Completing Function Documentation Comments Using Structural Information

Ciurumelea, Adelina; Alexandru, Carol V.; Gall, Harald C.; Proksch, Sebastian

DOI
10.1007/s10664-022-10284-6

Publication date
2023

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
Ciurumelea, A., Alexandru, C. V., Gall, H. C., & Proksch, S. (2023). Completing Function Documentation Comments Using Structural Information. *Empirical Software Engineering*, *28*(4), Article 86. https://doi.org/10.1007/s10664-022-10284-6

Empirical Software Engineering (2023) 28:86
https://doi.org/10.1007/s10664-022-10284-6

Completing Function Documentation Comments Using Structural Information

Adelina Ciurumelea[1] · Carol V. Alexandru[1] · Harald C. Gall[1] · Sebastian Proksch[2]

Accepted: 12 December 2022

Abstract

Source code comments are a cornerstone of software documentation facilitating feature development and maintenance. Well-defined documentation formats, like Javadoc, make it easy to include structural metadata used to, for example, generate documentation manuals. However, the actual usage of structural elements in source code comments has not been studied yet. We investigate to which extent these structural elements are used in practice and whether the added information can be leveraged to improve tools assisting developers when writing comments. Existing research on comment generation traditionally focuses on automatic generation of summaries. However, recent works have shown promising results when supporting comment authoring through a *next-word* prediction. In this paper, we present an in-depth analysis of commenting practice in more than 18K open-source projects written in Python and Java showing that many structural elements, particularly parameter and return value descriptions are indeed widely used. We discover that while a majority are rather short at about 6 to 9 words, many are several hundred words in length. We further find that Python comments tend to be significantly longer than Java comments, possibly due to the weakly-typed nature of the former. Following the empirical analysis, we extend an existing language model with support for structural information, substantially improving the Top-1 accuracy of predicted words (Python 9.6%, Java 7.8%).

Communicated by: Christoph Treude

✉ Adelina Ciurumelea
ciurumelea@ifi.uzh.ch

Carol V. Alexandru
alexandru@ifi.uzh.ch

Harald C. Gall
gall@ifi.uzh.ch

Sebastian Proksch
S.Proksch@tudelft.nl

[1] University of Zurich, Zurich, Switzerland

[2] Delft University of Technology, Delft, Netherlands

Published online: 23 May 2023

Keywords Comment completion · Python documentation strings · Javadocs · Neural language models

1 Introduction

Source code comments are an important part of any software project as they are essential for developers to understand, use and maintain projects. However, writing comments is a tedious, manual process and, as a result, software is often poorly documented (Aghajani et al. 2020). Research has long attempted to reduce this effort through automated comment generation. The earliest works were focused on generating function and class comments (Haiduc et al. 2010; Moreno et al. 2013) through custom heuristics and templates. More recent data-driven approaches are inspired from the field of machine translation and use deep learning and natural language processing to generate summaries of the source code (Hu et al. 2018; Wan et al. 2018; Le Clair et al. 2019). What all these approaches have in common is that they focus on the first sentence of the documentation comment. This sentence is treated as the summary and the rest of the comment is ignored. In practice, however, comments contain a wide range of topics, ranging from functional descriptions to discussions of the rationale behind design decision (Writing system software: code comments 2021; Pascarella and Bacchelli 2017). It is also well known that developers spend more than half of their time on source code comprehension activities (Xia et al. 2018). While a good summary can be helpful, it is just the tip of the iceberg and represents only a small fraction of the whole comment. Unfortunately, automatically generating complete comments is still out of reach for current machine learning techniques.

Several documentation formats exist that introduce structural elements for languages such as Python and Java. These elements allow the organization of comment contents into various sections, for example, said summary, but also a long description, a description of parameters and return values, details about possible runtime errors, or other fields that are specific to the programming language or the documentation framework. A popular example of such a format is Javadoc for documentation comments in Java code. Having structural elements does not only improve the readability of a comment, but also enables automated processing to automatically generate API documentation or to provide better tool support in development environments.

