Improving API Documentation Quality Through Identifying Missing Code Examples

by

Afiya Fahmida Sarah

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 often provide documentation to describe the Application Programming Interfaces (APIs) they offer. Such API documentation helps potential users understand how to use the library. Studies have shown that having code examples is one of the key criteria of good software documentation quality and that lack of examples is one of the reasons why API documentation becomes difficult to understand for library users. Thus, in this thesis, our goal is to improve API documentation quality by identifying missed opportunities for code examples.

Specifically, we propose a methodology to identify APIs from library API documentation websites that require code examples for improving documentation quality. We use web scraping and heuristics to identify APIs with missing code examples ($\text{APIs}_{\text{noeg}}$) from a library's documentation website. We then look for Stack Overflow questions about the $\text{APIs}_{\text{noeg}}$. Next, we use a large language model (LLM) to determine if the post discusses the author's struggle with using those $\text{APIs}_{\text{noeg}}$. If it does, then we mark that api_{noeg} as a potential API that may need a code example. Finally, we identify a list of APIs, for which, adding code examples can improve the documentation quality. We can then encourage library authors to add code examples for the APIs in this list.

Based on our methodology, we create a prototype tool. We evaluate the intermediate steps of our tool for the API documentation of 3 libraries and then analyze the API documentation of 12 libraries using the tool. We find that although there are Stack Overflow posts where authors discuss an api_{noeg}, it is not always an indicator that having code examples would resolve the authors' issues.

Preface

This thesis is an original work by Afiya Fahmida Sarah. The research project, of which this thesis is a part, received research ethics approval from the University of Alberta Research Ethics Board, Project Name "Task-based Code Recommender Systems", No. Pro00074107-AME7, Oct 19, 2023. .

To my amazing husband, Rafid

Whose endless support, love, and patience has been my strength throughout this journey

And to my beautiful son, Rayan

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Sarah Nadi, for her continuous guidance, support, and patience throughout my graduate studies. She has always given me the best advice and constant motivation throughout this research journey, without which, this would not have been possible.

I would also like to thank Dr. Christoph Treude from Singapore Management University, for his insightful feedback, suggestions, and help with data annotation, which greatly helped my work.

Lastly, my heartfelt thanks to my husband and colleague, Muhtasim Fuad Rafid, for his unwavering support, encouragement, and constructive feedback throughout my research. I would also like to thank him for assisting me in data annotation.

Contents

1 Introduction **1**
- 1.1 Thesis overview 4

2 Literature Review **6**
- 2.1 Why Good Software Documentation is Important 6
- 2.2 Types of Software Documentation 7
- 2.3 Software Documentation Quality Aspects 8
- 2.4 Using Stack Overflow for Detecting Software Documentation Issues 10
- 2.5 Using LLMs to analyze Stack Overflow data 13

3 Methodology **16**
- 3.1 Finding APIs$_{\text{doc}}$: FILTER$_{\text{APIdoc}}$ 18
- 3.2 Finding APIs$_{\text{noeg}}$: FILTER$_{\text{APInoeg}}$ 22
- 3.3 Finding Stack Overflow posts that mention the APIs$_{\text{noeg}}$: FILTER$_{\text{hasAPI}}$ 23
- 3.4 Finding Stack Overflow posts that discuss the APIs$_{\text{noeg}}$: FILTER$_{\text{discussAPI}}$ 27
  - 3.4.1 Heuristic Based Approach 27
  - 3.4.2 LLM Based Approach 29
- 3.5 Prioritize APIs$_{\text{noeg}}$: FILTER$_{\text{prioritize}}$ 31

4 Evaluation of Intermediate Pipeline Steps **35**
- 4.1 Detecting APIs$_{\text{doc}}$ from a documentation website: FILTER$_{\text{APIdoc}}$ 35
- 4.2 Detecting APIs$_{\text{noeg}}$ from a documentation website: FILTER$_{\text{APInoeg}}$ 37
- 4.3 Detecting Stack Overflow posts that discuss the APIs$_{\text{noeg}}$: FILTER$_{\text{discussAPI}}$ 38

5 Pipeline Evaluation **42**
- 5.1 Evaluation Setup 42
- 5.2 Result 43
- 5.3 Manual Verification of Result 44

6 Discussion **50**

7 Threats to validity **54**
- 7.1 Internal Validity 54
- 7.2 Construct Validity 55
- 7.3 External Validity 55

8 Conclusion **57**

References **58**

Appendix A LLM Query Prompt **63**
A.1 Template of the Few-shot Query Prompt With Real Example Posts 63
A.2 Template of the Chain-of-Thought Query Prompt With Real Example Posts 65

Appendix B Non-determinism and Mean Error of LLM Model **68**

List of Tables

4.1 Performance scores (Medians) of the different approaches for $\text{FILTER}_{\text{discussAPI}}$ 39

5.1 Library statistics and results of running our pipeline 43
5.2 Result of $\text{FILTER}_{\text{prioritize}}$ 44

B.1 Error and Non-determinism of LLM model 68

List of Figures

1.1 Examples of an API documentation page of an api$_{\text{noeg}}$ and a SO post discussing the API 3

3.1 The architecture of our system. Colored rectangles are the different processing components of our pipeline and the white rectangles show the input/output of each component. 17
3.2 Example of a SO post that mentions but not discusses the API function `requests.head` 18
3.3 Different layouts of documentation websites 19
3.4 Example of an API reference page and its corresponding HTML 21
3.5 Example of different ways of calling APIs 22
3.6 A flowchart of FILTER$_{\text{hasAPI}}$ 24
3.7 Examples of SO posts 28
3.8 Zero-shot prompt template 30
3.9 Few-shot prompt template 33
3.10 Chain-of-Thought prompt template 34

4.1 Comparison of performance scores across the different approaches for FILTER$_{\text{discussAPI}}$, each data point in the box plot represents an API and each box represents 19 APIs from the three libraries 40

5.1 Code example of `numpy.random.Generator.random` 46
5.2 Code example of `requests.auth.AuthBase` 46
5.3 A Stack Overflow post, where a developer is asking about a performance issue while using the *boto3.session.client* from the boto3 Python library 47
5.4 SO posts discussing APIs$_{\text{noeg}}$ 48
5.5 Result explanation of *numpy.ndarray.clip* 49

6.1 Example of a SO post where the user discusses their issue with API parameter 51
6.2 An SO post that discusses an API but does not imply the need of an example 51

Chapter 1

Introduction

Software libraries are collections of pre-written code that solve common problems of software development, making code more efficient, modular, and reusable. To make use of these libraries, developers use the offered Application Programming Interfaces (APIs) to interact with the library code. Developers learn about how to properly use these APIs from the documentation of these libraries, making software documentation an important part of a software library. However, the effectiveness of API documentation in communicating the proper use of APIs highly depends on the quality of the documentation. Good documentation can attract developers to use a particular library as understanding the usage of APIs becomes easy if documentation has all the needed information [1].

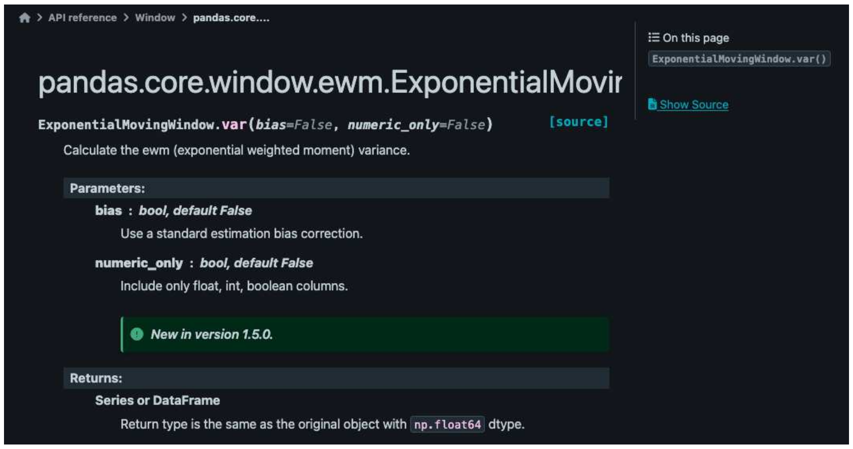
Specifically, previous research shows that if the documentation has enough example code showing how the APIs should be used, it becomes easier for the developers to work with a software library as they do not have to delve deep into the documentation to find out how to properly use the API, saving their time and effort [2]. However, if the software documentation does not have enough code examples, it makes the library APIs more difficult to use [3]. Thus, the existence of code examples is seen as one of the main positive quality attributes of software documentation. However, library authors may not consider updating the documentation a priority. Additionally, adding examples of all APIs may be tedious and/or infeasible. Thus, the thesis statement of this dissertation is as follows:

- **Thesis Statement:** A system that notifies library authors about potential places for adding code examples to their API documentation can encourage the library authors to improve the library's documentation quality.

The first part of this thesis statement requires us to devise a technique that can identify potential places for adding code examples. The literature shows that when faced with difficulty in using APIs, developers often explore other sources of information to get more clarification about API usage, like Question and Answer (Q&A) websites, such as Stack Overflow. Hence, Stack Overflow is often considered an alternate source of API documentation when the documentation itself is insufficient [4]. Based on these assumptions, researchers have even proposed methods to augment API documentation by using code samples from Stack Overflow [5], [6].

Figure 1.1 shows an example of a Stack Overflow post, where the user discusses their struggle to understand an API. Figure 1.1a shows the API documentation page of the `pandas.core.window.ewm.ExponentialMovingWindow.var` API. The documentation page does not have any examples of the API. The Stack Overflow post shows that the user is facing issues while trying to use this API, which suggests that having a code example for this API in the API documentation might have helped the user understand the API better.

Specifically, to operationalize the first part of our thesis statement, we use Stack Overflow as a proxy for identifying the problematic APIs that need code example(s). We hypothesize that if an API is missing examples and is frequently discussed on Stack Overflow, it signifies the need for code examples of that API. Based on this hypothesis, we design a technique that analyzes the API documentation of a given library to detect which APIs are missing code examples in their documentation. We then search for questions about these APIs in Stack Overflow. We use a Large Language Model (LLM) to detect if these questions discuss the usage of a particular API. We mark the APIs that have at least 3 posts that discuss that API and identify the opportunity of potentially improving the documentation quality by adding an example for



API reference > Window > pandas.core....

pandas.core.window.ewm.ExponentialMovir

```
ExponentialMovingWindow.var(bias=False, numeric_only=False)
```

[source]

Calculate the ewm (exponential weighted moment) variance.

Parameters:

bias : *bool, default False*

Use a standard estimation bias correction.

numeric_only : *bool, default False*

Include only float, int, boolean columns.

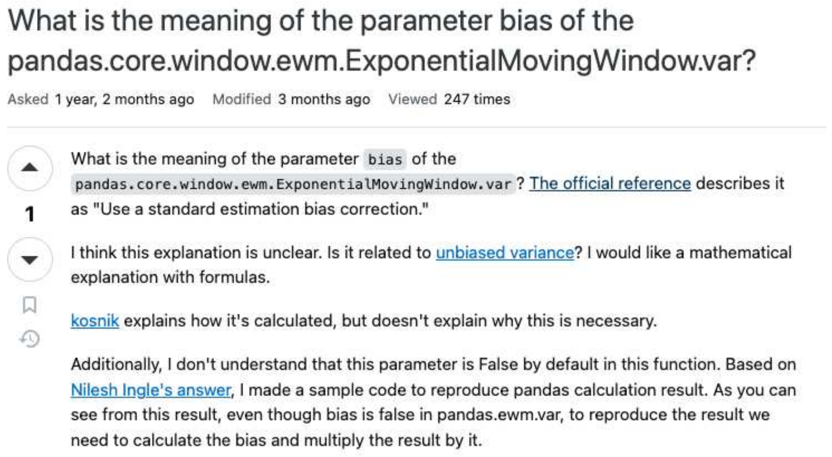
New in version 1.5.0.

Returns:

Series or DataFrame

Return type is the same as the original object with `np.float64` dtype.

(a) API documentation of `pandas.core.window.ewm.ExponentialMovingWindow.var`



What is the meaning of the parameter bias of the pandas.core.window.ewm.ExponentialMovingWindow.var?

Asked 1 year, 2 months ago Modified 3 months ago Viewed 247 times

1

What is the meaning of the parameter `bias` of the `pandas.core.window.ewm.ExponentialMovingWindow.var`? The official reference describes it as "Use a standard estimation bias correction."

I think this explanation is unclear. Is it related to unbiased variance? I would like a mathematical explanation with formulas.

kosnik explains how it's calculated, but doesn't explain why this is necessary.

Additionally, I don't understand that this parameter is False by default in this function. Based on Nilesh Ingle's answer, I made a sample code to reproduce pandas calculation result. As you can see from this result, even though bias is false in pandas.ewm.var, to reproduce the result we need to calculate the bias and multiply the result by it.

(b) SO post discussing the API shows the need for an example

Figure 1.1: Examples of an API documentation page of an api$_{\text{noeg}}$ and a SO post discussing the API

these APIs.

After verifying our technique using some manually annotated ground truth data, we proceed to operationalize the second part of our thesis statement. Specifically, our goal is to use our technique to identify potential improvements to documentation and then contact the corresponding library authors to notify them about these potential improvements. Our hypothesis is that when provided with a backup from Stack Overflow about how users are struggling with a given API, library authors may be more willing to update their documentation. Accordingly, we analyze the API documentation of 12 Python software libraries to detect which APIs are missing code examples in their documentation to contact the corresponding library authors with our findings.

Analyzing the 12 Python libraries using our technique, we did not find sufficient Stack Overflow posts for the APIs that are missing code examples that would lead us to contact the library authors. We therefore could not complete the second part of our thesis statement. We show the results of our technique in the later chapters of this thesis and discuss the possible reasons for these results in our findings.

1.1 Thesis overview

The thesis is divided into 6 chapters. Chapter 2 discusses the related literature, and establishes the importance of code examples in software documentation. We talk about previous studies that show the importance of documentation quality and how a lack of examples can make using library APIs difficult. We also talk about how Stack Overflow data is used in detecting and augmenting issues in API documentation, establishing the premise of our methodology. In Chapter 3, we describe our proposed methodology. We propose a pipeline that detects all the APIs in a Python library, identifies the ones that do not have code examples, and uses Stack Overflow to detect which of these APIs are causing difficulties for software developers. In Chapter 4, we verify the intermediate steps of our pipeline using a manually created ground truth. In Chapter 5, we evaluate the results obtained from analyzing 12 Python libraries using our prototype tool. We discuss future work and the limitations of our work in Chapter 6, followed by the threats to validity in Chapter 7. Finally, we conclude the thesis in Chapter 8.

The main contributions of this thesis are:

- A proposed methodology that can identify APIs from the documentation that are missing code examples

- A method to determine which APIs without code examples are creating difficulties for developers, by using data from Stack Overflow. We compare a heuristic-based approach and an LLM-based approach for this step.

- A tool written in Python that uses our proposed methodology to determine problematic library APIs of Python libraries from their API documentation.
- Evaluation of our prototype tool using manually created ground truth value to determine its performance.
- Evaluation of the results of our prototype tool when applied to 12 Python libraries.

Chapter 2

Literature Review

In this chapter, we discuss different research areas that are related to our thesis topics. First, we discuss the importance of good software documentation in the software documentation life-cycle. We then talk about the different documentation types and the type of documentation we focus on in this thesis (i.e., API documentation). Next, we discuss different quality aspects of software documentation based on previous studies, and establish the motivation behind our choice of focusing on the 'existence of code examples' aspect of API documentation quality in our thesis. Next, we discuss the use of Stack Overflow data to identify API usability problems. Finally, we discuss the usage of large language models (LLMs) to analyze Stack Overflow data. All these directions help to provide the necessary background and to motivate the design choices we make in this thesis.

2.1 Why Good Software Documentation is Important

Software documentation is an integral part of the software development and maintenance life cycle [7]–[9]. There have been several studies in the literature that highlight the importance of good software documentation. Parnas [10] discussed different scenarios stating the need for precise software documentation for program developers and maintainers. Parnas [10] argued that if a software is developed to be used by anyone other than a solo developer (i.e., other developers, maintainers, or client users), it needs documentation.

For developing trustworthy software, he mentioned easier reuse of old designs, better communication about requirements, more useful design reviews, easier integration of separately written modules, more effective code inspection, more effective testing, more efficient corrections and improvements, and ease of maintenance as the benefits of good documentation.

Lethbridge et al. [11] ran a survey with 48 software engineers, and concluded that software engineers use software documentation for (1) learning a software system, (2) testing a software system, (3) working with a new software, (4) solving problems when other developers are unavailable to answer questions and (5) looking for big-picture information about a software system. Moreover, documentation serves as a means of communication between the individuals involved in the production of a software, and also between library developers and client developers [12]. Hence, software documentation has important use cases in the software development process, and it should be of good quality for it to be useful for all stakeholders.

2.2 Types of Software Documentation

There can be different types of software documentation for the various stages of software development [13]. Among the different types of documentation are requirements, specifications, architectural/design, testing/quality, technical, and user documentation [13], [14]. Each documentation type can target a different audience and can be useful in different situations [10].

