015.7332478.

[27] *How to meet wcag (quick reference)*, 2008. [Online]. Available: https://www.w3.org/WAI/WCAG21/quickref/?showtechniques=245#multiple-ways.

[28] Ijl, *Ijl/orjson: Fast, correct python json library supporting dataclasses, datetimes, and numpy*. [Online]. Available: https://github.com/ijl/orjson.

[29] A. Jazzar and W. Scacchi, "Understanding the requirements for information system documentation: An empirical investigation," in *Proceedings of Conference on Organizational Computing Systems*, ser. COCS '95, Milpitas, California, USA: Association for Computing Machinery, 1995, pp. 268–279, ISBN: 0897917065. DOI: 10.1145/224019.224048. [Online]. Available: https://doi.org/10.1145/224019.224048.

[30] *Jbinary*. [Online]. Available: https://github.com/jDataView/jBinary.

[31] *Jquery*. [Online]. Available: https://jquery.com/.

[32] *Jquery.get*. [Online]. Available: https://api.jquery.com/jQuery.get/.

[33] J. M. Kleinberg *et al.*, "Authoritative sources in a hyperlinked environment.," in *SODA*, Citeseer, vol. 98, 1998, pp. 668–677.

[34] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 245–256, ISBN: 9781450370431. DOI: 10.1145/3368089.3409711. [Online]. Available: https://doi.org/10.1145/3368089.3409711.

[35] S. Lee, R. Wu, S.-C. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated api names in documentation," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 653–675, 2021. DOI: 10.1109/TSE.2019.2901459.

[36] R. Likert, "A technique for the measurement of attitudes.," *Archives of psychology*, 1932.

[37] R. Longhurst, "Semi-structured interviews and focus groups," *Key methods in geography*, vol. 3, no. 2, pp. 143–156, 2003.

[38] X. Lu, "Automatic analysis of syntactic complexity in second language writing," *International journal of corpus linguistics*, vol. 15, no. 4, pp. 474–496, 2010.

[39] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol '09, Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 57–62, ISBN: 9781605586786. DOI: 10.1145/1595808.1595821. [Online]. Available: https://doi.org/10.1145/1595808.1595821.

[40] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Testing – Practice and Research Techniques*, L. Bottaci and G. Fraser, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 173–180, ISBN: 978-3-642-15585-7.

[41] F. L. de la Mora and S. Nadi, "An empirical study of metric-based comparisons of software libraries," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18, Oulu, Finland: Association for Computing Machinery, 2018, pp. 22–31, ISBN: 9781450365932. DOI: 10.1145/3273934.3273937. [Online]. Available: https://doi.org/10.1145/3273934.3273937.

[42] *Named entity recognition*. [Online]. Available: https://stanfordnlp.github.io/CoreNLP/ner.html.

[43] C. Nicolle and J. Abascal, *Inclusive design guidelines for HCI*. CRC Press, 2001.

[44] *Nltk*. [Online]. Available: https://www.nltk.org/.

[45] *Nltk.parse package*. [Online]. Available: https://www.nltk.org/api/nltk.parse.html.

[46] *Nltk.tag package*. [Online]. Available: https://www.nltk.org/api/nltk.tag.html.

[47] *Online artifact page*, https://github.com/ualberta-smr/tang-task-extractor.

[48] *Overview*. [Online]. Available: https://stanfordnlp.github.io/CoreNLP/.

[49] *Qunit*. [Online]. Available: https://qunitjs.com/.

[50] *React*. [Online]. Available: https://reactjs.org/.

[51] *Requests*. [Online]. Available: https://requests.readthedocs.io/en/latest/.

[52] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[53] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009. DOI: 10.1109/MS.2009.193.

[54] J. Rowley, "Guidelines on the evaluation and selection of library software packages," in *Aslib proceedings*, MCB UP Ltd, 1990.

[55] J. Saldaña, *The coding manual for qualitative researchers*. sage, 2021.

[56] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, e1958, 2018.

[57] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503707.

[58] D. Schreck, V. Dallmeier, and T. Zimmermann, "How documentation evolves over time," in *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. IWPSE '07, Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 4–10, ISBN: 9781595937223. DOI: 10.1145/1294948.1294952. [Online]. Available: https://doi.org/10.1145/1294948.1294952.

[59] P. Sedgwick, "Convenience sampling," *Bmj*, vol. 347, 2013.

[60] R. van Solingen (Revision), V. Basili (Original article 1994 ed.), G. Caldiera (Original article 1994 ed.), and H. D. Rombach (Original article 1994 ed.), "Goal question metric (gqm) approach," in *Encyclopedia of Software Engineering*. John Wiley & Sons, Ltd, 2002, ISBN: 9780471028956. DOI: https://doi.org/10.1002/0471028959.sof142. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471028959.sof142. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof142.