In this paper we investigate the idea that leveraging the structural information available in comments can improve the assistance systems for developers for writing comments. Instead of generating full comments though, we keep the developer in the loop. Previous work has introduced a promising comment-completion tool that works in a semi-automated way (Ciurumelea et al. 2020), which -similar to email clients (Chen et al. 2019) and messaging apps- suggests likely next words to reduce repetitive typing and the required time and effort. We enhance this previous approach by considering the section information as additional context to improve the quality of the predicted comment completions.

In this study, we answer the following research questions to investigate the context and the feasibility of our vision:

RQ1: How structured are function documentation comments?
RQ2: Can structural information be used to improve the accuracy of a comment completion tool?
RQ3: How dependent are the results on a concrete programming language?

To answer these questions, we have compiled two datasets that contain more than 14K Java and Python projects. We have comprehensively analyzed this dataset and have measured various characteristics of the contained comments. Our results show that structured documentation comments are indeed widely used in practice, for example, we could identify a formatting style for over half of the Python projects. While summaries, also called short descriptions are indeed the most widely used comment section across both languages, also long descriptions, declarations of parameters and return values, and explanations of possible exceptions/raises are frequently contained in a comment.

These results made us confident that it is possible to improve the accuracy of the completion tool, so we trained several variants of a language model that uses the section information. The evaluation results show improvements between 4.1% for the *Long description* section and 9.6% for the *Return* section of the obtained Top-1 accuracy, when compared to the baseline for the Python dataset. We also found that some sections are significantly more predictable than others and achieved significantly different Top-1 absolute accuracy values from 0.194 (for the *Long description*) and 0.323 (for the *Raises* section) for the Python base model represented by the Context LM.

A comparison across the Python and Java languages datasets has revealed several similarities. Comments in both languages contain structured comments, however while Python has a diverse set of formatting styles, Java is dominated by Javadoc. We found that Java comments contain more structural elements than Python and that documentation comments of Java projects are more complete and include the different sections more often. Also the language model can achieve similar results for Java and Python, which suggests that our initial observations can be generalized to other programming languages. However, further research is required to solidify these initial findings and replicate them in a larger array of programming languages.

Overall, this paper presents the following main contributions:

- An empirical study over two programming languages that investigates the use of function documentation formats in practice.
- An extension of an existing language model for the task of *code comment completion* that leverages the additional section structure to improve the accuracy of the generated completion suggestions.
- A comparative evaluation of the study and the language models for two programming languages to strengthen our findings.

We have released a replication package that contains the datasets and the corresponding source code we used for the analysis of the projects and the training and evaluation of the neural language models (Replication package (2022)).

2 Overview

Motivated by an empirical study on comment sections, this paper proposes a novel comment suggestion engine that is assessed in an extensive evaluation approach in two different programming languages. To clarify this complex methodology of the paper, we provide an overview over the paper structure in Fig. 1 to illustrate the different high-level parts.

The first part of this paper is an empirical study on source code documentation practices in the wild. We base this initial investigation on the CodeSearchNet dataset (Husain et al. 2019b). This dataset is clean and ready to use, which provides an excellent starting point