In our thesis, we narrow our scope to a software library's documentation, which targets the end users of the software library, i.e., the developers who are going to use the different publicly exposed APIs of the library (a.k.a *client developers*). Such library documentation is normally hosted on external websites, which are easily accessible and can provide information that helps a client developer understand and learn a library [8]. However, we focus on the parts of the documentation with respect to how their APIs are used, rather than other parts such as installation guide, updates of a library, etc.

Moreover, we limit ourselves to Python libraries and their documentation in this thesis. Documentation of a Python library can be written manually by the developers or can be generated. To make documentation easier, there are different documentation generation tools for Python libraries, such as Sphinx [15], pdoc [16], pydoctor [17], Doxygen [18], etc. Among these generation tools, Sphinx is the most popular and recommended generation tool that can convert reStructuredText [19] markup language into various output formats including HTML, LaTeX, etc. In this thesis, we focus on the Python library documentation that are generated by Sphinx.

2.3 Software Documentation Quality Aspects

Identifying ‘good’ vs ‘bad’ characteristics of documentation quality is a difficult task. However, several studies looked at what developers find useful in documentation and concluded several quality aspects of good documentation and issues in bad documentation.

In this thesis, our main focus is to improve documentation quality. We present the findings of previous studies that summarized the different quality aspects of documentation and also state the findings of different studies on the issues of software documentation. Finally, we show our motivation behind choosing one quality aspect that we focus on improving in this thesis.

Truede et al. [20] summarized 10 dimensions of software documentation quality based on previous studies through which one can assess documentation quality:

1. **Quality** - The spelling, grammar, and overall correctness of the documentation

2. **Appeal** - If the documentation is interesting to read

3. **Readability** - If the documentation is easy to read

4. **Understandability** - If the language of the documentation is understandable

5. **Structure** - If the documentation is well structured and easily navigable

6. **Cohesion** - If the text in the documentation makes sense when put together

7. **Conciseness** - The terseness of the information in the documentation

8. **Effectiveness** - If the vocabulary in the text is used effectively

9. **Consistency** - If the terminologies used in the text are consistent and do not vary

10. **Clarity** - There is no ambiguity in the text

Tang and Nadi [21] also summarized documentation quality aspects in their literature review and from the different aspects, they focused on (1) The existence of code examples, (2) Completeness, (3) Ease of Use, (4) Readability, and (5) Up-to-date in their study to evaluate software documentation quality.

While good documentation helps a client developer to better understand and learn a software, incorrect or inadequate documentation may lead to miscommunication and result in wrong decisions in the software development process. Uddin et al. [22] surveyed 69 employees related to software development (47 developers, 15 architects, three consultants, three managers, and one tester) and determined that incompleteness, ambiguity, obsoleteness, incorrectness, inconsistency, and unexplained examples were blockers when it came to understanding software documentation.

In another study [3], Robillard surveyed 83 software developers from Microsoft, Redmond, Washington to find out what aspects of documentation cause hindrance to learning APIs, and different API learning strategies. For the first part, he found that (1) lack of good examples, (2) incompleteness (3) lack of support for complex usage scenarios (4) bad organization, and (5) lack of relevant design elements were the main causes that make APIs hard to learn. The absence of code examples was an obstacle for more than 25% of the respondents. Also, for the second part of the survey on API learning strategies, of 80 respondents, 55% used code examples to learn APIs.

Aghajani et al. [23], discussed different issues in software documentation and mentioned wrong, outdated, and missing code examples as a few of the issues. The same issue was also identified by Sohan et al. [24] where they conducted a controlled study with 26 experienced software engineers to understand the problems faced by the client developers of REST APIs face while using APIs that lack code examples. They concluded that having code examples leads to higher satisfaction of the client developers while those APIs, with less development time and better success rate.

McLellan et al. [2] observed 20 programmers from Schlumberger, having several years to 20 years of experience, to determine how they use API documentation and interviewed them to understand what makes APIs more usable. One of the interviewed programmers found code examples as an important part of API documentation to show the context and the different ways an API can be used. They also found that the programmer uses the given code examples while using the APIs as a means of getting started.

All these studies highlight the importance of code examples in API documentation and show that lack of code examples makes an API less usable. This motivates us to focus on the 'existence of code examples' quality aspect of API documentation in our thesis.

2.4 Using Stack Overflow for Detecting Software Documentation Issues

In this section, we first mention the studies that showed how data from Stack Overflow can be an alternative source of documentation where people seek solutions for API-related problems and how studies used that data to improve documentation. We then present related work that looked at Stack Overflow to identify API usability problems. These served as a motivation for us to use Stack Overflow data in our study.

API documentation provides code examples to show how to use a certain API. Often when developers do not find the examples they are looking for in API documentation, they seek solutions elsewhere. A study by Treude et al. [25] described Stack Overflow as a good source of 'how-to' documentation, where answers to 'how-to' questions can be easily found. They also stated that

Stack Overflow would often become a substitute for official documentation if the documentation itself is inadequate. Beyer et al. [26] classified these 'how-to' questions as questions related to API usage which includes questions like 'how to use an API'. This motivates us to also look into Stack Overflow to find questions related to API usability.

Another study by Parnin and Treude [27] analyzed different documentation on the web to find out the extent to which the web resources (Blog posts, Q&A websites, forums, official doc, etc) cover the different methods of the API - 'jQuery'. They searched each API method on Google, e.g., 'jquery add', and analyzed the first 10 search results. They analyzed 1,730 search results and concluded that, among different web resources, Stack Overflow covered 84.4% of the API methods.

Multiple studies have used Stack Overflow posts for augmenting software documentation. A study by Treude and Robillard [6] extracted important sentences from Stack Overflow posts and used them to augment software documentation. They used a machine-learning approach to detect 'insight sentences' from Stack Overflow and surveyed GitHub users to verify if the sentences indeed add more meaning to the documentation. The results indicated that 47.5% of the participants found the detected sentences meaningful and that they added new information to the documentation.

Another study by Zhang et al. [5] used code examples from Stack Overflow to enrich API documentation. They point out how there is a scarcity of code examples in a lot of API documentation, and how Stack Overflow answers with a very high count of upvotes can be used to mitigate this scarcity. They introduced a new algorithm called ADECK (API Documentation Enrichment with Crowd Knowledge) which would detect code examples from Stack Overflow, cluster similar code examples, and embed the example clusters to the relevant API in the documentation. They designed a collection of programming tasks in which 21 developers participated to complete those tasks. The results showed that ADECK-enriched API documentation increased the productivity of developers. Similar to their work, Subramanian et al. [28] developed a method called Baker that can enhance API documentation by linking source code examples from Stack Overflow to the API in the documentation.

These studies show that Stack Overflow is used as an alternative source of documentation when the documentation does not have enough examples or developers fail to get adequate information on API from the documentation.

There are many studies that look into Stack Overflow to identify discussions about API-related aspects. Liu et al. [29] looked into API-related Stack Overflow posts to identify fine-grained developer needs expressed in sentences. They performed an empirical study on 266 Stack Overflow posts to create a taxonomy of developer needs in Stack Overflow posts, the information needed to express them (relevant information), and the different roles of the APIs in the post's questions and answers. They identified eight developer need types, which are (1) functionality implementation, (2) Non-Functional Improvement, (3) Functional Improvement, (4) Error Handling, (5) Rationale Analysis, (6) API Comparison, (7) Alternative Solution, and (8) API Usage Learning.

They also developed an approach that automatically identifies developer needs in Stack Overflow posts. They defined this task as a sentence classifier and trained multiple binary classifiers, one for each type of developer needs, and each type of relevant information. Each classifier classifies an input Stack Overflow question into two classes: 'Yes' or 'No', which determines if the sentence contains any of the types or not. To identify developer needs, they first identify API using their own implementation of Baker [28] for Java APIs. We use some of their techniques of API identification in our methodology. However, instead of training binary classifiers, we use an LLM to detect if a Stack Overflow post discusses an API.

Uddin et al. [4] proposed a framework that automatically mines API usage scenarios from Stack Overflow. The framework links code examples from the forum to the API mentioned in the textual contents of the forum post and generates a task description of the code example.

Another study by Wang et al. [30] mentions Stack Overflow as a source of information that could help API developers to design APIs according to developer needs. They present an approach that, among various other tasks, that recommends Stack Overflow posts related to API design issues. Similar to the Wang et al., Ahasanuzzaman et al. [31] also mentioned that Stack Overflow posts concerning API issues can help API developers/designers in both learning about the issues in their APIs and in solving those issues faster. To identify and separate the posts that discuss API issues from the other posts, they modeled the problem as a binary classification problem. They build a supervised learning technique that can identify API issue-related sentences from Stack Overflow posts. In addition to that, they used other Stack Overflow data to build a technique called CAPS that can classify Stack Overflow posts related to API issues.

Uddin and Khomh [32] studied API-related opinions of developers in Stack Overflow and concluded that Stack Overflow provides opinions about different aspects of an API. These opinions can affect other developers' perception of the API and the choices they make, such as, whether to use that API or not, how to use the API, etc. They developed a tool called Opiner that can detect opinionated sentences and associate those to the mentioned APIs. It can also detect API aspects, such as usability, and performance, that are discussed in the reviews of the posts.

All these studies show the vast usage of Stack Overflow data in identifying API-related aspects or issues. This motivated us to look into Stack Overflow data to find API usability-related posts to determine if developers struggle with using APIs that do not have examples in the documentation.

2.5 Using LLMs to analyze Stack Overflow data

Large language models (LLMs) are a category of foundation models trained on immense amounts of data making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks [33]. Since the introduction of LLMs, they have been used in software engineering research in various ways. Studies have pitted Stack Overflow against LLMs to determine which can assist software developers better. Kabir et al. [34] collected ChatGPT answers to 517 Stack Overflow questions. They manually analyzed the answers given by ChatGPT and compared them with the human answers for those questions from Stack Overflow. They found that 52% of ChatGPT answers contained misinformation, 77% of the answers were more verbose than human answers, and 78% of them had different degrees of inconsistencies.

Another study by Liu et al. [35] compared the performance of Stack Overflow and ChatGPT in assisting programmers and increasing their productivity in solving different programming-related tasks, such as algorithmic challenges, library usage, debugging, etc. Studies have also used ChatGPT API to create IDE plugins to assist software developers with code explanations. Chen et al. [36] introduced GPTutor, a Visual Studio Code extension that can help developers by providing code explanations. Similar to their work, Nam et al. [37] developed an IDE plugin that can explain programming code without any explicit prompts by the user. They used OpenAI's GPT-3.5-turbo model to build the plugin and answer API-related questions from a code snippet. These studies show the different ways LLMs have been used in research along with Stack Overflow data and also show the capability of LLMs in comprehending code snippets.

LLMs such as the different models of OpenAI have exceptional capabilities in understanding natural language from different contexts [38], [39]. As Stack Overflow posts are unstructured and written in natural language, we use OpenAI's GPT-3.5-turbo model [40] to comprehend and analyze the core topic of Stack Overflow posts. Studies have shown that the performance of LLMs can be improved by following different prompting techniques and giving proper instructions and examples [41]. Accordingly, we use the model off-the-shelf without any fine-tuning but experiment and evaluate its response by using different prompting techniques [42]. We finally select one prompting technique to build our methodology.

A study by Kocoń et al. [43] showed that ChatGPT is particularly good at binary classification tasks. They tested ChatGPT on 25 tasks that included simple binary classification of texts like spam, humor, sarcasm, aggression detection, or grammatical correctness of text. Moreover, they generate prompts to force ChatGPT to answer with a specified value. Similar to their work, we ask our model a binary question and create our prompt in a way that forces the model to answer either yes or no. More details of our methodology and prompt generation are described in Chapter 3.

Chapter 3

Methodology

Our goal is to (1) find library APIs that do not have code examples in the documentation and (2) determine which of these APIs users struggle to use, and as such, adding a code example for this API would improve the API documentation quality.

Below are some of the terms that we define for the APIs listed in a library's documentation.

- An API that is listed in the documentation: api_{doc}
- All the APIs that are listed in the documentation: $APIs_{doc}$
- An $\mathrm{api}_{\mathrm{doc}}$ that does not have any code example: api_{noeg}
- All the $\mathrm{APIs}_{\mathrm{doc}}$ that do not have any code example: $APIs_{noeg}$
- An $\mathrm{api}_{\mathrm{noeg}}$ that would benefit from additional code example(s): api_{needeg}
- All the $\mathrm{APIs}_{\mathrm{noeg}}$ that would benefit from additional code example(s): $APIs_{needeg}$

Our full pipeline, shown in Figure 3.1, follows the pipe and filter architecture. It is divided into 5 components. Each colored box represents each component with its name written on top. We name the components $\mathrm{FILTER}_{\mathrm{APIdoc}}$, $\mathrm{FILTER}_{\mathrm{APInoeg}}$, $\mathrm{FILTER}_{\mathrm{hasAPI}}$, $\mathrm{FILTER}_{\mathrm{discussAPI}}$, and $\mathrm{FILTER}_{\mathrm{prioritize}}$ and will refer to them by these names in the rest of the thesis.

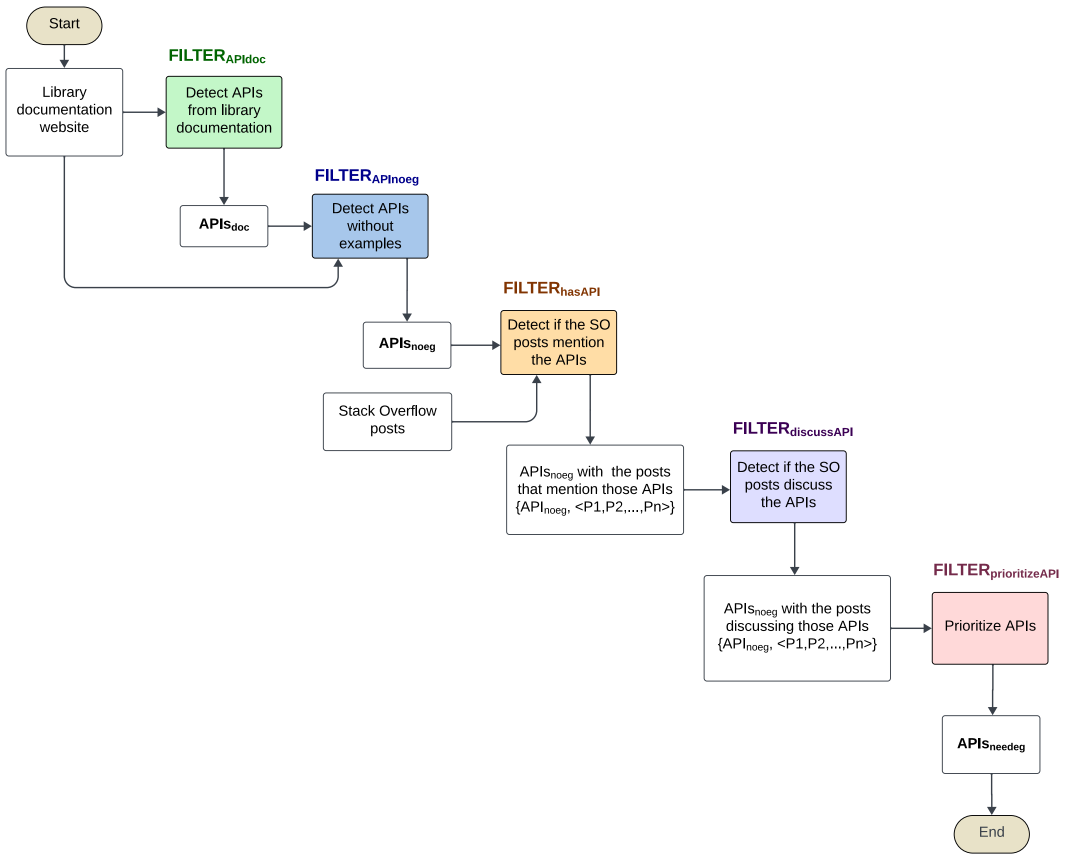
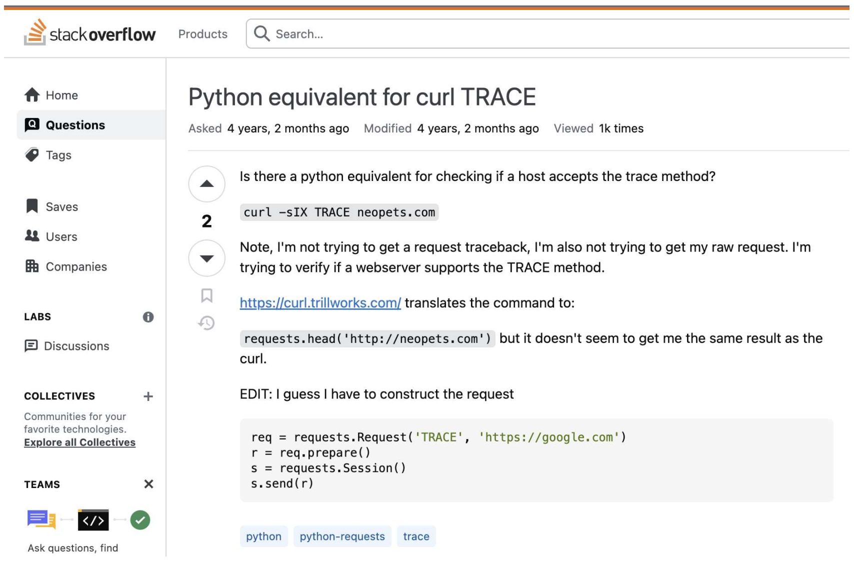

Start
Library documentation website
FILTER_APIdoc
Detect APIs from library documentation
APIs_doc
FILTER_APInoeg
Detect APIs without examples
APIs_noeg
FILTER_hasAPI
Detect if the SO posts mention the APIs
Stack Overflow posts
APIs_noeg with the posts that mention those APIs {API_noeg, <P1,P2,...,Pn>}
FILTER_discussAPI
Detect if the SO posts discuss the APIs
APIs_noeg with the posts discussing those APIs {API_noeg, <P1,P2,...,Pn>}
FILTER_prioritizeAPI
Prioritize APIs
APIs_needeg
End

Figure 3.1: The architecture of our system. Colored rectangles are the different processing components of our pipeline and the white rectangles show the input/output of each component.