[61] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, ser. SIGDOC '05, Coventry, United Kingdom: Association for Computing Machinery, 2005, pp. 68–75, ISBN: 1595931759. DOI: 10.1145/1085313.1085331. [Online]. Available: https://doi.org/10.1145/1085313.1085331.

[62] Stleary, *Json in java*. [Online]. Available: https://github.com/stleary/JSON-java.

[63] A. L. Strauss, *Qualitative analysis for social scientists*. Cambridge university press, 1987.

[64] F. Thung, "Api recommendation system for software development," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 896–899.

[65] C. Treude, J. Middleton, and T. Atapattu, "Beyond accuracy: Assessing software documentation quality," *CoRR*, vol. abs/2007.10744, 2020. arXiv: 2007.10744. [Online]. Available: https://arxiv.org/abs/2007.10744.

[66] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 565–581, 2015. DOI: 10.1109/TSE.2014.2387172.

[67] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015. DOI: 10.1109/MS.2014.80.

[68] A. S. M. Venigalla and S. Chimalakonda, "Understanding emotions of developer community towards software documentation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, IEEE, 2021, pp. 87–91.

[69] W3C, *How to meet wcag (quick reference)*. [Online]. Available: https://www.w3.org/WAI/WCAG21/quickref/?showtechniques=245#multiple-ways.

[70] R. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis, "Api documentation and software community values: A survey of open-source api documentation," in *Proceedings of the 31st ACM International Conference on Design of Communication*, ser. SIGDOC '13, Greenville, North Carolina, USA: Association for Computing Machinery, 2013, pp. 165–174, ISBN: 9781450321310. DOI: 10.1145/2507065.2507076. [Online]. Available: https://doi.org/10.1145/2507065.2507076.

[71] A. Wingkvist, M. Ericsson, R. Lincke, and W. Löwe, "A metrics-based approach to technical documentation quality," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, 2010, pp. 476–481. DOI: 10.1109/QUATIC.2010.88.

[72] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, benefits and quality of software development documentation: A systematic mapping," *Journal of Systems and Software*, vol. 99, pp. 175–198, 2015, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.09.042. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002131.

[73] H. Zhong and Z. Su, "Detecting api documentation errors," *SIGPLAN Not.*, vol. 48, no. 10, pp. 803–816, 2013, ISSN: 0362-1340. DOI: 10.1145/2544173.2509523. [Online]. Available: https://doi.org/10.1145/2544173.2509523.

[74] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 27–37. DOI: 10.1109/ICSE.2017.11.

[75] Y. Zhou, X. Yan, T. Chen, S. Panichella, and H. Gall, "Drone: A tool to detect and repair directive defects in java apis documentation," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 115–118. DOI: 10.1109/ICSE-Companion.2019.00052.

Appendix A

Ground truth guidelines

A.1 Task Extraction Guidelines

1. Paragraphs are designated with the HTML `<p>` tag
2. Task definition: *Any usage of a library, including its API, installation instructions, and interaction with other software*
3. Ignore negative verbs (e.g., do *not*), because negatives usually describe a "default" behaviour, rather than a task that a developer would want to *do*

A.2 Task Linking Guidelines

1. Code examples are designated with the HTML `<pre>` tag
2. Paragraphs are designated with the HTML `<p>` tag
3. The paragraph needs to be in the same section as the code example (regardless of position relative to the code example), where sections are separated using HTML header tags (e.g., `<h1>`, `<h2>`)
4. The paragraph needs to describe the task the code example is illustrating

A.3 Documentation Code Reference/Example and Source Code Matching Guidelines

1. Code references are designated with the HTML `<code>` and `<dt>` tags

2. Code examples are designated with the HTML `<pre>` tag

3. Extract method and class names from the code references/examples

4. Scan the source code for matching methods (and potential parent classes) and track the source file name

Appendix B

Programming verb lists

B.1 Original Verb List

access, acquire, activate, add, adjust, align, append, apply, archive, ask, assign, attach, bind, book, boost, bootstrapp, break, cache, calculate, call, catch, change, check, checkout, clean, clone, combine, compare, compile, compose, compress, configure, convert, count, create, customize, deactivate, define, deploy, describe, design, determine, develop, disable, discontinue, display, divide, document, download, duplicate, edit, embed, enable, encode, exclude, execute, expand, expose, extend, fill, find, fit, fix, flush, force, format, generate, get, group, handle, hide, identify, ignore, implement, import, include, indent, inherit, initialize, insert, install, instantiate, integrate, introduce, isolate, learn, limit, link, list, load, locate, log, login, manage, manipulate, mark, match, mock, modify, move, obtain, omit, open, order, output, override, overwrite, pass, paste, patch, place, play, prefix, prepare, present, prevent, print, process, push, raise, reach, read, receive, recompile, recreate, redefine, redirect, refactor, reference, regenerate, release, remove, rename, render, repeat, replace, replicate, request, resolve, restart, retrieve, return, reuse, run, search, select, send, separate, seperate, set, share, shorten, show, sign, simulate, skip, sort, specify, split, store, submit, subscribe, summarize, support, surround, switch, test, throw, track, translate, trigger, trim, update, use, wrap, write, crop, scale, perform, center, produce, delete, encapsulate, save, fetch, differentiate, compute, defer, provide, hardcode, multiply, choose, complete, enter, set up, sell, purchase, price, rebuild, offer, host, log in