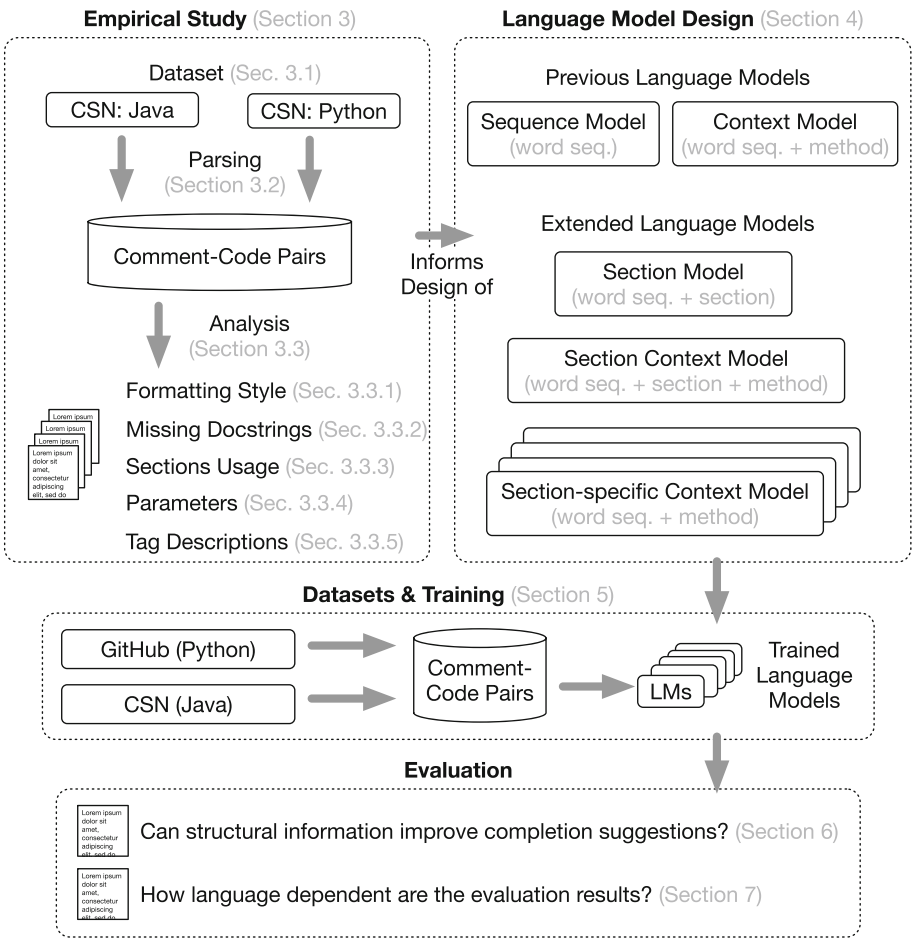


Fig. 1 Overview over the paper structure

for our research. The dataset also covers different programming languages, so it is possible to compare results on a similar ground. The study that is presented in this paper is based on Java and Python to cover to popular programming languages with different paradigms. We will introduce more details about the two (sub-)datasets in Section 3.1.

We preprocess the existing data for both languages and establish a pool of source code/comment pairs. Details about this preprocessing will be provided in Section 3.2. Based on this dataset, we can answer RQ1 by investigating several aspects of documentation comments: which format guideline is most popular (Section 3.3.1), how often are docstrings used/missing (Section 3.3.2), which sections are commonly used in docstrings (Section 3.3.3), how consistent is the documentation of method parameters (Section 3.3.4), and how long are the descriptions of *tag-like* structures (Section 3.3.5).

The results of these analyses are used to inform our design of the completion engine that is presented in Section 4. We start with two neural language models for comment completion that have been introduced in previous work and introduce three extensions that leverage the additional information that is available when section information is taken into account: both

models will be extended through an additional embedding layer to encode the enclosing section in the comment. While the base model only considers the word sequence, the context model also considers the method body as input. The third model is an ensemble technique that contains multiple instances of the context model, which are all trained/specialized for one particular comment section.

To instantiate the different models, we have established another dataset. While for Java, the CSN dataset was extensive, we found the extent of the Python dataset limited and decided to complement it with additional data that we mined from GitHub. Section 5 contains detailed information about the second dataset and our approach to training the different language models. Using the resulting models allowed us to investigate the two final research questions: As a first step, we investigate in Section 6 for Python, whether the additional section information can actually improve the performance of the completion suggestion (RQ2). In the next step, we answer RQ3 by shedding light onto the questions whether these results also generalize to other languages like Java (Section 7).

3 Structure Analysis of Function Documentation Comments (RQ1)

We noticed during our previous work (Ciurumelea et al. 2020) that Python documentation comments (docstrings) often have a well defined structure as a result of following a specific formatting style, and are not only simple natural language textual descriptions of the accompanying code. As this observation seems to be relevant for approaches that analyse the content of comments or build tools for supporting developers to write comments, we decided to further investigate how common formatting styles are and how they are used in practice. While past research has studied the content of comments (Pascarella and Bacchelli 2017) as well as the co-evolution of source code and comments (Fluri et al. 2007; Wen et al. 2019), they did not consider the structure of documentation comments. In addition, we also investigate the prevalence and size of documentation contained in different sections.