We achieve our first goal through $\text{FILTER}_{\text{APIdoc}}$ and $\text{FILTER}_{\text{APInoeg}}$. For achieving the second goal, our intuition is that if a user struggles with an api_{noeg}, it indicates that having a code example for that api_{noeg} would benefit the user. Often when the documentation of an API is not sufficient, developers depend on online forums to learn an API [44]. Stack Overflow is a question-answering website where people ask questions about different programming-related problems that they face. We, therefore, look at Stack Overflow to find the posts related to the $\text{APIs}_{\text{noeg}}$ and assume that an api_{noeg} with more posts that discuss it suggests that more users struggle with that api_{noeg}. We use this assumption to prioritize the $\text{APIs}_{\text{noeg}}$ using $\text{FILTER}_{\text{prioritize}}$.

A user can mention an API in a Stack Overflow post for different reasons. An API can be a part of a code snippet, or it can be mentioned as a part of



Python equivalent for curl TRACE

Asked 4 years, 2 months ago Modified 4 years, 2 months ago Viewed 1k times

2

Is there a python equivalent for checking if a host accepts the trace method?

```
curl -sIX TRACE neopets.com
```

Note, I'm not trying to get a request traceback, I'm also not trying to get my raw request. I'm trying to verify if a webserver supports the TRACE method.

https://curl.trillworks.com/ translates the command to:

`requests.head('http://neopets.com')` but it doesn't seem to get me the same result as the curl.

EDIT: I guess I have to construct the request

```
req = requests.Request('TRACE', 'https://google.com')
r = req.prepare()
s = requests.Session()
s.send(r)
```

python python-requests trace

Figure 3.2: Example of a SO post that mentions but not discusses the API function `requests.head`

the steps that a user performs to achieve a different goal, where the main topic of the post is not the API of interest. When the main topic of discussion of a post is not the API of interest, we cannot use it as evidence that developers struggle with this API. Figure 3.2 is an example of a post where the user mentions `requests.head` as a part of the steps that they tried, but the main topic of discussion of this post is not `requests.head`. Accordingly, to find the posts where users discuss their struggles with using an API, we first find the Stack Overflow threads that mention that API using $\text{FILTER}_{\text{hasAPI}}$ and then find out if the core topic of the post is that API using $\text{FILTER}_{\text{discussAPI}}$.

Figure 3.1 shows the full pipeline of our technique. In the rest of this chapter, we explain the 5 parts of our pipeline in detail.

3.1 Finding APIs_{doc}: $\text{FILTER}_{\text{APIdoc}}$

Our goal in this step is to find all the listed APIs from a library's documentation website. The input of this filter is a library documentation URL, and the output is a list of APIs_{doc}.

In this section, we first describe the different features of library documentation websites that we consider and then describe the different steps of

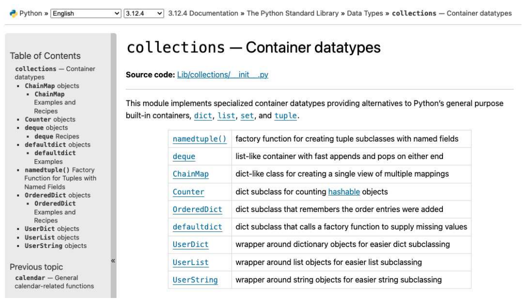

Python » English 3.12.4 3.12.4 Documentation » The Python Standard Library » Data Types » **collections** — Container datatypes

Table of Contents

- `collections` — Container datatypes
  - `ChainMap` objects
    - `ChainMap` Examples and Recipes
  - `Counter` objects
  - `deque` objects
    - `deque` Recipes
  - `defaultdict` objects
    - `defaultdict` Examples
  - `namedtuple()` Factory Function for Tuples with Named Fields
  - `OrderedDict` objects
    - `OrderedDict` Examples and Recipes
  - `UserDict` objects
  - `UserList` objects
  - `UserString` objects

Previous topic

`calendar` — General calendar-related functions

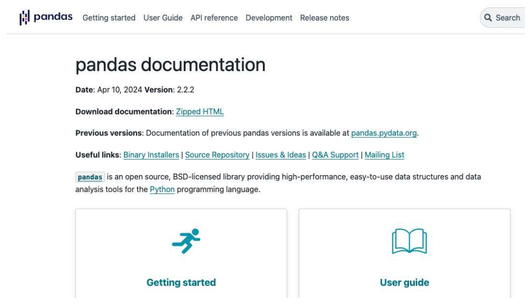
`collections` — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

(a) API Documentation of Collections[1]



pandas documentation

Date: Apr 10, 2024 **Version**: 2.2.2

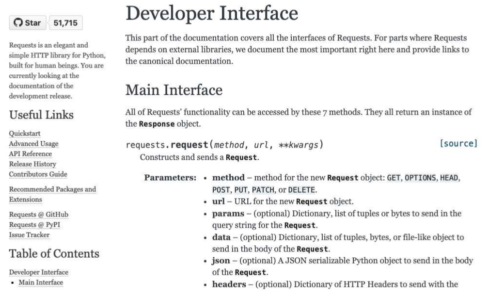
Download documentation: Zipped HTML

Previous versions: Documentation of previous pandas versions is available at pandas.pydata.org.

Useful links: Binary Installers | Source Repository | Issues & Ideas | Q&A Support | Mailing List

`pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

(b) Documentation of Pandas[2]



Requests is an elegant and simple HTTP library for Python, built for human beings. You are currently looking at the documentation of the development release.

Useful Links

Quickstart
Advanced Usage
API Reference
Release History
Contributors Guide

Recommended Packages and Extensions

Requests @ GitHub
Requests @ PyPI
Issue Tracker

Table of Contents

Developer Interface
- Main Interface

Developer Interface

This part of the documentation covers all the interfaces of Requests. For parts where Requests depends on external libraries, we document the most important right here and provide links to the canonical documentation.

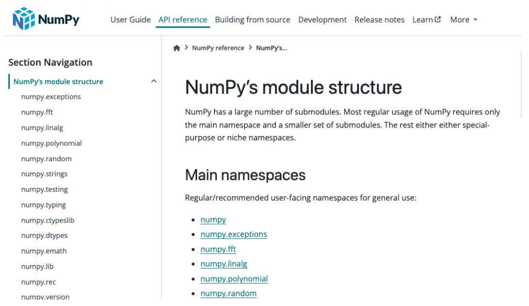
Main Interface

All of Requests' functionality can be accessed by these 7 methods. They all return an instance of the `Response` object.

```
requests.request(method, url, **kwargs)                    [source]
```
Constructs and sends a `Request`.

Parameters:
- **method** – method for the new `Request` object: `GET`, `OPTIONS`, `HEAD`, `POST`, `PUT`, `PATCH`, or `DELETE`.
- **url** – URL for the new `Request` object.
- **params** – (optional) Dictionary, list of tuples or bytes to send in the query string for the `Request`.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) A JSON serializable Python object to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the

(c) API Documentation of Requests[3]



Section Navigation

NumPy's module structure
numpy.exceptions
numpy.fft
numpy.linalg
numpy.polynomial
numpy.random
numpy.strings
numpy.testing
numpy.typing
numpy.ctypeslib
numpy.dtypes
numpy.emath
numpy.lib
numpy.rec
numpy.version

NumPy's module structure

NumPy has a large number of submodules. Most regular usage of NumPy requires only the main namespace and a smaller set of submodules. The rest either either special-purpose or niche namespaces.

Main namespaces

Regular/recommended user-facing namespaces for general use:

- numpy
- numpy.exceptions
- numpy.fft
- numpy.linalg
- numpy.polynomial
- numpy.random

(d) API Documentation of Numpy[4]

Figure 3.3: Different layouts of documentation websites

$\text{FILTER}_{\text{APIdoc}}$.

Most software libraries have a documentation URL, which is usually the landing page of the library, which we refer to as the documentation homepage. A library documentation usually has API reference page(s), where they specifically list their APIs, their parameters, and other information related to the APIs. There can be other pages, such as example page(s), getting started page, etc. Moreover, there can be different layouts for each library's documentation page. Some libraries have all the information in one single webpage, such as Python's collections[1] library (Figure 3.3a), and some libraries can have other page references linked in the homepage, such as a user guide page, getting started page, API reference page, etc. The pandas[2] library (Figure 3.3b) is an example of such a library. Moreover, some libraries have a single webpage as their API reference page, such as the requests[3] library (Figure 3.3c), whereas some libraries document their API references in multiple webpages, such as

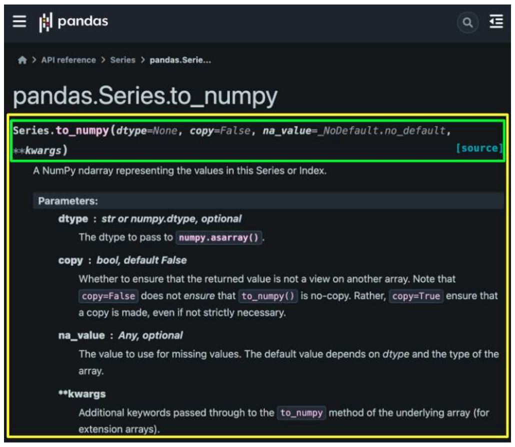
[1]https://docs.python.org/3/library/collections.html
[2]https://pandas.pydata.org/docs/
[3]https://requests.readthedocs.io/en/latest/api/

the numpy[4] library (Figure 3.3d). For extracting APIs_{doc}, we are mainly interested in the API reference page of a library's documentation. However, while looking for $\text{APIs}_{\text{noeg}}$, we look across the other pages of the documentation too.

Since libraries have different documentation layouts and URL formats, it is challenging to find a generalized way to automatically detect the URL for the API reference page(s). Going through the documentation websites of different libraries, we observe that there are certain keywords in the URL of most libraries' API reference pages, such as, 'api', 'reference', etc. We make a list of those keywords such that we can search for those words in a URL to determine if that URL belongs to the API reference section of a library's documentation. We also make a list of some stop words to look for in the URL so that we do not consider the URLs that contain one of those words, such as 'releasenotes', 'deprecated', 'updates', etc.

For this research, we limit ourselves to the documentation of Python libraries. As we discussed in the previous paragraph, different library documentation websites might have different layouts and it is hard to generalize all of them to extract necessary information from the websites. Sphinx [45] is a documentation generation tool for Python code. We find that the library documentation pages generated by Sphinx usually maintain a specific HTML format while documenting their APIs, so we focus on the documentation generated by Sphinx in this study. To explain the HTML format, we show an example of an API reference page and its corresponding HTML structure in Figure 3.4. We find that the API signature and its details are generally written inside a `<dl>` tag, which has a 'class' attribute that specifies the type of the API, such as function, method, class, attribute, or property. The yellow highlighted box in Figure 3.4b shows the `<dl>` tag. The `<dl>` tag has a `<dt>` tag inside of it with the attribute 'id' which contains the fully qualified name (FQN) of the API, shown in the green highlighted box in Figure 3.4b. We now describe how $\text{FILTER}_{\text{APIdoc}}$ uses the information in these tags to extract the list of APIs in the documentation, APIs_{doc}.

[4]https://numpy.org/doc/stable/reference/index.html



pandas.Series.to_numpy

```
Series.to_numpy(dtype=None, copy=False, na_value=_NoDefault.no_default,
**kwargs)                                                      [source]
```

A NumPy ndarray representing the values in this Series or Index.

Parameters:

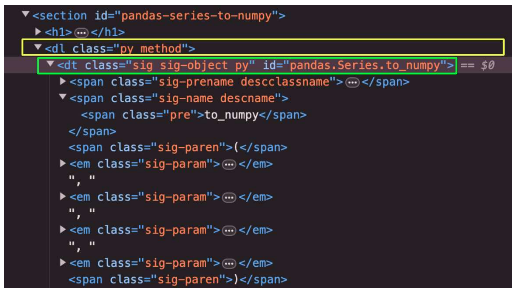
dtype : *str or numpy.dtype, optional*
The dtype to pass to `numpy.asarray()`.

copy : *bool, default False*
Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value : *Any, optional*
The value to use for missing values. The default value depends on *dtype* and the type of the array.

****kwargs**
Additional keywords passed through to the `to_numpy` method of the underlying array (for extension arrays).

(a) A portion of an API reference page



```
<section id="pandas-series-to-numpy">
  <h1>…</h1>
  <dl class="py method">
    <dt class="sig sig-object py" id="pandas.Series.to_numpy"> == $0
      <span class="sig-prename descclassname">…</span>
      <span class="sig-name descname">
        <span class="pre">to_numpy</span>
      </span>
      <span class="sig-paren">(</span>
      <em class="sig-param">…</em>
      ", "
      <em class="sig-param">…</em>
      ", "
      <em class="sig-param">…</em>
      ", "
      <em class="sig-param">…</em>
      <span class="sig-paren">)</span>
```

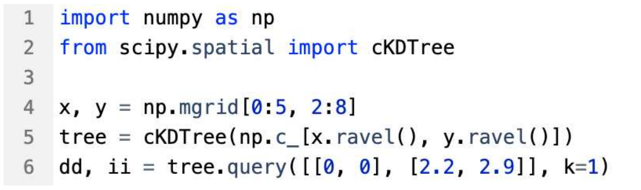
(b) A part of the corresponding HTML

Figure 3.4: Example of an API reference page and its corresponding HTML

- FILTER$_{\text{APIdoc}}$ takes the URL of the homepage of a library's documentation website. As described above, the homepage may contain references to the links of the other pages, such as the API reference page, user guide page, etc. FILTER$_{\text{APIdoc}}$ uses the Python library BeautifulSoup [46] to identify the URLs and only visits the URLs that contain one of the keywords identifying API reference pages and does not contain any of the stop words.

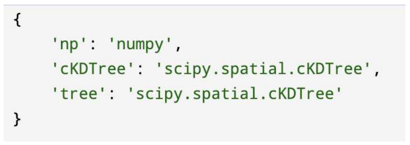
- For each API reference page, FILTER$_{\text{APIdoc}}$ looks for the `<dl>` tag to identify the type of the API. FILTER$_{\text{APIdoc}}$ only considers 'method', 'function', 'class', and 'exception' APIs. To retrieve the FQN of the API, it looks for the `<dt>` tag inside the `<dl>`.

- For extracting the different parameters of the API, it then uses Python's 'ast' (Abstract Syntax Trees) module to parse the API's signature, shown by the green highlighted box in Figure 3.4a. The ast module cannot parse some special characters like asterisk (*), and slash (/), which are sometimes present in the parameters of some APIs. FILTER$_{\text{APIdoc}}$ preprocesses the API signature to remove such characters and then parses the API signature to get the API's simple name, package name, and the required and optional parameters.



```python
import numpy as np
from scipy.spatial import cKDTree

x, y = np.mgrid[0:5, 2:8]
tree = cKDTree(np.c_[x.ravel(), y.ravel()])
dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=1)
```

(a) A code example from Scipy documentation [V1.13.1][5]



```
{
    'np': 'numpy',
    'cKDTree': 'scipy.spatial.cKDTree',
    'tree': 'scipy.spatial.cKDTree'
}
```

(b) Mapping of the variables to their FQN

Figure 3.5: Example of different ways of calling APIs

At the end of these steps, we have a list of all the APIs that appear in the library's API documentation, APIs_{doc}, along with their signature.

3.2 Finding $\textbf{APIs}_{\textbf{noeg}}$: $\textbf{FILTER}_{\textbf{APInoeg}}$

The input of this filter is the URL of the library's documentation website and the list of APIs_{doc} that we get from the previous filter, and the output is a list of $\text{APIs}_{\text{noeg}}$.

To find APIs that appear in a library's documentation but do not have code examples, i.e., $\text{APIs}_{\text{noeg}}$, we first find all the code examples in the documentation pages, and then find which of APIs_{doc} appear in those code examples. The APIs_{doc} that do not appear in any code example are the $\text{APIs}_{\text{noeg}}$. Below we describe the steps of $\text{FILTER}_{\text{APInoeg}}$ in detail.