B.2 NLP Verb List

access, acquire, activate, add, adjust, align, append, apply, archive, ask, assign, attach, bind, book, boost, bootstrapp, break, cache, calculate, call, catch, change, check, checkout, clean, clone, combine, compare, compile, compose, compress, configure, convert, count, create, customize, deactivate, define, deploy, describe, design, determine, develop, disable, discontinue, display, divide, document, download, duplicate, edit, embed, enable, encode, exclude, execute, expand, expose, extend, fill, find, fit, fix, flush, force, format, generate, get, group, handle, hide, identify, ignore, implement, import, include, indent, inherit, initialize, insert, install, instantiate, integrate, introduce, isolate, learn, limit, link, list, load, locate, log, login, manage, manipulate, mark, match, mock, modify, move, obtain, omit, open, order, output, override, overwrite, pass, paste, patch, place, play, prefix, prepare, present, prevent, print, process, push, raise, reach, read, receive, recompile, recreate, redefine, redirect, refactor, reference, regenerate, release, remove, rename, render, repeat, replace, replicate, request, resolve, restart, retrieve, return, reuse, run, search, select, send, separate, seperate, set, share, shorten, show, sign, simulate, skip, sort, specify, split, store, submit, subscribe, summarize, support, surround switch, test, throw, track, translate, trigger, trim, update, use, wrap, write, crop, scale, perform, center, produce, delete, encapsulate, save, fetch, differentiate, compute, defer, provide, hardcode, multiply, choose, complete, enter, set up, sell, purchase, price, rebuild, offer, host, log in, supports, serialize, deserialize, decode, parse, validate, specialize

B.2.1 NLP Verb Filter List

adjust, apply, combine, count, customize, describe, download, extend, find, generate, implement, include, learn, list, locate, match, omit, place, process, purchase, read, redirect, run, search, separate, seperate, store, use, write

B.3 JSON Verb List

access, acquire, activate, add, adjust, align, append, apply, archive, ask, assign, attach, bind, book, boost, bootstrapp, break, cache, calculate, call, catch, change, check, checkout, clean, clone, combine, compare, compile, compose, compress, configure, convert, count, create, customize, deactivate, define, deploy, describe, design, determine, develop, disable, discontinue, display, divide, document, download, duplicate, edit, embed, enable, encode, exclude, execute, expand, expose, extend, fill, find, fit, fix, flush, force, format, generate, get, group, handle, hide, identify, ignore, implement, import, include, indent, inherit, initialize, insert, install, instantiate, integrate, introduce, isolate, learn, limit, link, list, load, locate, log, login, manage, manipulate, mark, match, mock, modify, move, obtain, omit, open, order, output, override, overwrite, pass, paste, patch, place, play, prefix, prepare, present, prevent, print, process, push, raise, reach, read, receive, recompile, recreate, redefine, redirect, refactor, reference, regenerate, release, remove, rename, render, repeat, replace, replicate, request, resolve, restart, retrieve, return, reuse, run, search, select, send, separate, seperate, set, share, shorten, show, sign, simulate, skip, sort, specify, split, store, submit, subscribe, summarize, support, surround, switch, test, throw, track, translate, trigger, trim, update, use, wrap, write, crop, scale, perform, center, produce, delete, encapsulate, save, fetch, differentiate, compute, defer, provide, hardcode, multiply, choose, complete, enter, set up, sell, purchase, price, rebuild, offer, host, log in, supports, serialize, deserialize, decode, parse, validate, specialize

B.3.1 JSON Verb Filter List

apply, change, compare, create, disable, handle, implement, include, indent, open, pass, provide, raise, remove, return, run, store, submit, support, test, use