Our previous work has focused on documentation comments for Python functions, however another popular programming language, Java, also uses a formatting style called Javadoc (How to write doc comments for the javadoc tool 2021) for writing documentation comments. An accompanying tool, also named `javadoc`, exists for automatically extracting these comments from the source code to different kinds of output formats. While formatting styles for other programming language do exist, to the best of our knowledge these are less common with fewer available open-source projects. Therefore, we restrict our investigations to these two programming languages.

In the next sections, we describe the dataset, the process for parsing the documentation comments and functions, and finally present the findings of our analysis. We analyse the prevalence of the different formatting style for the Python dataset. Following, we study if the Python and Java documentation comments have the structure we expect and investigate where the *parameter*, *return* and *raises/throws* sections should be included but are missing. We also analyse the distribution of the different sections among the projects in our dataset, comparing the findings between the two programming languages. Additionally, we look at how complete and correct the parameters sections are. Finally, we collect statistics on the size of section-based comments, and investigate whether parameters for which a dedicated section is missing are mentioned elsewhere in the docstring description.

We would like to note that throughout the paper we often use the term *docstring* to refer to a documentation comment accompanying a method for Java and/or a function for Python. When necessary we also specify the corresponding programming language.

3.1 Dataset

To study the structure of documentation comments we use the Python and Java datasets provided during the CodeSearchNet challenge (Husain et al. 2019b), which aimed to explore semantic code search, i.e., retrieving relevant code given a natural language query (Husain et al. 2019a). This data is suited to our goal, as it provides comment-and-code pairs at the function level. It comprises 4,769 Java repositories with 136,336 distinct files and 13,590 Python repositories with 92,354 files, respectively.

To build the corpus, the authors of Husain et al. (2019b) chose publicly available open-source non-fork GitHub repositories that were identified as libraries (used by at least one other project), and sorted by "popularity" represented by the number of stars and forks.

The following filtering steps for selecting the comment-function pairs were applied:

- only include functions that do have an associated documentation comment;
- filter-out pairs for which the comment is shorter than 3 tokens;
- filter-out pairs for which the function is shorter than 3 lines;
- remove functions with the word "test" in the name, and constructors and extension methods such as __str__ in Python or toString in Java;
- remove duplicates using an algorithm described by Allamanis (2018) based on Jaccard similarity from the dataset and keeping a single copy.

In Table 1 we include statistics for the analysed datasets on a repository level. These include the files and functions that were not removed after the filtering step. The code is tokenized using TreeSitter (GitHub's universal parser), while the documentation comments are tokenized using NLTK's tokenizer (Nltk sentence tokenizer 2021). Although, software-specific tokenizers are subject to ongoing research (e.g., S-POS (Ye et al. 2016)), these are not as readily usable, and we found NLTK sufficiently precise. A full word is considered

Table 1 Statistics per Repository for the Python and Java Datasets

Dataset	25%	50%	75%	95%	100%
Python					
Files	1	2	5	16	517
Function-Docstring Pairs	4	10	27	127	11211
Code Length	43	72	132	341	28410
Docstring Length	11	26	61	184	8510
Java					
Files	1	2	4	12	1181
Function-Docstring Pairs	6	20	67	369	22028
Code Length	42	66	121	331	68278
Docstring Length	15	30	56	137	7135

a token, but also punctuation signs or mathematical operators. As an example, this function with the largest number of code tokens for Python includes a large number of matrix multiplications.

3.2 Parsing Documentation Comments

The current best practice is for Python and Java function documentation comments to follow a specific style and structure, nevertheless, this is largely just a recommendation, and tools enforcing docstring formatting are not widely applied, even though they exist[1]. To understand the structure of these comments, we first need to parse them and extract the different sections, as described next.