- Unlike $\text{FILTER}_{\text{APIdoc}}$, $\text{FILTER}_{\text{APInoeg}}$ considers all the URLs of a library's documentation to look for code examples, except the ones that have any of the stop words in them. Although code examples can be written in different styles in different library documentation websites, we find that the documentation generated by Sphinx usually has each block of code examples inside a `<pre>` HTML tag. Similar to the approach of Zhong and Su [47] and Tang and Nadi [21], $\text{FILTER}_{\text{APInoeg}}$ looks for the `<pre>` tag to collect all the code examples.

- To identify an API call within a code block, $\text{FILTER}_{\text{APInoeg}}$ uses a simple heuristic of searching for open round parenthesis ("(") in each code block. However, an API is not always called using the FQN of the API.

There are different ways a function or class constructor can be called. For example, it can be called using a variable declared in a previous line of the code block or in the import statements. It can also be called using its simple name which was imported from a package of a library. For example, in Figure 3.5a[5], the class `cKDTree` is called in line number 5 using the simple name of the API that was imported in line number 2 from the package `scipy.spatial`. So `cKDTree` refers to the API `scipy.spatial.cKDTree`. Similarly, the variables 'np' and 'tree' are declared in lines 1 and 5 respectively. So, in line 4, `np.mgrid` actually refers to the API `numpy.mgrid` and in line 6, `tree.query` actually refers to the API `scipy.spatial.cKDTree.query`. To determine these APIs, $\text{FILTER}_{\text{APInoeg}}$ parses each code block and constructs a mapping between each variable that appears in the code and its FQN. Figure 3.5b show an example of the mapping constructed from the code block.

- After determining the called APIs, $\text{FILTER}_{\text{APInoeg}}$ matches the APIs from each code block with the FQN of the APIs_{doc}. If there are multiple matches, it compares the number of parameters to find the correct match of each called API.

- Lastly, $\text{FILTER}_{\text{APInoeg}}$ collects any APIs_{doc} that do not appear in any of the code blocks as the $\text{APIs}_{\text{noeg}}$.

3.3 Finding Stack Overflow posts that mention the $\text{APIs}_{\text{noeg}}$: $\text{FILTER}_{\text{hasAPI}}$

The input of this filter is the list of $\text{APIs}_{\text{noeg}}$ of a given library, and the Stack Overflow posts related to that library. The output is a map of $\text{APIs}_{\text{noeg}}$ with the Stack Overflow posts that mention those $\text{APIs}_{\text{noeg}}$.

$\text{FILTER}_{\text{hasAPI}}$ uses a matching heuristic to find if a Stack Overflow post mentions a certain API. It first fetches the posts from Stack Overflow using StackAPI [48]. We arbitrarily choose to get posts from the last 6 years (Jan

[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.query.html

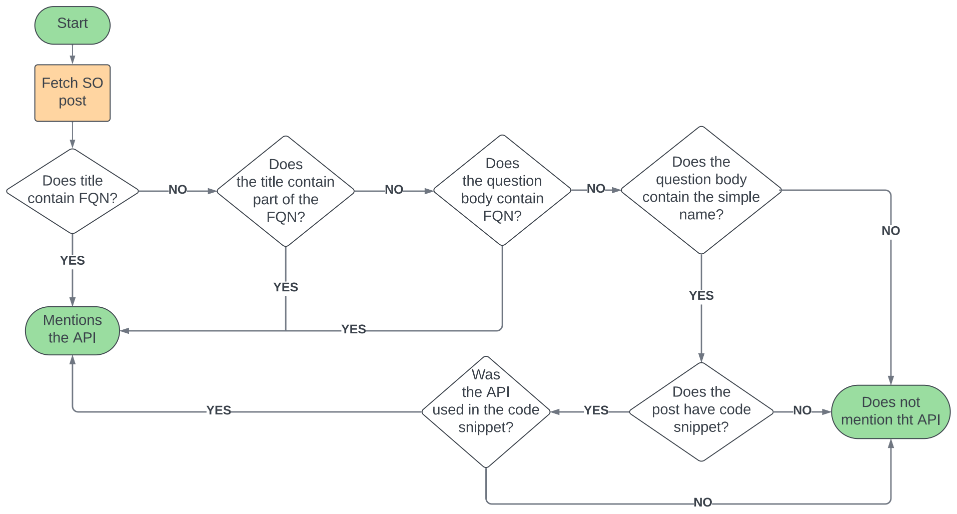

Start
Fetch SO post
Does title contain FQN?
NO
Does the title contain part of the FQN?
NO
Does the question body contain FQN?
NO
Does the question body contain the simple name?
NO
YES
YES
YES
YES
Mentions the API
Was the API used in the code snippet?
YES
Does the post have code snippet?
NO
Does not mention tht API
YES
NO

Figure 3.6: A flowchart of FILTER$_{\text{hasAPI}}$

2018 - June 2024) so that we do not get old posts that are no longer valid. We do not choose a smaller number so that we get enough posts per API for our pipeline to process. FILTER$_{\text{hasAPI}}$ uses a filter to get posts from the last 6 years and the ones that use the library-specific tags. We select the tags for each library by manually looking at the tags of several Stack Overflow posts related to each library. We select the most common tag that the users use for a library. For example, for the pandas library, we find that 'python' and 'pandas' are the two most common tags that users use, so we use the tag 'pandas;python' to fetch the posts related to the pandas library. StackAPI by default returns 500 Stack Overflow posts using its default parameter values. To get more data, FILTER$_{\text{hasAPI}}$ sets the parameters 'page_size' and 'max_pages' for the StackAPI endpoint to 100 and 400 respectively.

After fetching all the Stack Overflow posts for a library, FILTER$_{\text{hasAPI}}$ determines if any of the APIs$_{\text{noeg}}$ are mentioned in the posts. For each post, it only looks at the title and question body along with any code snippet provided in the question body. Figure 3.6 shows the flowchart of FILTER$_{\text{hasAPI}}$, and below we describe each step in detail. Note that each API has a fully qualified name (FQN). We refer to each part of the FQN as the *components* of the API, and the last part of the FQN as the *simple name* of the API. For example, `pandas.read_csv` is the fully qualified name, `pandas` and `read_csv` are the two components of the FQN and `read_csv` is the simple name.

FILTER$_{\text{hasAPI}}$ uses the following heuristics to determine if an API is mentioned in a post:

- If a post has a question title and question body, and does not have any code snippet, FILTER$_{\text{hasAPI}}$ first looks for any mention of the API in the post title; if not found there, it looks into the post's question body:
  - Search in post title:
    - If FILTER$_{\text{hasAPI}}$ finds the API's FQN in the post title, it concludes that the post mentions that API.
    - The FQN of an API can have multiple components. We find that SO post titles often mention APIs in a sentence with a space or other words in between each component. For example, 'how to get pandas get_dummies to emit N-1 variables to avoid collinearity?', here `pandas.get_dummies` is referred to with a space between the two components. Another example is 'Difference between groupby and pivot_table for pandas dataframes', here two APIs are mentioned, `pandas.DataFrame.groupby` and `pandas.DataFrame.pivot_table`. To identify such mentions, FILTER$_{\text{hasAPI}}$ splits the FQN into components and looks for all the components of an FQN in the title.
    - If an API has 2 or 3 components, FILTER$_{\text{hasAPI}}$ will look for all these components in the post. However, there are many APIs that have more than 3 components in their FQN. We design FILTER$_{\text{hasAPI}}$ such that that if an FQN has more than 3 components, it will conclude there is a match if there are at least 2 matched components. This is because, looking through multiple posts, we find that users often do not write all the component names for longer FQN. However, we find many examples where removing one component from a FQN that consists of 3

    components gives false results. For example, we want to distinguish between `pandas.DataFrame.pivot` and `pandas.pivot`. If FILTER$_{\text{hasAPI}}$ finds a match according to these heuristics, it says the post mentions the API.

  – Search in post question body:

    * FILTER$_{\text{hasAPI}}$ looks for the whole FQN either in the question body's text or in any of the ⟨*code*⟩ tags in the question body. It marks that the post mentions the API if it finds any such match.

- If one or more code snippets exist in the post body, then the API should be mentioned both in the paragraph and the code snippet for FILTER$_{\text{hasAPI}}$ to conclude that the post mentions that API. In this case, if the simple name appears in ⟨*code*⟩ tag or ⟨$href$⟩ tag in the paragraph, and it also appears in the code snippet, FILTER$_{\text{hasAPI}}$ marks it as a match. It uses regular expressions to look for the simple name of the API followed by the opening bracket ("(") to find the API call inside the code snippets.

We note that our above methodology for detecting if an API is mentioned in a Stack Overflow post is similar to a previous study by Liu et al. [29] that also detected APIs from SO posts. However, they look for any API mentions in a Stack Overflow post's question and answer, but in our case, we have a fixed list of APIs to search for in a Stack Overflow post's question, which makes our scope smaller. In their work, they look for Java APIs in text using (1) regular expressions to identify FQN, and method name (e.g., `String.split()`), (2) they look for texts wrapped in ⟨*code*⟩ tag. However, as we are not looking for generalized API names, we are not using regular expressions, but rather looking for a direct match of the API names.

3.4 Finding Stack Overflow posts that discuss the $\text{APIs}_{\text{noeg}}$: $\text{FILTER}_{\text{discussAPI}}$

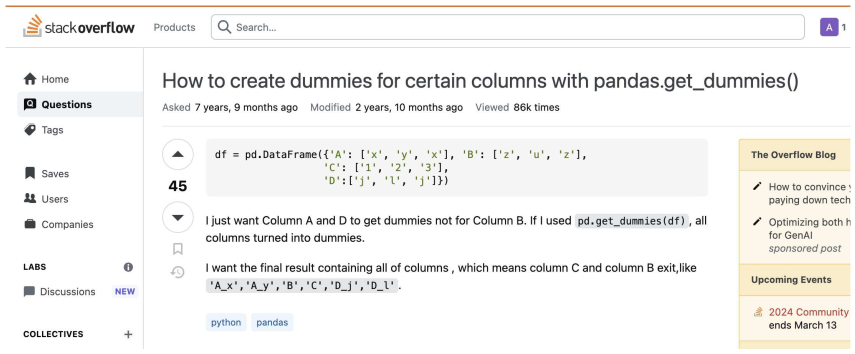
The input of this filter is a map of $\text{APIs}_{\text{noeg}}$ with the Stack Overflow posts that mention those $\text{APIs}_{\text{noeg}}$, and the output is a map of the $\text{APIs}_{\text{noeg}}$ with the Stack Overflow posts that discuss those $\text{APIs}_{\text{noeg}}$.

At this point, we have a list of posts that we know mention one of the $\text{APIs}_{\text{noeg}}$. Recall that we want to identify posts that may suggest that users struggle with using an API. However, as we discussed earlier, just because a post mentions an api_{noeg} does not necessarily mean that the core topic of the post is that API; it can be a part of a code snippet and might not be directly relevant to the main issue of the post. In such a case, we cannot say that the post discusses that api_{noeg} or conclude that users had problems using the API. Figure 3.7 shows examples of two Stack Overflow posts, where both posts mention the API: *pandas.get_dummies*, but the API is the core topic of the post in Figure 3.7a, whereas in Figure 3.7b, the main topic is something else, and the API is mentioned as a part of reproducing the problem that the user is facing. Given a Stack Overflow post and an api_{noeg}, the goal of $\text{FILTER}_{\text{discussAPI}}$ is to find if the user is struggling with the API of interest.

To find if a Stack Overflow post discusses a certain API, we explore two different methods: (1) a heuristic-based method and (2) an LLM-based method. In this section, we describe the details of these two methods. We later empirically compare the two methods in Section 4.3 to decide which one to use for our experiment on the 12 libraries.

3.4.1 Heuristic Based Approach

A text summary conveys the main ideas and important information from the original text in significantly less space [49]. With that in mind, our key insight in this method is that if we summarize the content of a Stack Overflow post, the summary should contain the core topic of the discussion. Therefore, if we find the API of interest (api_{noeg}) in the summary of the post, we can say that the core topic of the post is the api_{noeg}.



How to create dummies for certain columns with pandas.get_dummies()

Asked 7 years, 9 months ago Modified 2 years, 10 months ago Viewed 86k times

45

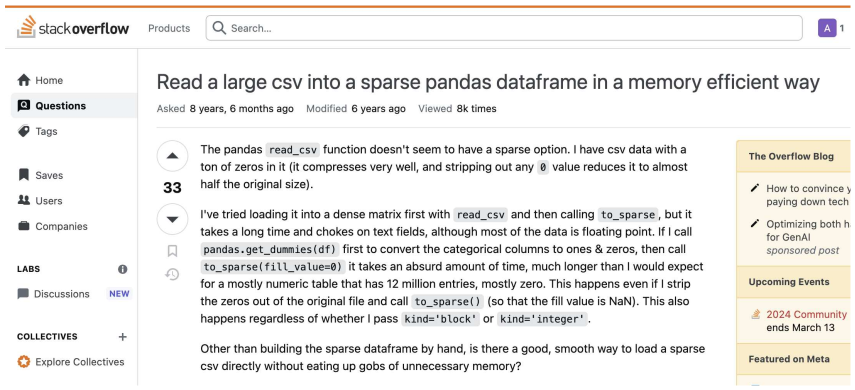
```
df = pd.DataFrame({'A': ['x', 'y', 'x'], 'B': ['z', 'u', 'z'],
                   'C': ['1', '2', '3'],
                   'D':['j', 'l', 'j']})
```

I just want Column A and D to get dummies not for Column B. If I used `pd.get_dummies(df)`, all columns turned into dummies.

I want the final result containing all of columns , which means column C and column B exit,like `'A_x','A_y','B','C','D_j','D_l'`.

python pandas

(a) Core topic is *pandas.get_dummies*



Read a large csv into a sparse pandas dataframe in a memory efficient way

Asked 8 years, 6 months ago Modified 6 years ago Viewed 8k times

33

The pandas `read_csv` function doesn't seem to have a sparse option. I have csv data with a ton of zeros in it (it compresses very well, and stripping out any `0` value reduces it to almost half the original size).

I've tried loading it into a dense matrix first with `read_csv` and then calling `to_sparse`, but it takes a long time and chokes on text fields, although most of the data is floating point. If I call `pandas.get_dummies(df)` first to convert the categorical columns to ones & zeros, then call `to_sparse(fill_value=0)` it takes an absurd amount of time, much longer than I would expect for a mostly numeric table that has 12 million entries, mostly zero. This happens even if I strip the zeros out of the original file and call `to_sparse()` (so that the fill value is NaN). This also happens regardless of whether I pass `kind='block'` or `kind='integer'`.

Other than building the sparse dataframe by hand, is there a good, smooth way to load a sparse csv directly without eating up gobs of unnecessary memory?

(b) Core topic is not *pandas.get_dummies*

Figure 3.7: Examples of SO posts

Extractive summarization and abstractive summarization are two text-summarization techniques. *Extractive summarization* picks the most important/relevant sentences and organizes them to form the summary, whereas *abstractive summarization* generates new sentences to convey the vital information from the text and forms the summary [50] [51]. In our case, we need to have the exact API in the summarized paragraph if that API is the core topic of the post, making extractive summarization more fit for our purpose. We chose one of the most popular extractive summarization algorithms - *LexRank* [52], from the Python library 'sumy', to create a summary of the post and look for the API of interest in the summary. We follow the following steps:

- For each post, we merge the post title and the text of the question body and create a summary.
- In the summary, we look for the fully qualified name of the API. If found, we conclude that the post discussed that API.

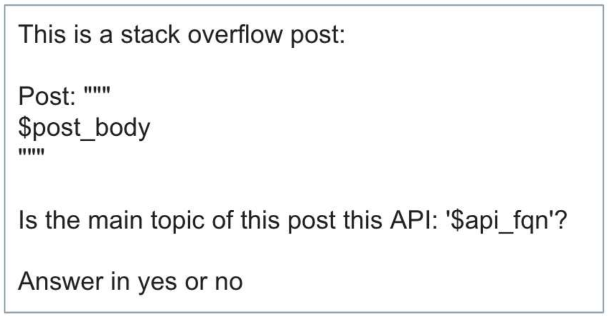
3.4.2 LLM Based Approach

We aim to find if a certain API is the key topic of discussion in a Stack Overflow post. Data on Stack Overflow is unstructured, so to find out the main topic of discussion of a Stack Overflow post, we not only need to process the text, but we also need to understand the semantics of the post, which is tedious to automate using regular expressions and heuristics. On the other hand, different large language models now have exceptional capabilities in natural language understanding (NLU) and can recognize semantic information from different contexts [38], [39]. We can use this capability of large language models (LLMs) to understand the semantics of Stack Overflow posts to find if a certain API is the core topic of discussion of the post.

Pre-trained LLMs such as GPT models [53] take instruction in natural language as a prompt or input. We can give them instructions, or ask them any question in natural language, and the models, understanding the question, provide a suitable answer to the question, which we can say is the model's response or output. The quality of the response highly depends on the quality of the prompt. We can improve their response by following different prompting techniques and giving proper examples and instructions [41]. Among the different LLMs, we choose to use OpenAI's GPT-3.5-Turbo [40] model for this study as this is fast and cost-efficient [54]. We use the OpenAI API from the `openai` Python package with temperature 0 to get more deterministic and focused results. We ask the model whether the core topic of a certain post is a certain API through a prompt, and process the model's response to get the result. Note that, all the results from the LLM model presented in this thesis were obtained by querying the GPT-3.5-Turbo model from January 2024 to June 2024.