B.4 DOM Manipulation Verb List

access, acquire, activate, add, adjust, align, append, apply, archive, ask, assign, attach, bind, book, boost, bootstrapp, break, cache, calculate, call, catch, change, check, checkout, clean, clone, combine, compare, compile, compose, compress, configure, convert, count, create, customize, deactivate, define, deploy, describe, design, determine, develop, disable, discontinue, display, divide, document, download, duplicate, edit, embed, enable, encode, exclude, execute, expand, expose, extend, fill, find, fit, fix, flush, force, format, generate, get, group, handle, hide, identify, ignore, implement, import, include, indent, inherit, initialize, insert, install, instantiate, integrate, introduce, isolate, learn, limit, link, list, load, locate, log, login, manage, manipulate, mark, match, mock, modify, move, obtain, omit, open, order, output, override, overwrite, pass, paste, patch, place, play, prefix, prepare, present, prevent, print, process, push, raise, reach, read, receive, recompile, recreate, redefine, redirect, refactor, reference, regenerate, release, remove, rename, render, repeat, replace, replicate, request, resolve, restart, retrieve, return, reuse, run, search, select, send, separate, seperate, set, share, shorten, show, sign, simulate, skip, sort, specify, split, store, submit, subscribe, summarize, support, surround, switch, test, throw, track, translate, trigger, trim, update, use, wrap, write, crop, scale, perform, center, produce, delete, encapsulate, save, fetch, differentiate, compute, defer, provide, hardcode, multiply, choose, complete, enter, set up, sell, purchase, price, rebuild, offer, host, log in, map, toggle, iterate, serialize, return,

B.4.1 DOM Manipulation Verb Filter List

add, apply, assign, call, change, compare, create, describe, disable, find, handle, implement, indent, insert, introduce, learn, receive, remove, render, return, reuse, specify, store, submit, test, fetch, use, write

Appendix C

Survey Questions

Blue text questions are ones that **do not** have Likert scale [36] responses. These questions have accepted responses below them, if applicable.

1. Please state whether you are familiar with the `<library name>` library:
   - I am
   - I am not
2. How many years of software development experience do you have? (Not just with this library)
   - Positive Integer Value
3. How useful is having a general rating of `<library name>` documentation's quality?
4. How useful is having a list of documented library tasks found in `<library name>`'s documentation?
5. How useful is having a rating of methods with a code example in `<library name>`'s documentation?
6. How useful is having a rating of classes with a code example in `<library name>`'s documentation?
7. How useful is having a rating of readability of text in `<library name>`'s documentation?

8. How useful is having a rating of readability of code examples in `<library name>`'s documentation?

9. How useful is having a rating of similarity between `<library name>`'s source code and documentation?

10. How useful is having a rating of navigability of `<library name>`'s documentation?

11. How useful is having a summary of `<library name>`'s documentation quality?

12. Where would you like to see this documentation quality summary?

    - README file Badge
    - Package Manager
    - Other (Allows the user to type in where they would like to see the summary integrated)

13. To what extent do you agree or disagree with this statement: The documentation quality metrics represented in this summary are consistent with my experience working with `<library name>` (e.g., the metrics indicate low documentation quality and your experience is that this library is poorly documented, or vice versa).

14. Please provide any additional feedback you have about `<library name>`'s documentation quality summary.

    - Free Text Answer

Appendix D

Survey General Comments

1. numpy's documentation is large. It's possible that your crawling just didn't find relevant portions. There is no way for me to tell from the summary what your crawler actually saw. There does not seem to be any relationship between the overall star summary (4/5) and the individual quality metrics (mostly 0/5).

2. Since jQuery is a popular package documenting some security practices/well-known issues might be helpful too. I am not sure how feasible this is though.

3. Doesn't seem to have discovered anything other than installation instructions

4. I don't know whether I am interested in methods examples and class examples (at least for Python).

5. It feels like some of the sections should have much higher important or scores than they do. It is not clear why readability of code examples is crossed out.

6. My experience with React's documentation has been overall pleasant. While I agree with the navigation rating, I disagree with the documentation quality metric for "Readability of text" and "Readability of code examples" - I think the text and code examples are very easy to understand and easy to read.

7. More difficult to properly assess the documentation quality given that the link for official documentation is more of a map for relevant documentation of methods rather than an all-encompassing page.

8. I find it difficult to understand how the metrics are actually populated: what data are they based on? How were they calculated? Based on what data? Which version of jQuery? jQuery is a very bizarre library. I just find it difficult to draw a conclusion from jQuery's documentation to these metrics. since most the functionality is exposed on the $(selector) object. How are code examples extracted?

9. The Flask project starts with a tutorial that teaches you how to use it from the start up. This is a good documentation and it is unclear how these metrics account for solid tutorials. That said it's sometimes difficult to find Flask API methods and types. These metrics do not reflect many of the practical considerations of Flask's documentation quality.

10. Most of the documentation quality metrics are 0 star which seems pretty strange and inaccurate to me. The documented library tasks section also seems quite limited and I feel a google search would be more effective.