3.2.1 Parsing Python Docstrings

Python documentation strings or docstrings are string literals that occur as the first statement in a function or method and can be accessed through a special attribute at runtime. The goal of a docstring is to summarize the function behavior and document the arguments, return value(s), side effects, exceptions raised, and any restrictions related to when the function can be called (Pep 257 – docstring conventions 2021). In Listing 1 we include an example of a function docstring written using the Google style format. For Python, the official recommendation is to use the reStructuredText (reST) style for formatting docstrings as mentioned in Pep 287 – restructuredtext docstring format (2021). Nevertheless, in practice several other formatting styles are also used, like Epytext (The epytext markup language 2022), Google (Google docstring style 2022) and NumPy (Numpy docstring style 2022). All these formats contain the following main sections (Numpydoc docstring guide 2021). Some sections are specified by corresponding markup indicators (*"tags"*):

- *Short description*: a one-line summary;
- *Long description*: an extended summary clarifying the functionality;
- *Parameters*: one tag for each function, describing the argument including its type;
- *Return*: a tag explaining the return type and value;
- *Raises*: one tag for each possible exception raised by the function and under what conditions.

To extract the different sections, we use a modified version of an open-source parser for Python docstrings called *docstring_parser* (Docstring parser 2021). This offers support for the reST, NumPy and Google formats and we extended it to also parse docstrings using Epytext. The recommended best practice is to always start a docstring with a one-line summary. Then, if necessary, more details can be added as a *Long description* after an empty line. We observed that following this recommendation the parser made the assumption that the first line of a docstring is always the *Short description*, and everything else that does not start one of the other section belongs to the *Long description*. In our dataset, developers would sometimes write first sentences that are longer than a single line, therefore the parser would cut the sentence and erroneously interpret the first part until the newline character as the *Short description* and the rest as the *Long description*. We modified the parser to interpret the first sentence as the *Short description* using the nltk sentence tokenizer (Nltk sentence tokenizer 2021), and if there are additional sentences, allocate them to the *Long*

[1] E.g., https://github.com/maet3608/splint (Python) or javadoc's `javadoc -Xdoclint`.

```
def mnist_cross_entropy(images, one_hot_labels):
    """Returns the cross entropy loss of the classifier on images.

    Args:
        images: A minibatch tensor of MNIST digits. Shape must be
        [batch, 28, 28, 1].
        one_hot_labels: The one hot label of the examples. Tensor size
        is [batch, 10].

    Returns:
        A scalar Tensor representing the cross entropy of the image
        minibatch."""

    logits = tfhub.load(MNIST_MODULE)(images)
    return tf.compat.v1.losses.softmax_cross_entropy(one_hot_labels,
            logits, loss_collection=None)
```

Listing 1 A Documented Python Function

description. This choice was a trade-off between what the standard recommends and how developers write comments. It seemed wrong to cut a sentence at the end of a newline, for this reason we decided to extract the first sentence, even if it extends the first line. However, if the first line contains multiple sentences, just the first one is interpreted as the short description. We encounter the situation when the first sentence is longer than a single line in 18% of the comments with *Short Descriptions*, while when this is shorter than a single line appears in around 6% of the cases. However, we also noticed that when the first line contains multiple sentences, the extra sentences represent additional explanations and would be more appropriate for the long description.

After these changes we perform a preliminary evaluation for the different sections to understand how well the parser works. We want to assess whether parsing errors might influence our conclusions, while we do not intend to perform a full evaluation of the parser itself, as this would be outside of the scope of this paper. We choose for each section a sample of 100 docstrings and manually analyse them. To choose this sample we first randomly select 500 projects from our dataset, then for each section we iterate through the list of projects and randomly select one docstring per project that contains the specific section until we obtain a sample of 100 docstrings per section. For each section we obtain a different sample of 100 docstrings and the corresponding extracted content. One author read each docstring and classified it as correct/incorrect and marked the incorrect examples according to the identified problems. Table 2 contains the results of our evaluation. Note that the Incorrect Cases column only includes a subset of the identified cases for reasons of brevity.

For the *Short description*, as mentioned, we modified the parser to split at the sentence level. We use a well established parser from the NLTK package trained for the English language. Nevertheless this relies on punctuation being used correctly and can return an incorrect sentence split if there is no punctuation or if abbreviations using a "." character are included. From the sample of 100 docstrings, in 91 of the cases the parser correctly extracts the first sentence as the *Short description*. In 7 cases, the tokenizer does not split correctly the sentences and in 2 cases the docstring is not written in English, although that does not necessarily mean that the extraction is not correct. If the docstring contains correct punctuation similar to the English language, the tokenizer is still able to split the sentence correctly.