To attain the best response from the model, we try three different prompting techniques:

- *Zero-shot Prompting*: This is a prompting technique to query the LLM without giving any labeled learning example [55]. In this technique, without giving any context, we give the post title followed by the post



```
This is a stack overflow post:

Post: """
$post_body
"""

Is the main topic of this post this API: '$api_fqn'?

Answer in yes or no
```

Figure 3.8: Zero-shot prompt template

question body within quotes to the model and ask if a certain API (FQN of the API) is the core topic of the given Stack Overflow post. Figure 3.8 shows the zero-shot prompt template that we use. Here, $api_fqn is the placeholder for the FQN of the API of interest, and $post_body placeholder contains the post title followed by the question body.

- *Few-shot Prompting*: This is a prompting technique to query the LLM by giving a few labeled examples to steer the model to perform better [56]. In this technique, we provide the model with two examples - first, we give one post (title plus question body) and the FQN of the API of interest, which is mentioned in the post but is not the core topic of the post. In this case, we ask the question - "Is the main topic of this post this API?", and also provide the answer - "no". Second, we give another post (title plus question body) and a certain API of interest, in this case, the API of interest is the core topic of the post. We ask the same question - "Is the main topic of this post this API?", and provide the answer - "yes". After giving these two examples, we ask the model to answer the same question for a new post, similar to the query of the zero-shot prompt. Figure 3.9 shows the template of the few-shot-prompt, and a template with the real example posts that we use in this thesis is shown in Section A.1. As we are asking a binary yes/no question, we aim to get better results from the model by providing one example from each case.

- *Chain-of-Thought Prompting* [57]: This is a prompting technique that rather than giving a straight answer as an example, describes the reasoning of the answer in steps [58]. Similar to few-shot prompting, we provide the model with two examples, but rather than giving their answers as a straight yes or no, we give a reason for each example, i.e - why we say the post discusses the API of interest for the first example, and why we say that the post does not discuss the API of interest for the second example. Lastly, we ask a similar question to the model with the new post body and API of interest like in the last two prompts. Figure 3.10 shows the prompt template for this prompting technique, and a template with the real post example that we use in this study is shown in Section A.2.

At the end of this step, for each $\text{APIs}_{\text{noeg}}$, $\text{FILTER}_{\text{discussAPI}}$ gives a list of Stack Overflow posts that discuss that API ({API1, {P11, P12, ... P1n }}). When we evaluate the intermediate steps of our pipeline in Section 4.3, we evaluate the above four approaches and compare their results.

3.5 Prioritize $\text{APIs}_{\text{noeg}}$: $\text{FILTER}_{\text{prioritize}}$

The input of this filter is a map of $\text{APIs}_{\text{noeg}}$ with the Stack Overflow posts that discuss those $\text{APIs}_{\text{noeg}}$, and the output is a list of $\text{APIs}_{\text{needeg}}$.

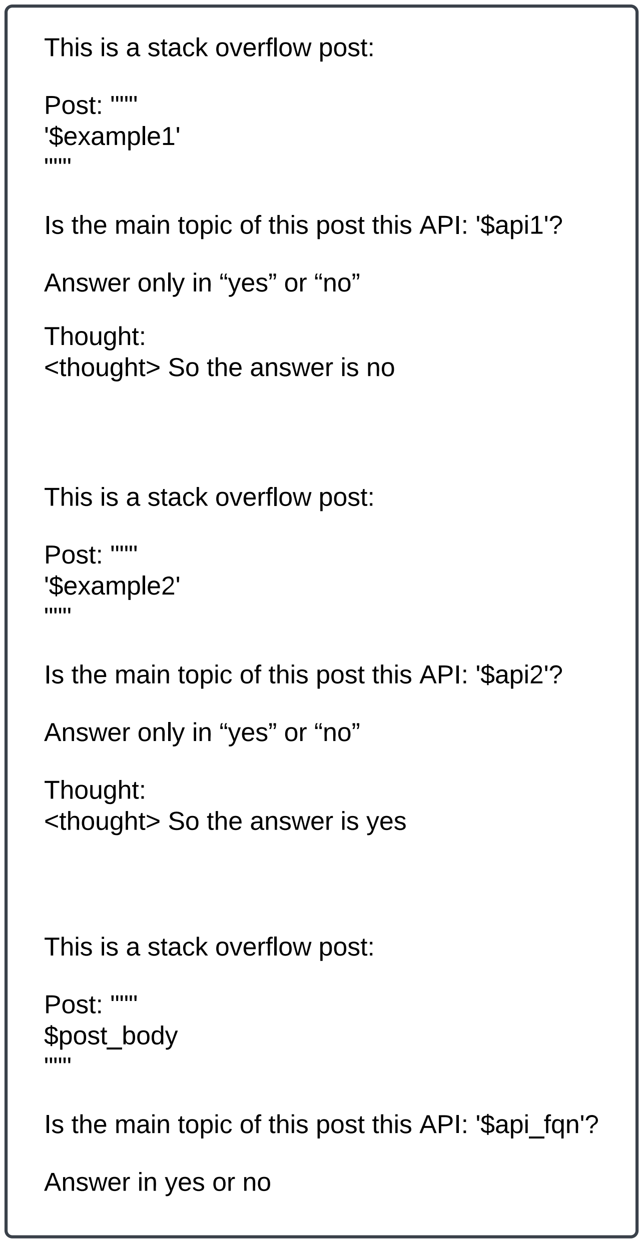
To find the $\text{APIs}_{\text{noeg}}$ that can benefit from additional code examples in their documentation, our intuition is that it should be frequently discussed in Stack Overflow. Stack Overflow contains many posts from newcomers [25], where the user might not know how to use an API. We want to filter out the one-off cases. While analyzing our pipeline for multiple libraries in the wild, we find that there are many $\text{APIs}_{\text{noeg}}$ that do not have any posts discussing them. For the remainder, it varies from 1 post per API to 100 posts per API, with a median of 2 posts per API. We choose to consider 1 more post per API than the median, that is, at least 3 posts discussing per API so that we can get rid of one-off cases and also have a sufficient number of posts to analyze of pipeline.

FILTER$_{\text{prioritize}}$ filters the APIs$_{\text{noeg}}$ that has at least 3 posts discussing them and gives a sorted list of APIs$_{\text{needeg}}$ by number of posts per API.

We create a prototype tool in Python implementing each step of the pipeline we describe here that we evaluate in the next two chapters.

```
This is a stack overflow post:

Post: """
'$example1'
"""

Is the main topic of this post this API: '$api1'?

Answer only in “yes” or “no”

Answer:
No



This is a stack overflow post:

Post: """
'$example2'
"""

Is the main topic of this post this API: '$api2'?

Answer only in “yes” or “no”

Answer:
Yes



This is a stack overflow post:

Post: """
$post_body
"""

Is the main topic of this post this API: '$api_fqn'?

Answer in yes or no
```

Figure 3.9: Few-shot prompt template



```
This is a stack overflow post:

Post: """
'$example1'
"""

Is the main topic of this post this API: '$api1'?

Answer only in “yes” or “no”

Thought:
<thought> So the answer is no



This is a stack overflow post:

Post: """
'$example2'
"""

Is the main topic of this post this API: '$api2'?

Answer only in “yes” or “no”

Thought:
<thought> So the answer is yes



This is a stack overflow post:

Post: """
$post_body
"""

Is the main topic of this post this API: '$api_fqn'?

Answer in yes or no
```

Figure 3.10: Chain-of-Thought prompt template

Chapter 4

Evaluation of Intermediate Pipeline Steps

In this chapter, we evaluate the intermediate steps of our pipeline described in Section 3 by running each step of our prototype tool. We answer the following research questions in this chapter:

- RQ1: How effectively can we detect APIs_{doc} from a library's documentation?
- RQ2: How effectively can we detect $\text{APIs}_{\text{noeg}}$ from a library's documentation?
- RQ3: How effectively can we detect if an api_{noeg} is the core topic of discussion of a Stack Overflow post?

4.1 Detecting APIs_{doc} from a documentation website: $\text{FILTER}_{\text{APIdoc}}$

In this section, we evaluate $\text{FILTER}_{\text{APIdoc}}$. Given an API reference page URL from a library's documentation website, the goal of this step is to extract all the APIs mentioned in that webpage.

Evaluation Setup

To evaluate $\text{FILTER}_{\text{APIdoc}}$, we first create a ground truth manually from a few pages of a library's documentation website and then run $\text{FILTER}_{\text{APIdoc}}$

for the same pages to evaluate the recall, precision, and f-measure score for its result. In this case, given a documentation page, a recall score tells us how many APIs the method correctly extracts from all the APIs that appear on the documentation page. A precision score tells us among all the APIs that our method extracts from the documentation page, how many of them are correct.

To evaluate this step, we choose one module from the API reference section of the Pandas library documentation (V2.31.0). Pandas has a large number of APIs in their API reference section (more than 1400), and API references of library documentation typically contain thousands of pages [47]. To get a feasible number of APIs for manual evaluation, we choose the module `pandas.Series`. Note that, the first page of this module, `https://pandas.pydata.org/docs/reference/series.html`, has the reference to all the API reference pages of the `pandas.Series` module. We analyze all these pages both for collecting APIs for the ground truth and running FILTER$_{\text{APIdoc}}$.

For the ground truth, we find 344 unique APIs from the different pages of the module `pandas.Series`. These 344 APIs include 'method', 'class', and 'property' type APIs. However, recall, FILTER$_{\text{APIdoc}}$ is designed to only consider methods and classes, so while manually collecting APIs for the ground truth, we discard the property APIs. We find 271 total APIs for our ground truth considering only methods and classes.

Evaluation Result: RQ1

We run FILTER$_{\text{APIdoc}}$ for all documentation pages from `pandas.Series` module. We find that FILTER$_{\text{APIdoc}}$ detects 270 APIs correctly out of the 271 total APIs. The recall, precision, and f-measure of FILTER$_{\text{APIdoc}}$ are 0.996, 1, and 0.998 respectively.

The only API that our algorithm fails to detect is `pandas.Series.to_dict`. Its signature in the documentation is as such: `Series.to_dict(*, into=<class 'dict'>)`. Our algorithm fails to parse the signature of this API as it includes the angle brackets.

4.2 Detecting APIs$_{\text{noeg}}$ from a documentation website: FILTER$_{\text{APInoeg}}$

In this section, we evaluate FILTER$_{\text{APInoeg}}$. Given a library's documentation page(s) that has code examples and a list of APIs$_{\text{doc}}$, the goal of this step is to extract all the APIs$_{\text{doc}}$ that do not appear in any of the code examples (APIs$_{\text{noeg}}$).

Evaluation Setup

To evaluate FILTER$_{\text{APInoeg}}$, we first create a ground truth manually from a library's documentation website page(s), which is the list of APIs$_{\text{noeg}}$, and run FILTER$_{\text{APInoeg}}$ for the same page(s) to evaluate the recall, precision, and f-measure for its result. In this case, given a library's documentation page(s), and a list of APIs$_{\text{doc}}$, a recall score tells us how many api$_{\text{noeg}}$ the method correctly identifies from the actual list of APIs$_{\text{noeg}}$. A precision score tells us among all the APIs$_{\text{noeg}}$ that the method identified, how many of them are correct.

To construct the ground truth for this step, we need to manually go through all the code examples of all the API documentation page(s). Accordingly, we choose a smaller documentation page to evaluate this step. We choose the API reference page from the request library's documentation website (V2.32.3) [6], which has all its API references on one page, which makes it feasible to evaluate manually. Additionally, evaluating another library gives us diversity showing that our pipeline steps work for different libraries.

We first use FILTER$_{\text{APIdoc}}$ to get the list of APIs$_{\text{doc}}$ from the above-mentioned page and then we manually go through the code examples of the page to find which of the APIs$_{\text{doc}}$ appear in the code examples. This gives us a list of APIs$_{\text{noeg}}$ for our ground truth. For the API reference page of the requests library, we get 112 APIs$_{\text{doc}}$ from the output of FILTER$_{\text{APIdoc}}$. We then manually identify 13 code examples and manually find 101 APIs$_{\text{noeg}}$ among the 112 APIs$_{\text{doc}}$. This serves as the ground truth for this step.

[6]https://requests.readthedocs.io/en/latest/api/

To evaluate FILTER$_{\text{APInoeg}}$, we provide it with the list of the 112 APIs$_{\text{doc}}$ and the API reference page. We expect a list of APIs$_{\text{noeg}}$ as the result, which we compare to the manually constructed ground truth above.

Evaluation Result: RQ2

We find that FILTER$_{\text{APInoeg}}$ detects 101 APIs$_{\text{noeg}}$. The recall, precision, and f-measure of FILTER$_{\text{APInoeg}}$ are 1, 1, 1 respectively.

4.3 Detecting Stack Overflow posts that discuss the APIs$_{\text{noeg}}$: FILTER$_{\text{discussAPI}}$

Evaluation Setup

In this section, we evaluate the four approaches that we described in Section 3.4 for FILTER$_{\text{discussAPI}}$. Given a Stack Overflow post, and an api$_{\text{noeg}}$, the goal of this step is to determine if the post discusses the APIs$_{\text{noeg}}$. To evaluate this, we first create a ground truth for a set of APIs and a set of Stack Overflow posts per API ({API1, {P11, P12, ... P1n }}, ...). For each data point (API, P), we determine if the post discusses that API, with a binary true/false answer. We then get the results for the same data points from FILTER$_{\text{discussAPI}}$ and calculate recall, precision, and f-measure for its results. In this case, given an API, recall is the number of posts a method identifies as discussing the API over the total posts in the ground truth that discuss the API. Precision is the number of posts a method correctly identifies as discussing the API over the total posts that the method identifies as discussing the API. Accuracy is the number of posts a method correctly identifies as discussing or not discussing the API over the total number of posts in the ground truth.

To create the ground truth for evaluating FILTER$_{\text{discussAPI}}$, we choose 10 APIs from the pandas library, 4 APIs from the requests library, and 5 APIs from the NumPy library. We choose the APIs in such a way that we have simple names of the APIs of both one-word (like `pandas.cut`, `pandas.merge`), and multiple-word (like `pandas.get_dummies`). Having a one-word simple name can be ambiguous with natural language, so to evaluate the effectiveness of

	Heuristic	**Zero-shot**	**Few-shot**	**Chain-of-Thought**
Recall	0.5	0.87	0.87	0.93
Accuracy	0.63	0.71	0.74	0.76
Precision	0.57	0.59	0.61	0.6
F-measure	0.53	0.67	0.73	0.72

Table 4.1: Performance scores (Medians) of the different approaches for $\text{FILTER}_{\text{discussAPI}}$

our techniques in mitigating the ambiguities, we choose to add those to our list.

We select a few Stack Overflow posts per API for our ground truth. For each API, we have an almost equal number of posts where the API appears and does not appear. We do this to reduce bias. To build our ground truth, we manually go through each of those posts and determine if the post discusses the API of interest.

We first select a subset of the data set and distribute it between the thesis author, the thesis author supervisor, and two external collaborators (a professor and an MSc student). Each data point is manually reviewed by two annotators We then calculate the agreement between the annotators and find that our Cohen's Kappa agreement coefficient is 0.84, which is nearly perfect agreement [59]. Accordingly, the thesis author proceeds to manually review the rest of the ground truth herself.

Evaluation Result: RQ3

We evaluate the heuristic-based approach and the LLM-based approach using three different prompting techniques by comparing their results with the ground truth that we created.

To evaluate, we calculate recall, accuracy, precision, and F-measure for the four approaches. Note that our goal is to determine if a Stack Overflow post discusses the API of interest so that we can consider that post as an indication that developers are struggling with that API. We want to choose a method that gives fewer false-positive values so that we can be confident that the posts we get per api_{noeg} in fact discuss the API. Therefore, in our case, having a good precision and accuracy score is more important than having a

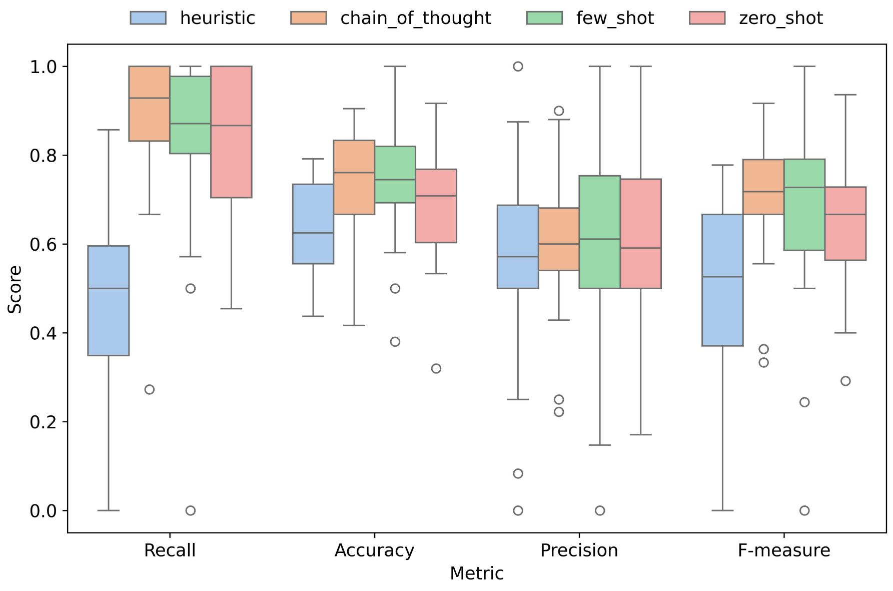


Figure 4.1: Comparison of performance scores across the different approaches for FILTER$_{\text{discussAPI}}$, each data point in the box plot represents an API and each box represents 19 APIs from the three libraries

good recall value.

We run the heuristic-based approach described in Section 3.4.1 and the LLM-based approaches described in Section 3.4.2 separately for FILTER$_{\text{discussAPI}}$ and evaluate their performance. For the LLM-based approach, we try the three prompting techniques for FILTER$_{\text{discussAPI}}$ that we discussed in the previous chapter. For the LLM-based approach, for each data point, FILTER$_{\text{discussAPI}}$ queries our prompt to the LLM eleven times and then chooses the majority answer from the eleven responses as the final answer. We chose to query the LLM eleven times because we wanted to minimize the non-determinism of the LLM model, and an odd number of queries makes it easy to determine the majority response. Later we see that for our scenario, the non-determinism of LLM is not very high (refer to the Appendix B), and only 3 queries to the LLM would be sufficient. Figure 4.1 shows a box plot of the performance scores for the 19 APIs in our ground truth across the four different approaches, and Table 4.1 shows the median scores of each metric. Looking at the scores, we see that the Few-shot Prompting technique of the LLM-based approach outweighs the other three techniques for precision and f-measure with median values of 0.61 and 0.73 respectively. We thus choose to use the LLM-based approach with few-shot learning for $\text{FILTER}_{\text{discussAPI}}$.

Chapter 5

Pipeline Evaluation

In this chapter, we apply our complete pipeline in the wild and analyze the results. A flowchart of our pipeline is shown in Figure 3.1. Recall, that given a library's API documentation, our pipeline determines which APIs of a library can benefit from adding code examples to their documentation. We want to run our pipeline on a few libraries and identify if any of the APIs need examples so that we can contact the library developers. To do so, we manually verify the results that our pipeline flagged as $\text{APIs}_{\text{needeg}}$ before contacting the developers.

5.1 Evaluation Setup

To evaluate our pipeline, we first choose libraries that we will analyze using our tool. We select libraries from a monthly dump of 80000 most downloaded PyPI packages [60]. The criteria we consider for choosing the libraries are: (1) the library has API documentation, (2) the documentation is of a specific format (the format of Sphynx generated documentation described in Section 3.1) and (3) it is actively maintained (has commits in at least last 3 months). The dump contained libraries with varying download counts from 1136326177 to 46921. We want to choose a stratified sample of this data from the top, mid, and lower ranges of download count so that we get libraries that are more popular and also the ones that are less popular. Our intuition behind this is that more popular libraries have greater chances to have good quality documentation than less popular ones. As the goal of our work is to find APIs for which we can improve library documentation quality, the chances of finding such APIs

Libraries	**Download count**	**APIs$_{\text{doc}}$**	**APIs$_{\text{noeg}}$**	**APIs$_{\text{noeg}}$ that have at least 1 SO post**	**APIs$_{\text{needeg}}$**
boto3	1136326177	88	76	0	0
botocore	522672231	38	35	1	0
urllib3	464629985	165	150	8	0
requests	385781127	110	84	9	2
typing-extensions	344505419	74	70	2	0
charset-normalizer	328506796	3	3	0	0
numpy	224195925	727	215	1	0
pandas	162319310	1436	776	14	1
scipy	82547255	622	306	3	1
opencensus	47059	70	67	1	0
desert	47055	8	3	0	0
dagster	46980	1071	859	1	0
Total				**40**	**4**

Table 5.1: Library statistics and results of running our pipeline

might differ for libraries of varying popularity. We start by selecting 5 libraries from each of those strata, if the libraries meet our criteria, we include them, if not, we do another iteration of the next 5 from each strata. We eventually did 3 iterations of the selections to get 12 libraries. The first column of Table 5.1 lists these 12 libraries.

5.2 Result

The first two columns of Table 5.1 show the names of the libraries and the download count of each library. The rest of the columns show the results of each step in our pipeline. Specifically, the number of APIs detected from their API documentation (APIs$_{\text{doc}}$), the number of APIs$_{\text{doc}}$ that do not have code examples in their documentation (APIs$_{\text{noeg}}$), the number of APIs$_{\text{noeg}}$ that have at least 1 Stack Overflow post that discusses them, and lastly, the number of APIs$_{\text{needeg}}$. After analyzing the 12 libraries, we find a total of 40 APIs$_{\text{noeg}}$ that have at least 1 SO post discussing them, and from FILTER$_{\text{prioritize}}$ we get 4 APIs$_{\text{needeg}}$, shown in the last column Table 5.2. We now manually go through the posts from the last two columns of Table 5.1 to critically judge our results.

APIs$_{\textbf{needeg}}$	**SO Post IDs**
pandas.pivot	77319901, 78159962, 77917091, 75901676, 75324700, 74172484, 73920596, 77331982, 76449353, 75554436, 74137744, 73966649, 73802333, 73691260, 73603656, 73347379, 73196573, 78052480, 75143012
requests.Response	67281120, 56442782, 68039452, 57312396, 70554533, 64773690
scipy.odr.odr	70406964, 55796398, 62460399, 71139423, 53048524, 62722907, 60889680, 77873145, 68370533
requests.ConnectionError	74253820, 64658898, 66137200, 28627162, 75720088, 76434691, 74613109, 70404894, 63933816, 62307838, 60200016, 74601518, 70272732

Table 5.2: Result of FILTER$_{\text{prioritize}}$

5.3 Manual Verification of Result

In this section, we manually analyze the results that we get for the 12 libraries. First, we go through all the Stack Overflow posts identified for the four APIs$_{\text{needeg}}$ and verify if any of those APIs actually need an example before contacting the library developers. We show the IDs of the corresponding Stack Overflow posts for each API in Table 5.2.

For the API `pandas.pivot`, we find that none of the identified posts actually discuss the API. The posts either discuss `pandas.pivot_table` or `pandas.DataFrame.pivot_table` or discuss `pandas.DataFrame.pivot`. The case for `scipy.odr.odr` is similar. The API `scipy.odr.odr` is a low-level API. The posts that we got from our pipeline discuss the APIs that belong to the package `scipy.odr` and our system could not distinguish between the two. The API `requests.Response` is a class, and it has multiple functions and properties. The posts that we find either discuss a property/function of the class, or talk about transforming the Response object into a different format. None of the posts particularly discuss having difficulty using the class `requests.Response`. Lastly, `requests.ConnectionError` is an error class. We find that the posts that our pipeline marks as discussing the error classes are mostly about an error that the user faced while using some other API, or how-to-resolve type questions. Having code examples is not really a solution for those error classes per se.

We thus conclude that none of the posts discussing the $\text{APIs}_{\text{needeg}}$ that our pipeline flagged indicates the need for a code example.

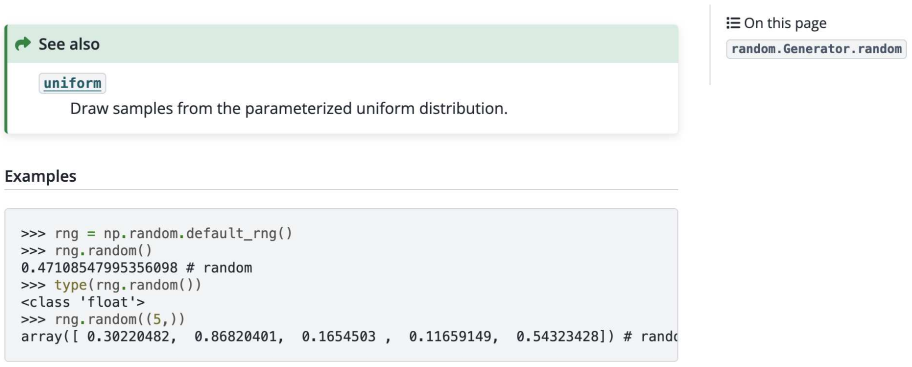
Since the final result of our pipeline was not promising, we want to take a step back and manually go through the results of $\text{FILTER}_{\text{discussAPI}}$. Recall that we get a list of $\text{APIs}_{\text{noeg}}$ and a list of Stack Overflow posts per API that discuss the $\text{APIs}_{\text{noeg}}$ from $\text{FILTER}_{\text{discussAPI}}$. We analyze the two parts of our results in two steps. First, we look at the $\text{APIs}_{\text{noeg}}$ and make sure that these do not have any code examples in their documentation that our pipeline failed to detect. Second, we verify the posts per $\text{APIs}_{\text{noeg}}$.

Validating that the APIs do not have any code examples

As we can see from the flowchart of our pipeline in Figure 3.1, the $\text{APIs}_{\text{noeg}}$ we find from $\text{FILTER}_{\text{APInoeg}}$ is passed to the next steps. Therefore, if $\text{FILTER}_{\text{APInoeg}}$ falsely marks an api_{noeg}, it will give a false-positive result at the end of the pipeline.

After manually analyzing the 40 $\text{APIs}_{\text{noeg}}$ that we got from $\text{FILTER}_{\text{discussAPI}}$, we see that 4 of them actually have code examples in their documentation that $\text{FILTER}_{\text{APInoeg}}$ could not detect. For example, `numpy.random.Generator.random` is one of the $\text{APIs}_{\text{noeg}}$ that our pipeline detected. However, it has a code example in the documentation shown in Figure 5.1. Here, the function `np.random.default_rng` returns `numpy.random.Generator`, which is mentioned in the API reference page of `numpy.random.Generator`. As $\text{FILTER}_{\text{APInoeg}}$ is not designed to consider that information, it fails to identify this API from the given code example.

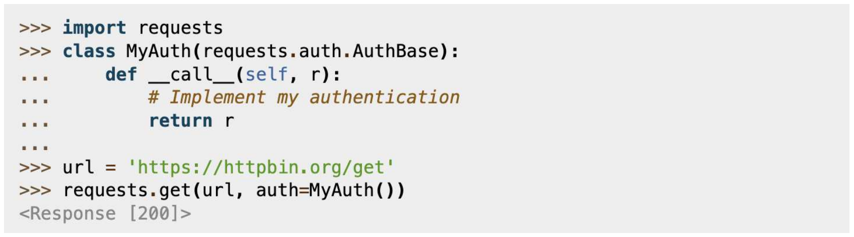
Another example is the API `requests.auth.AuthBase`. It is a base class that all authentication classes derive from. Figure 5.2 shows the example of this API, where this base class was inherited. As our implementation only looks for a direct call of the API of interest and does not take the inheritance chain into account, it could not identify this API from the example.



uniform
Draw samples from the parameterized uniform distribution.

```
>>> rng = np.random.default_rng()
>>> rng.random()
0.47108547995356098 # random
>>> type(rng.random())
<class 'float'>
>>> rng.random((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428]) # rando
```

Figure 5.1: Code example of `numpy.random.Generator.random`



```
>>> import requests
>>> class MyAuth(requests.auth.AuthBase):
...     def __call__(self, r):
...         # Implement my authentication
...         return r
...
>>> url = 'https://httpbin.org/get'
>>> requests.get(url, auth=MyAuth())
<Response [200]>
```

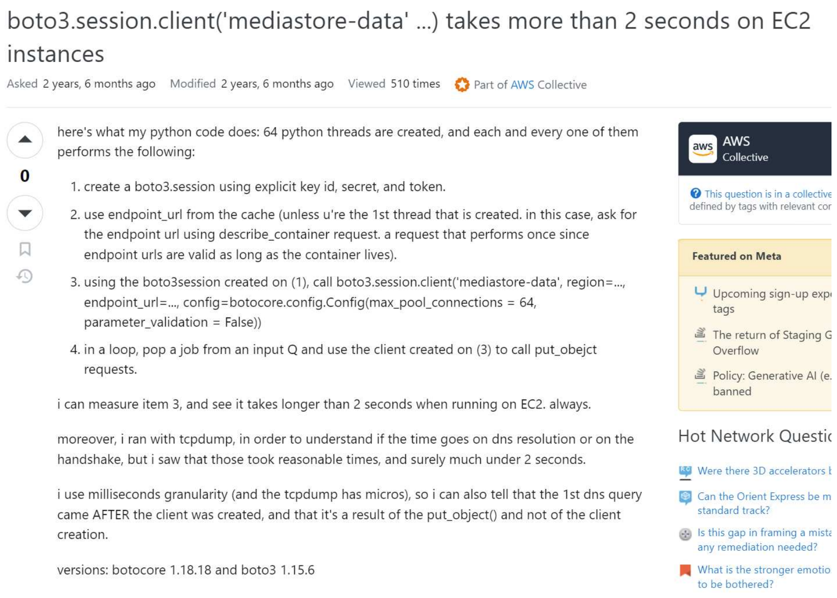
Figure 5.2: Code example of `requests.auth.AuthBase`

Validating the detected posts

Among the 40 $\text{APIs}_{\text{noeg}}$ that we get from $\text{FILTER}_{\text{discussAPI}}$, 4 actually have code examples, we already discussed the posts that discuss the 4 $\text{APIs}_{\text{needeg}}$ (prioritized APIs) at the beginning of this section, and 11 of these $\text{APIs}_{\text{noeg}}$ are error or exception classes. As we mentioned before, we conclude that the error class APIs would not benefit from additional code examples. For the remaining 21 APIs, we get a total of 24 Stack Overflow posts from $\text{FILTER}_{\text{discussAPI}}$. We manually go through these 24 Stack Overflow posts to determine (1) if they discuss the $\text{APIs}_{\text{noeg}}$ and (2) if they indicate that having a code example would benefit that user.

(1) We find 8 among the total 60 posts that actually discuss an $\text{APIs}_{\text{noeg}}$.

(2) Among the 8 posts, we found only 2 posts that we think discuss the user's struggle to use the API and having an appropriate code example would help them understand the API better. Figure 5.4 shows the 2 posts. The remaining 6 posts, although they discuss the API, going through them does not seem that having a code example for them would help the developers solve


boto3.session.client('mediastore-data' ...) takes more than 2 seconds on EC2 instances

Asked 2 years, 6 months ago Modified 2 years, 6 months ago Viewed 510 times Part of AWS Collective

0

here's what my python code does: 64 python threads are created, and each and every one of them performs the following:

1. create a boto3.session using explicit key id, secret, and token.
2. use endpoint_url from the cache (unless u're the 1st thread that is created. in this case, ask for the endpoint url using describe_container request. a request that performs once since endpoint urls are valid as long as the container lives).
3. using the boto3session created on (1), call boto3.session.client('mediastore-data', region=..., endpoint_url=..., config=botocore.config.Config(max_pool_connections = 64, parameter_validation = False))
4. in a loop, pop a job from an input Q and use the client created on (3) to call put_obejct requests.

i can measure item 3, and see it takes longer than 2 seconds when running on EC2. always.

moreover, i ran with tcpdump, in order to understand if the time goes on dns resolution or on the handshake, but i saw that those took reasonable times, and surely much under 2 seconds.

i use milliseconds granularity (and the tcpdump has micros), so i can also tell that the 1st dns query came AFTER the client was created, and that it's a result of the put_object() and not of the client creation.

versions: botocore 1.18.18 and boto3 1.15.6

AWS
Collective
This question is in a collective
defined by tags with relevant cor
Featured on Meta
Upcoming sign-up exp
tags
The return of Staging G
Overflow
Policy: Generative AI (e.
banned
Hot Network Questio
Were there 3D accelerators
Can the Orient Express be m
standard track?
Is this gap in framing a mista
any remediation needed?
What is the stronger emotio
to be bothered?

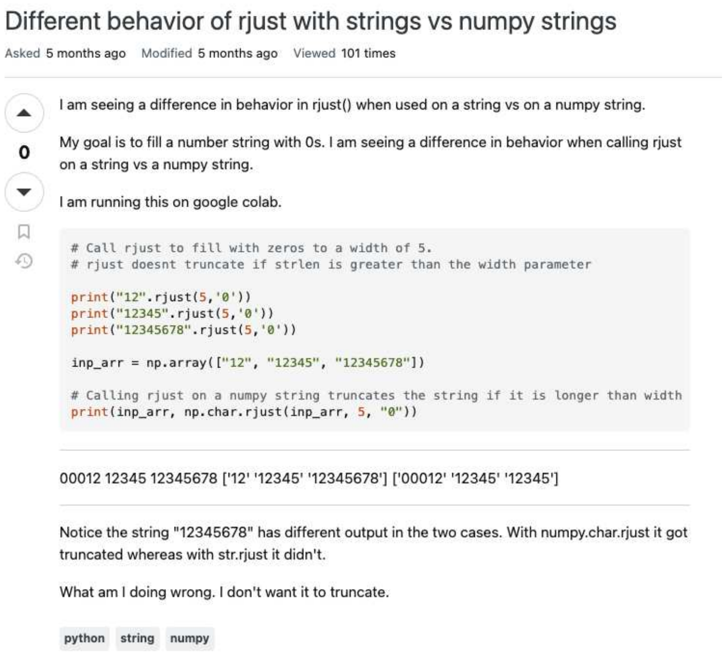
Figure 5.3: A Stack Overflow post, where a developer is asking about a performance issue while using the *boto3.session.client* from the boto3 Python library

their issues.