For the *Long description*, we obtain correct results in 81 cases. In 7 cases the sentence split is not correct, in 9 cases the docstring content is not formatted correctly and the *Long description* contains descriptions of return or parameter values. In 2 cases the docstring is

Table 2 Evaluation of Extracted Section

Section type	Correct	Incorrect	Incorrect cases
Python			
Short Description	91	9	incorrect sentence split
Long Description	81	19	incorrect sentence split, wrong formatting
Parameters	77	23	wrong formatting or parsing for type
Return	76	24	wrong formatting or parsing for type
Raises	74	26	wrong formatting or parsing for type
Java			
Short Description	96	4	incorrect sentence split
Long Description	96	4	incorrect sentence split
Parameters	97	3	docstring/signature name mismatch
Return	100	0	–
Raises	100	0	–

not written in English and in only one case the parser made an error and terminated the *Long description* too early, thus skipping some of its content.

For the *Parameters* section the parser extracts for each parameter the name, the type, whether it is optional or not and a potential default value. In this case it is able to correctly extract the parameter tag descriptions in 77 of the docstrings, from which two of the docstrings contain an empty description. The most common mistake we identified is that the parser is not able to extract the type of the parameter correctly, nevertheless, the parameter name and description are returned correctly. We encountered one example that was not written in English and in two cases the developer included extra content after including the parameters section which the parser erroneously attributed to the last parameter description.

For the *Return* section the parser extracts the type and the corresponding description. The parser correctly extracts this information for 76 of the cases, however in 5 of the cases the developers documents returning None, and in 1 case the description is empty. The most common mistake identified because of wrong formatting is that the type is not correctly detected and is included in the description. In 4 of the cases, the return value is a tuple that is not formatted correctly, therefore the parser only extracts one return value.

For the *Raises* section the parser returns a tuple with a type and description for each potential error that is documented. It works correctly in 74 of the cases, although for 2 cases the description section is left empty. The most common mistake encountered is that the type is not correctly identified, this is either because of a formatting error, or because the parser is not able to extract it, however the description is correctly identified. In 1 more case the docstring is not written in English.

We include in Table 3 several examples that could not be parsed correctly. For the *Short description*, the sentence tokenizer incorrectly detected a sentence split because of the '.'. For the *Long description* example, it is a bit difficult to understand what the developer meant to write, it seems that they used the wrong punctuation. For *Parameters* the parser extracts tuples of the form: (name, description, type, is optional). Therefore, for the example it is clear that the developer wanted to specify the type 'bool', however this should have been split by the rest of the description through the colon character and not the dot character. Finally, the *Raises* sections are extracted by the parsers as tuples of the form: (type, description). However, as the exception type is not followed by a newline, as the formatting style

Table 3 Examples of Incorrect Extracted Sections for Python

Section type	Partial docstring	Extracted section
Short description	Sample from a Polya-Gamma distribution, as in Proc Int Conf Mach Learn. 2012; 2012: 1343–1350.	Sample from a Polya-Gamma distribution, as in Proc Int Conf Mach Learn.
Long description	setLEDBrightness()	0 or 0x00=off 255 or
	Sets the brightness of the motion LED to the decimal hex value provided. 0 or 0x00=off 255 or 0xFF=Full Bright. (0-255 or 0x00-0xFF)	0xFF=Full Bright. (0-255 or 0x00-0xFF)
Parameters	:param metadata_dict: (dict) Metadata dictionary	('metadata_dict', '(dict) Metadata dictionary', None, None)
Return	Returns:	(None, 'bool. if the RPC
	bool. if the RPC interface file was written.	interface file was written.')
Raises	Raises	(None, 'OAuth1Error if the request is invalid.')
	- - - - - -	
	ValueError if broadcast fails	

specifies (Example numpy style python docstrings 2022), it was wrongly extracted together with the description.

After performing this analysis we can conclude that the parser works well enough for our purposes for the Python programming language. The most common identified problem, is that the parser is not able to correctly extract the type from the *Parameters*, *Return* or *Raises* tag descriptions. However, we do not consider this information when analysing the structure of Python documentation strings, therefore it does not affect our results.

3.2.2 Parsing Java Documentation Comments

Doc Comments, also called *Javadoc*, is the de-facto standard format for writing comments in Java, intended for use with the *javadoc* documentation generator. Thus, unlike in Python, where multiple commenting styles exist, documentation comments in Java usually adhere to one specific structure. Furthermore, programming environments (such as IntelliJ IDEA or Eclipse), automatically generate correctly formatted comment templates for the developer to fill in. API documentation to well-maintained Java projects, be it as HTML pages or in other formats, is commonly generated entirely from the documentation comments written in the source code. Hence, the Java documentation specifically states two primary goals for commenting code: "as API specification, or as programming guide documentation" (How to write doc comments for the javadoc tool 2021), with the former being a priority.

Java documentation comments can be written in HTML, placed directly before class, field, constructor or method declarations and typically consist of two parts: 1. A *description*, where the first sentence is a short summary, followed by 2. *block tags*, such as @author, @param, @see, or @deprecated, among many others.

In this study, we are interested only in the description and the @param, @throws and @return tags. Observe Listing 2 for an example of a Java method that includes a correctly formatted comment. Note that unlike in Python docstrings, the type of a parameter is *not* included in a block tag (although it may be optionally linked to).

```
/**
 * Create a new {@link XContentBuilder} using the given content.
 * <p>
 * The builder uses an internal {@link ByteArrayOutputStream} output
 * stream to build the content.
 * </p>
 *
 * @param c the content
 * @return a new {@link XContentBuilder}
 * @throws IOException if an {@link IOException} occurs while
 * building the content
 */
public static XContentBuilder builder(XContent c) throws IOException {
    return new XContentBuilder(c, new ByteArrayOutputStream());
}
```

Listing 2 A Documented Java Method

Because we wish to know the types of parameters for our study, we do not only parse the docstrings in our dataset, but also the method signatures. This way, we can determine (i) the types of documented parameters, provided their names are correctly mentioned in the block tag, and (ii) whether there are undocumented parameters in the method signature.

To parse the comments and methods, we use javalang (Javalang 2021). For the description section of each comment, we used NLTK's sent_tokenize to separate the first sentence (the summary or what we call the *Short Description*) from the rest (what we call the *Long description*).

All HTML tags and entities were stripped from the comments using the following regex: <.*? >| & ([a-z0-9]+ | #[0-9]1,6 | #x[0-9a-f]1,6);.

We implemented a two-fold verification to ensure that the data is extracted correctly. First, we handcrafted a code and comment pair as a sample representative of the data set and manually extracted the expected data. Then, we wrote automated tests to ensure that our approach extracts the expected data. Second, we drew a random sample of 100 from the extracted data and had one author inspect each of them for discrepancies between the expected and extracted values. The extraction of *Short* and *Long descriptions*, *Parameters*, *Return* and *Throws* statements worked perfectly in 97 out of 100 cases, with zero errors in identifying and extracting the different docstring sections, as well as extracting the types from the source code where applicable. Due to a limitation in the parser, the type of two parameters and one return statement were extracted incompletely: when a type is specified using a fully qualified identifier, the parser only extracts the first path component, e.g., only java instead of java.util.Map.

3.3 Comments Sections Analysis

In this part of the paper, we perform an in-depth investigation into the nature of docstring usage in Python and Java in order to answer the following questions:

- What is the prevalence of the four different styles for Python docstrings?
- How often are docstring sections missing where they should be present?
- How common are the different sections among the projects in our dataset?
- How well do parameter docstrings match up with the parameters specified in the function or method (correctness and completeness)?
- How often are parameters described in the short or long descriptions instead of having their own tag description?

– How can the length of parameter, return and exception descriptions be quantified?