Going through those 6 posts, we find two types of results.

1. The post discusses the APIs$_{\text{noeg}}$, but the documentation of the API mentions looking at another alternate API from a different package. The alternate API has code example(s). Figure 5.5 shows an example of this scenario, where `numpy.ndarray.clip` was one of the APIs$_{\text{needeg}}$. (Figure 5.5a). The documentation page of `numpy.ndarray.clip` mentions to refer to `numpy.clip`, which is an alternate API for `numpy.ndarray.clip`, where it has examples of `numpy.clip` (Figure 5.5b and 5.5c)

2. The post discusses the API, but, we do not think that having a code example would help the user solve their issue. For example, Figure 5.3 is a post that discusses the API `boto3.session.Session.client`. However, the post suggests that the user knows how to use the API, but faced a performance issue while using the API. Having a code example of the API might not help to resolve this issue.


Different behavior of rjust with strings vs numpy strings

Asked 5 months ago Modified 5 months ago Viewed 101 times

0

I am seeing a difference in behavior in rjust() when used on a string vs on a numpy string.

My goal is to fill a number string with 0s. I am seeing a difference in behavior when calling rjust on a string vs a numpy string.

I am running this on google colab.

```
# Call rjust to fill with zeros to a width of 5.
# rjust doesnt truncate if strlen is greater than the width parameter

print("12".rjust(5,'0'))
print("12345".rjust(5,'0'))
print("12345678".rjust(5,'0'))

inp_arr = np.array(["12", "12345", "12345678"])

# Calling rjust on a numpy string truncates the string if it is longer than width
print(inp_arr, np.char.rjust(inp_arr, 5, "0"))
```

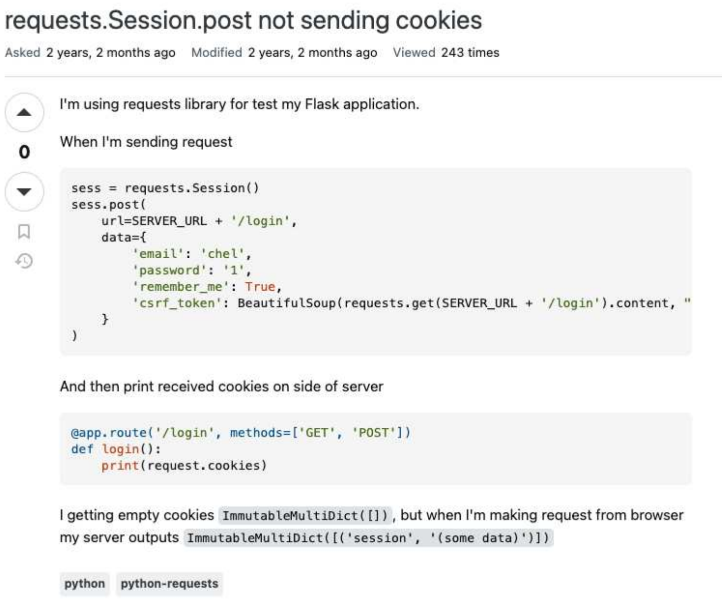
00012 12345 12345678 ['12' '12345' '12345678'] ['00012' '12345' '12345']

Notice the string "12345678" has different output in the two cases. With numpy.char.rjust it got truncated whereas with str.rjust it didn't.

What am I doing wrong. I don't want it to truncate.

python string numpy

(a) Post discussing *numpy.char.rjust*


requests.Session.post not sending cookies

Asked 2 years, 2 months ago Modified 2 years, 2 months ago Viewed 243 times

0

I'm using requests library for test my Flask application.

When I'm sending request

```
sess = requests.Session()
sess.post(
    url=SERVER_URL + '/login',
    data={
        'email': 'chel',
        'password': '1',
        'remember_me': True,
        'csrf_token': BeautifulSoup(requests.get(SERVER_URL + '/login').content, "
    }
)
```

And then print received cookies on side of server

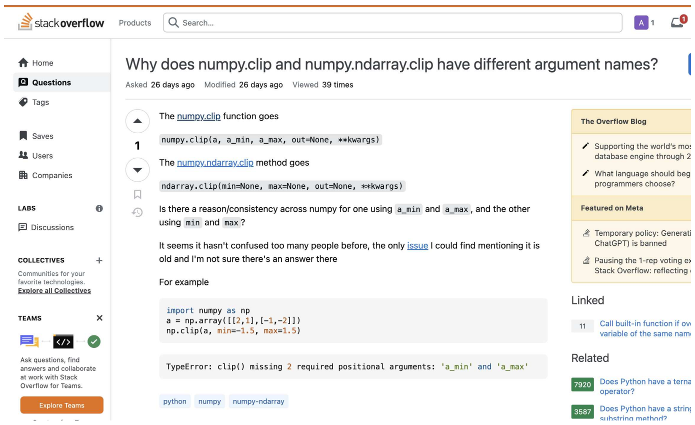
```
@app.route('/login', methods=['GET', 'POST'])
def login():
    print(request.cookies)
```

I getting empty cookies `ImmutableMultiDict([])`, but when I'm making request from browser my server outputs `ImmutableMultiDict([('session', '(some data)')])`

python python-requests

(b) Post discussing *requests.Session.post*

Figure 5.4: SO posts discussing APIs$_{\text{noeg}}$



Why does numpy.clip and numpy.ndarray.clip have different argument names?

Asked 26 days ago Modified 26 days ago Viewed 39 times

1

The numpy.clip function goes

`numpy.clip(a, a_min, a_max, out=None, **kwargs)`

The numpy.ndarray.clip method goes

`ndarray.clip(min=None, max=None, out=None, **kwargs)`

Is there a reason/consistency across numpy for one using `a_min` and `a_max`, and the other using `min` and `max`?

It seems it hasn't confused too many people before, the only issue I could find mentioning it is old and I'm not sure there's an answer there

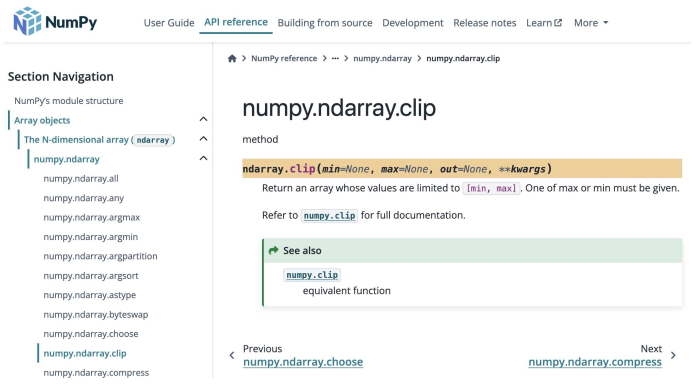
For example

```
import numpy as np
a = np.array([[2,1],[-1,-2]])
np.clip(a, min=-1.5, max=1.5)
```

```
TypeError: clip() missing 2 required positional arguments: 'a_min' and 'a_max'
```

python numpy numpy-ndarray

(a) Post discussing *numpy.ndarray.clip*



numpy.ndarray.clip

method

`ndarray.clip(min=None, max=None, out=None, **kwargs)`

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

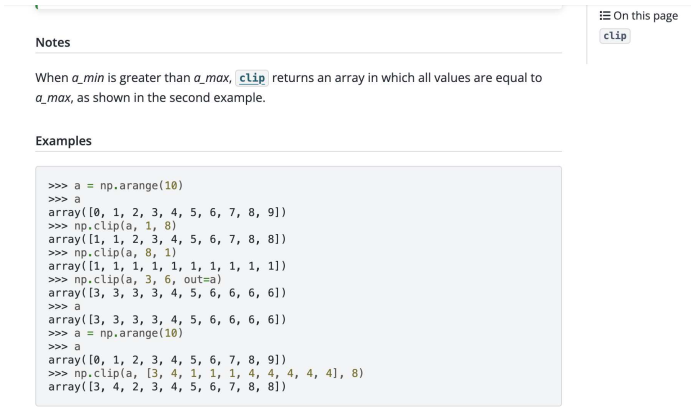
Refer to `numpy.clip` for full documentation.

See also

`numpy.clip`
equivalent function

Previous numpy.ndarray.choose Next numpy.ndarray.compress

(b) *numpy.ndarray.clip* documentation



Notes

When *a_min* is greater than *a_max*, `clip` returns an array in which all values are equal to *a_max*, as shown in the second example.

Examples

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> np.clip(a, 8, 1)
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

(c) Code example of *numpy.clip*

Figure 5.5: Result explanation of *numpy.ndarray.clip*

Chapter 6

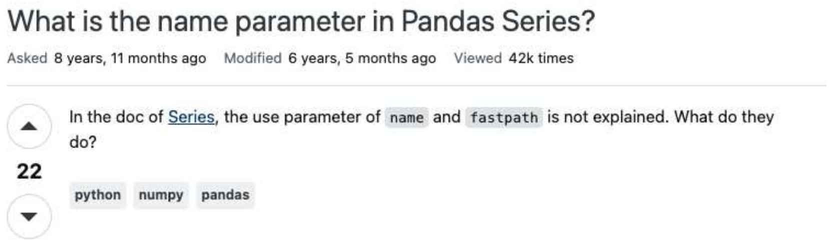
Discussion

In this chapter, we discuss the implications of our results, the possible shortcomings of our approach and implementation, and potential future work to improve the results.

The goal of our work was to find the places in a library's API documentation that can be improved by adding code examples. To find the poorly documented APIs, we chose to look for the APIs that do not have any code examples in the documentation ($\text{APIs}_{\text{noeg}}$), and then we looked at Stack Overflow data to find posts discussing those $\text{APIs}_{\text{noeg}}$. We wanted to finally convey the results that we got from our pipeline to the respective library's authors so they can use them to update their documentation. We created a prototype tool implementing our approach and evaluated the intermediate steps of our tool using our manually created ground truth and had good performance scores. However, when we ran our pipeline on 12 libraries, we could not identify any APIs that we would want to contact the library authors about.

Below we point out the possible reasons for our negative results:

- To find places to improve, we only look at $\text{APIs}_{\text{noeg}}$. We do not consider the APIs that already have examples and do not consider the possibility that the already documented code examples might not cover all the possible ways the API could be used. For example, Figure 6.1 shows a Stack Overflow post where the user is facing issues understanding the parameters 'name' and 'fastpath' of the API `pandas.Series` that are not well documented. Although the API has code examples, it does not


What is the name parameter in Pandas Series?

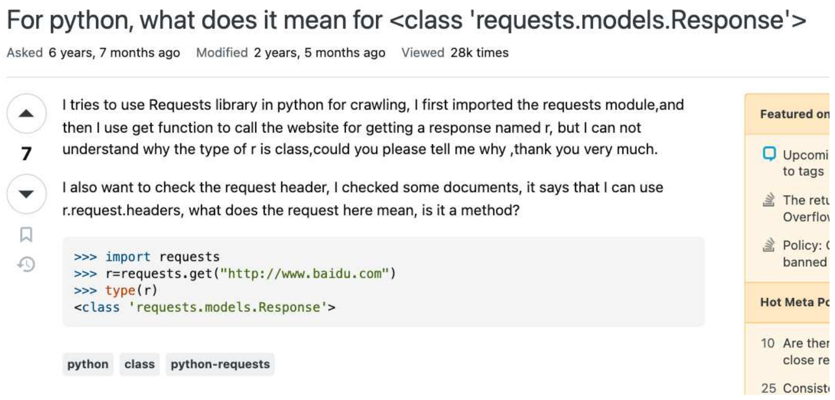
Asked 8 years, 11 months ago Modified 6 years, 5 months ago Viewed 42k times

In the doc of Series, the use parameter of `name` and `fastpath` is not explained. What do they do?

22

python numpy pandas

Figure 6.1: Example of a SO post where the user discusses their issue with API parameter


For python, what does it mean for <class 'requests.models.Response'>

Asked 6 years, 7 months ago Modified 2 years, 5 months ago Viewed 28k times

I tries to use Requests library in python for crawling, I first imported the requests module,and then I use get function to call the website for getting a response named r, but I can not understand why the type of r is class,could you please tell me why ,thank you very much.

I also want to check the request header, I checked some documents, it says that I can use r.request.headers, what does the request here mean, is it a method?

```
>>> import requests
>>> r=requests.get("http://www.baidu.com")
>>> type(r)
<class 'requests.models.Response'>
```

7

python class python-requests

Figure 6.2: An SO post that discusses an API but does not imply the need of an example

have code examples showing all the optional parameters of the function.

- While looking for Stack Overflow posts for a certain API, our pipeline found some posts that do not imply the need for a code example. Figure 6.2 shows an example of such a post, where the user discusses the API `requests.Response`. Some users are new to or exploring certain libraries and ask questions regarding their confusion or to better understand an API. Such posts, though they are discussing a certain API, do not indicate that having code examples would benefit them. Therefore we decided not to contact the authors with such posts. This also shows that a post discussing an API does not always imply that the API needs code example(s).

- We assumed in our study that if a Stack Overflow post discusses an API, it indicates there is an API usability problem. We thus ask the

LLM if a given post is discussing a certain API. (Full prompt is shown in Figure 3.9). However, as we discussed in the previous point, there are Stack Overflow posts that discuss certain APIs but do not imply the need for a code example. It shows that discussing a certain post does not always indicate an API usability problem, and thus does not always indicate that adding a code example for that API would be beneficial.

- We designed our approach to consider the $\text{APIs}_{\text{noeg}}$ that have at least 3 posts. Therefore, although we found one Stack Overflow post for some of the $\text{APIs}_{\text{noeg}}$, we did not include those APIs in the list of $\text{APIs}_{\text{needeg}}$, for which we got a negative result. We still think that prioritizing APIs was a good design choice as in that way we can filter the one-off cases.

Future opportunities

We analyzed our tool on a small dataset and for only 12 libraries. In the future, this tool could be used to analyze more libraries with more Stack Overflow posts which might give more positive results.

We chose the libraries such that we have libraries of varying popularity in our list. However, we observed that most of the more popular libraries in our list are already well-documented with less opportunity for improvement. On the other hand, less popular libraries do not have many questions in Stack Overflow discussing their APIs. Work can be done to find libraries that are popular but not well documented.

Our current implementation is limited to Sphinx-generated documentation of Python libraries. In the future, work can be done to generalize the implementation so that it can analyze other different layouts of API documentation, and also the documentation of libraries from other languages. In that way, more diverse library documentation can be analyzed using our approach.

Moreover, we designed our study under the hypothesis that if a Stack Overflow post discusses an API, it implies the need for a code example. However, our results show that it is not always true. Therefore, an area of improvement would be to redesign the prompt to ask the LLM a more specific question to identify if a Stack Overflow post is about an API's usability problem.

Chapter 7

Threats to validity

7.1 Internal Validity

In our prototype tool, we limit ourselves to only Python library API documentation generated by Sphinx. This limits the number of libraries that can be assessed using our tool. Our tool is not designed to identify decorator functions, which might yield wrong results for $\text{APIs}_{\text{noeg}}$.

We use a StackAPI endpoint to fetch Stack Overflow data. We set the parameter values for page_size and max_page to 100 and 400 respectively, which can give us a maximum of 40000 Stack Overflow posts for each library. So, our pipeline will not get all Stack Overflow data for any library that has more than 40000 Stack Overflow posts, which might affect the result.

In this study, we try different prompting techniques and different prompts to get better results from the LLM. However, there is always potential to engineer a better prompt that can give better results from the LLM. Also, LLMs can be inconsistent in their results [61]. Although we query our LLM model multiple times and use the majority answer to overcome non-determinism, there is a chance that the results may vary between multiple runs of the tool for the same library.

We prepare a ground truth for evaluating $\text{FILTER}_{\text{discussAPI}}$ where we manually annotate Stack Overflow posts whether they discuss a certain API or not. As manual annotations are subjective, the evaluation results might have varied if more annotators were involved.

7.2 Construct Validity

In this thesis, we focus on APIs that do not have code examples and try to find if a lack of examples is causing developers to struggle while using those APIs. However, there might be scenarios where the code example might not be adequate or does not cover particular scenarios that the users are struggling with. For example, an API might have optional parameters that were not shown in the code example. There might be users who struggle with using those parameters. So, the assumption that lack of code examples is causing developers to struggle might not be sufficient, as bad code examples might also cause the same issue.

Another assumption we make is that if a Stack Overflow post discusses a particular API, it indicates that the developer who posted it is struggling with that API. This assumption might not always be true, as the post might talk about some different aspects of the API e.g. performance. Figure 5.3 shows an example where the user is facing a performance issue while using the `boto3.session.client` API from the boto3 Python library. In this case, the user is not struggling to use the API itself, but rather with inconsistent behavior of the API when it is being used in a different setting. This issue cannot be addressed properly by adding a code example in the documentation for this particular API.

7.3 External Validity

As different API documentation websites have different layouts, it is difficult to generalize our tool to detect APIs from all types of API documentation. We therefore limit ourselves to Sphinx-generated API documentation for the prototype tool that we created. However, the concept of our approach is applicable to any API documentation. So this is an implementation limitation rather than a conceptual limitation.

Also, we have analyzed only 12 Python libraries using our tool. If we analyze more libraries with our tools and verify the results attained, we can get a clearer picture of how our proposed method performs in a more generalized scenario.

We used the OpenAI's GPT-3.5-Turbo [40] model for this study as this is fast and cost-efficient. However, there are newer and bigger models available, with higher parameter counts, such as OpenAI's GPT-4 [62] [63]. They are trained with more data and have improved capabilities. Using the latest models might yield different or better results.

Chapter 8

Conclusion

The lack of code examples in API documentation has been established as an obstacle to learning and using APIs. Using this premise, our goal in this thesis was to design a technique to determine which APIs in a library's API documentation require code examples to improve the documentation quality. We proposed a pipeline to determine missing examples in API documentation that cause difficulty for software developers to use these APIs. We determine which APIs are missing examples from the API documentation. We then explore an LLM-based method along with data from Stack Overflow to determine if developers are struggling with those APIs. Lastly, we prioritize those APIs according to the number of detected posts. We indicate the APIs that have at least 3 Stack Overflow posts discussing them as the potential APIs that need code examples to improve their documentation quality.

Based on our proposed pipeline, we built a tool in Python. We evaluated all the intermediate steps of our tool and presented their performance scores. We analyzed 12 Python libraries using our tool and presented our results. Although the overall result of our tool is not promising, we believe there are improvement opportunities such as engineering better prompts and analyzing more libraries. We discussed different potential opportunities to improve this work in the future. We believe that with future improvements, our proposed pipeline may be able to identify places in the API documentation that require examples, and library authors can be warned about these places to improve their API documentation quality.

References

[1] B. Dagenais and M. P. Robillard, “Creating and evolving developer documentation: Understanding the decisions of open source contributors,” in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 127–136.

[2] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, “Building more usable APIs,” *IEEE software*, vol. 15, no. 3, pp. 78–86, 1998.

[3] M. P. Robillard, “What makes APIs hard to learn? answers from developers,” *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[4] G. Uddin, F. Khomh, and C. K. Roy, “Mining API usage scenarios from stack overflow,” *Information and Software Technology*, vol. 122, p. 106 277, 2020.

[5] J. Zhang, H. Jiang, Z. Ren, T. Zhang, and Z. Huang, “Enriching API documentation with code samples and usage scenarios from crowd knowledge,” *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1299–1314, 2019.

[6] C. Treude and M. P. Robillard, “Augmenting API documentation with insights from stack overflow,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 392–403.

[7] 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.

[8] 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.

[9] 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*, 2020, pp. 245–256.

[10] D. L. Parnas, “Precise documentation: The key to better software,” in *The future of software engineering*, Springer, 2010, pp. 125–148.

[11] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE software*, vol. 20, no. 6, pp. 35–39, 2003.

[12] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," in *Proceedings of the 2002 ACM symposium on Document engineering*, 2002, pp. 26–33.

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

[14] A. Synko and A. Peleshchyshyn, "Software development documenting–documentation types and standards,", vol. 98, no. 2, pp. 120–128, 2020.

[15] A. Turner, B. Tran, and C. Sewell, *Sphinx*, 2024. [Online]. Available: `https://www.sphinx-doc.org/en/master/index.html`.

[16] A. Gallant, *Pdoc*, 2024. [Online]. Available: `https://github.com/mitmproxy/pdoc`.

[17] M. Hudson-Doyle, *Pydoctor*, 2024. [Online]. Available: `https://github.com/twisted/pydoctor`.

[18] D. v. Heesch, *Doxygen*, 2024. [Online]. Available: `http://www.stack.nl/~dimitri/doxygen/index.html`.

[19] D. Goodger, *Restructuredtext*, 2024. [Online]. Available: `https://docutils.sourceforge.io/rst.html`.

[20] C. Treude, J. Middleton, and T. Atapattu, "Beyond accuracy: Assessing software documentation quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1509–1512.

[21] H. Tang and S. Nadi, "Evaluating software documentation quality," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, IEEE, 2023, pp. 67–78.

[22] G. Uddin and M. P. Robillard, "How API documentation fails," *Ieee software*, vol. 32, no. 4, pp. 68–75, 2015.

[23] 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)*, IEEE, 2019, pp. 1199–1210.

[24] S. Sohan, F. Maurer, C. Anslow, and M. P. Robillard, "A study of the effectiveness of usage examples in REST API documentation," in *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, IEEE, 2017, pp. 53–61.

[25] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?(nier track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 804–807.

[26] S. Beyer, C. Macho, M. Pinzger, and M. Di Penta, "Automatically classifying posts into question categories on stack overflow," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 211–221.

[27] C. Parnin and C. Treude, "Measuring API documentation on the web," in *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, 2011, pp. 25–30.

[28] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 643–652.

[29] M. Liu, X. Peng, A. Marcus, S. Xing, C. Treude, and C. Zhao, "API-related developer information needs in stack overflow," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4485–4500, 2021.

[30] W. Wang, H. Malik, and M. W. Godfrey, "Recommending posts concerning API issues in developer q&a sites," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, IEEE, 2015, pp. 224–234.

[31] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Classifying stack overflow posts on API issues," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, IEEE, 2018, pp. 244–254.

[32] G. Uddin and F. Khomh, "Automatic mining of opinions expressed about APIs in stack overflow," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 522–559, 2019.

[33] IBM, *What are llms?* 2024. [Online]. Available: `https://www.ibm.com/topics/large-language-models`.

[34] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–17.

[35] J. Liu, X. Tang, L. Li, P. Chen, and Y. Liu, "Chatgpt vs. stack overflow: An exploratory comparison of programming assistance tools," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, IEEE, 2023, pp. 364–373.

[36] E. Chen, R. Huang, H.-S. Chen, Y.-H. Tseng, and L.-Y. Li, "Gptutor: A chatgpt-powered programming tool for code explanation," in *International Conference on Artificial Intelligence in Education*, Springer, 2023, pp. 321–327.

[37] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[38] J. Ye, X. Chen, N. Xu, *et al.*, “A comprehensive capability analysis of gpt-3 and gpt-3.5 series models,” *arXiv preprint arXiv:2303.10420*, 2023.

[39] L. Zhang, M. Wang, L. Chen, and W. Zhang, “Probing gpt-3’s linguistic knowledge on semantic tasks,” in *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, 2022, pp. 297–304.

[40] OpenAI, *Gpt-3.5-turbo*, 2023. [Online]. Available: `https://platform.openai.com/docs/models/gpt-3-5-turbo`.

[41] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, “Why johnny can’t prompt: How non-ai experts try (and fail) to design llm prompts,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–21.

[42] DAIR.AI, *Prompting techniques*, 2024. [Online]. Available: `https://www.promptingguide.ai/techniques`.

[43] J. Kocoń, I. Cichecki, O. Kaszyca, *et al.*, “Chatgpt: Jack of all trades, master of none,” *Information Fusion*, vol. 99, p. 101 861, 2023.

[44] G. Uddin, O. Baysal, L. Guerrouj, and F. Khomh, “Understanding how and why developers seek and analyze API-related opinions,” *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 694–735, 2019.

[45] G. Brandl, “Sphinx documentation,” *URL http://sphinx-doc. org/sphinx. pdf*, 2021.

[46] L. Richardson, “Beautiful soup documentation,” *April*, 2007.

[47] H. Zhong and Z. Su, “Detecting API documentation errors,” in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 803–816.

[48] A. Wegner, *StackAPI documentation*, 2016. [Online]. Available: `https://stackapi.readthedocs.io/en/latest/index.html`.

[49] D. Radev, E. Hovy, and K. McKeown, “Introduction to the special issue on summarization,” *Computational linguistics*, vol. 28, no. 4, pp. 399–408, 2002.

[50] M. Allahyari, S. Pouriyeh, M. Assefi, *et al.*, “Text summarization techniques: A brief survey,” *arXiv preprint arXiv:1707.02268*, 2017.

[51] A. Payong, *Introduction to extractive and abstractive summarization techniques*, 2024. [Online]. Available: `https://blog.paperspace.com/extractive-and-abstractive-summarization-techniques/`.

[52] G. Erkan and D. R. Radev, “Lexrank: Graph-based lexical centrality as salience in text summarization,” *Journal of artificial intelligence research*, vol. 22, pp. 457–479, 2004.

[53] OpenAI, *Gpt models*, 2023. [Online]. Available: `https://platform.openai.com/docs/models`.

[54] OpenAI, *Openai API pricing*, 2023. [Online]. Available: `https://openai.com/api/pricing/`.

[55] DAIR.AI, *Zero-shot prompting*, 2024. [Online]. Available: `https://www.promptingguide.ai/techniques/zeroshot`.

[56] DAIR.AI, *Few-shot prompting*, 2024. [Online]. Available: `https://www.promptingguide.ai/techniques/fewshot`.

[57] DAIR.AI, *Chain-of-thought prompting*, 2024. [Online]. Available: `https://www.promptingguide.ai/techniques/cot`.

[58] L. Craig, *Chain-of-thought prompting*, 2024. [Online]. Available: `https://www.techtarget.com/searchenterpriseai/definition/chain-of-thought-prompting`.

[59] M. L. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[60] H. v. Kemenade, M. Thoma, R. Si, and Z. Dollenstein. "Top pypi packages." (), [Online]. Available: `https://hugovk.github.io/top-pypi-packages/` (visited on 03/06/2024).

[61] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: The threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[62] OpenAI, *Gpt-4*, 2023. [Online]. Available: `https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4`.

[63] C. Emmanuel, *Gpt-3.5 and gpt-4 comparison*, 2023. [Online]. Available: `https://medium.com/@chudeemmanuel3/gpt-3-5-and-gpt-4-comparison-47d837de2226`.

Appendix A

LLM Query Prompt

A.1 Template of the Few-shot Query Prompt With Real Example Posts

```
This is a stack overflow post:

Post:
"""
pandas pivot_table column names:

For a dataframe like this:

d = {'id': [1,1,1,2,2], 'Month':[1,2,3,1,3],'Value
    ':[12,23,15,45,34], 'Cost':[124,214,1234,1324,234]}
df = pd.DataFrame(d)

      Cost   Month   Value   id
0     124        1      12    1
1     214        2      23    1
2     1234       3      15    1
3     1324       1      45    2
4     234        3      34    2
to which I apply pivot_table

df2 =     pd.pivot_table(df,
                          values=['Value','Cost'],
                          index=['id'],
                          columns=['Month'],
                          aggfunc=np.sum,
                          fill_value=0)
to get df2:

        Cost                 Value
Month      1     2      3      1    2    3
id
1        124   214   1234     12   23   15
2       1324     0    234     45    0   34
```

```
is there an easy way to format resulting dataframe column names
    like

id       Cost1     Cost2       Cost3  Value1    Value2    Value3
1          124       214        1234      12        23        15
2         1324         0         234      45         0        34
If I do:

df2.columns =[s1 + str(s2) for (s1,s2) in df2.columns.tolist()]
I get:

     Cost1  Cost2  Cost3  Value1  Value2  Value3
id
1      124    214   1234      12      23      15
2     1324      0    234      45       0      34
How to get rid of the extra level?

thanks!
"""

Is the main topic of this post this API: pandas.pivot_table?
Answer only in "yes" or "no"

Answer:
No


This is a stack overflow post:

Post:
"""
define aggfunc for each values column in pandas pivot table:

Was trying to generate a pivot table with multiple "values"
    columns. I know I can use aggfunc to aggregate values the way
    I want to, but what if I don't want to sum or avg both columns
     but instead I want sum of one column while mean of the other
    one. So is it possible to do so using pandas?

df = pd.DataFrame({
          'A' : ['one', 'one', 'two', 'three'] * 6,
          'B' : ['A', 'B', 'C'] * 8,
          'C' : ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
          'D' : np.random.randn(24),
          'E' : np.random.randn(24)
})
Now this will get a pivot table with sum:

pd.pivot_table(df, values=['D','E'], rows=['B'], aggfunc=np.sum)
And this for mean:

pd.pivot_table(df, values=['D','E'], rows=['B'], aggfunc=np.mean)
How can I get sum for D and mean for E?
```

```
Hope my question is clear enough.
"""

Is the main topic of this post this API: pandas.pivot_table?
Answer only in "yes" or "no"

Answer:
Yes

This is a stack overflow post:

Post: """
$post_body
"""

Is the main topic of this post this API: '$api_fqn'?
Answer in "yes" or "no"
```

A.2 Template of the Chain-of-Thought Query Prompt With Real Example Posts

```
This is a stack overflow post:

Post:
"""
pandas pivot_table column names:

For a dataframe like this:

d = {'id': [1,1,1,2,2], 'Month':[1,2,3,1,3],'Value
    ':[12,23,15,45,34], 'Cost':[124,214,1234,1324,234]}
df = pd.DataFrame(d)

     Cost  Month  Value  id
0    124       1     12   1
1    214       2     23   1
2    1234      3     15   1
3    1324      1     45   2
4    234       3     34   2
to which I apply pivot_table

df2 =    pd.pivot_table(df,
                        values=['Value','Cost'],
                        index=['id'],
                        columns=['Month'],
                        aggfunc=np.sum,
                        fill_value=0)
to get df2:

       Cost             Value
```

```
Month       1     2      3      1    2    3
id
1         124   214   1234     12   23   15
2        1324     0    234     45    0   34
is there an easy way to format resulting dataframe column names
    like

id       Cost1     Cost2       Cost3  Value1    Value2    Value3
1          124       214        1234     12         23        15
2         1324         0         234     45          0        34
If I do:

df2.columns =[s1 + str(s2) for (s1,s2) in df2.columns.tolist()]
I get:

    Cost1   Cost2   Cost3   Value1   Value2   Value3
id
1      124     214    1234       12       23       15
2     1324       0     234       45        0       34
How to get rid of the extra level?

thanks!
"""

Is the main topic of this post this API: pandas.pivot_table?

Thought:
No. Because the user asked the question about formatting the
    column names.
The main question is not regarding the API pandas.pivot_table.


This is a stack overflow post:

Post:
"""
define aggfunc for each values column in pandas pivot table:

Was trying to generate a pivot table with multiple "values"
    columns. I know I can use aggfunc to aggregate values the way
    I want to, but what if I don't want to sum or avg both columns
     but instead I want sum of one column while mean of the other
    one. So is it possible to do so using pandas?

df = pd.DataFrame({
          'A' : ['one', 'one', 'two', 'three'] * 6,
          'B' : ['A', 'B', 'C'] * 8,
          'C' : ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
          'D' : np.random.randn(24),
          'E' : np.random.randn(24)
})
Now this will get a pivot table with sum:

pd.pivot_table(df, values=['D','E'], rows=['B'], aggfunc=np.sum)
```

```
And this for mean:

pd.pivot_table(df, values=['D','E'], rows=['B'], aggfunc=np.mean)
How can I get sum for D and mean for E?

Hope my question is clear enough.
"""

Is the main topic of this post this API: pandas.pivot_table?

Thought:
Yes. Because the user is talking about a specific usage of the
    aggfunc
parameter of the API pandas.pivot_table that they are struggling
    with.


This is a stack overflow post:

Post:
"""
$post_body
"""

Is the main topic of this post this API: '$api_fqn'?
Answer in yes or no
```

Appendix B

Non-determinism and Mean Error of LLM Model

Prompting Technique	**Error Count vs No. of Data Point <No. of error response, No. of Data Point>**	**Mean Error**	**No. of Data Points for which LLM Gave Same Results**	**No. of Data Points for which LLM Gave Different Results**
chain-of-thought	<0: 458>, <11: 139>, <10: 8>, <1: 6>, <8: 4>, <5: 4>, <7: 4>, <6: 3>, <9: 3>, <3: 3>, <4: 3>, <2: 2>	0.252	597	40
few-shot	<0: 443>, <11: 138>, <8: 13>, <3: 10>, <1: 7>, <10: 6>, <2: 5>, <6: 4>, <9: 4>, <4: 3>, <7: 2>, <5: 2>	0.261	581	56
zero-shot	<0: 404>, <11: 164>, <3: 10>, <1: 9>, <8: 9>, <10: 8>, <2: 7>, <9: 7>, <5: 6>, <6: 5>, <7: 5>, <4: 3>	0.311	568	69
	Total data point: 637			

Table B.1: Error and Non-determinism of LLM model

To overcome the non-determinism of the LLM, we queried each data point (API, post) to the LLM 11 times for each prompting technique and chose the majority answer. We found that, for the total 637 data points, LLM gave a deterministic answer for most of the cases. The second last column of the table B.1 shows the number of data points for which LLM gave the same result (11 times), and the last column indicates the number of data points for which the LLM gave different responses (out of 11 times).

To determine the error, we compare the results of the LLM with our ground truth. The second column of the table B.1 shows the number of wrong responses among the 11 queries and the corresponding number of data points for which LLM gave that many wrong responses. For example, <11: 139> indicates that for 139 data points, LLM gave 11 wrong responses (errors) among the 11 queries, i.e., all wrong responses. The mean errors for each of the prompting techniques are shown in the third column of the table.