3.3.1 Distribution of Python Formatting Styles

A particularity of Python documentation comments is that there does not exist a consensus on a single formatting style, there are several recommended styles while plain text docstrings are also accepted. To analyse the distribution of the formatting style, we identify the majority style as the style of a particular project. We can only determine the style of a docstring, if this contains one of the *Parameters*, *Return* or *Raises* sections, because the *Short* and *Long descriptions* look identically for all the styles as they do not contain any tag identifiers. We plot in Fig. 2 the obtained distribution. We notice that reST is the most popular style, followed by the Google, NumpyDoc and finally the Epytext styles. In 6,551 projects we were not able to identify a style, this either means that the docstrings do not describe the parameters, return and raises values or that the formatting used does not follow any of the known conventions. The projects for which a style could not be identified are included in all the analyses that follow in the next sections, as otherwise the comparison would not be fair with the Java dataset.

> From our analysis of the use of formatting styles of Python projects, we are able to identify one for over half of the projects in our dataset of 13,483 projects. We can conclude that a large part of projects do employ several different sections for writing documentation strings. The most common documentation style for the Python dataset is reST, followed by the Google, NumpyDoc and Epytext styles.

3.3.2 Analysis of Missing Docstring Sections

We start by analysing if the Python and Java documentation comments from our dataset have the structure we expect and investigate where the parameter, return and raises/throws tag descriptions should be included but are missing. We would like to emphasise that in our analysis we only consider function-comment pairs, therefore we only take into account functions for which the developer decided to write a comment. While we do not believe that

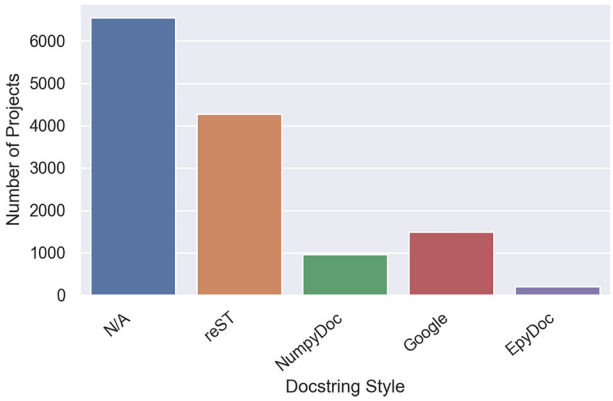


Fig. 2 Distribution of Docstrings Formatting Styles

a general rule to document all code is necessary, we also do not adhere to the saying that *"good code is self-documenting"*. By taking into account only functions with comments pairs, we select only cases for which a comment was deemed necessary. And, we do not see a valid reason why a comment should not be complete, once it had been included.

To find the cases for which the *Parameters*, *Return* and *Raises* sections are missing when they should be included, we parse the docstrings and extract the different tag descriptions. Please, note that we use the terms "Raises", "Throws" and "Exception" interchangeably to refer both to raises sections in Python docstrings and throws sections in Java documentation comments. We then conduct a static analysis on the functions from the dataset to determine if a particular function contains parameters in the signature, or return or exception-raising statements in the body. Then, we compare this information with the parsed docstring sections and compute the fraction of docstrings that have missing *Parameter*, *Return* or Raises docstring tags per project. For example, for a project with 100 function-docstring pairs, of which 30 have parameters included in the function signature, but lack a parameter section in the comment, we obtain a fraction of 0.3 for missing parameters. After obtaining these values for each project and section we compute and include the summary statistics results of the fractions of missing sections in Table 4. Here, we would like to note that for the Python dataset, we only consider projects for which a formatting style has been identified, that is we were able to parse correctly at least one of the parameter, return or raises sections in the docstrings of the project.

For this analysis we consider only absolute values. For instance, if the function has parameters in the signature, we only verify if there is at least one parameter documented in the docstring. Similarly, for the returns and raises we only report a missing case when there is at least one corresponding statement in the function body but none in the docstring. If the function does not include parameters in the signature, or has no return or raises statements in the body, then this is ignored in the corresponding analysis as to not inflate the results.

From Table 4 we can observe that developers omit these sections quite often for the Python programming language, although they should be included. The section developers include most often, is the *Parameters* section. The average fraction of project docstrings that should include a parameters section but do not, is 0.424. This value increases to 0.565 for the return section, while the raises section is the one developers include least frequently and is missing on average in over 0.85 of the project docstrings. This represents a convincing argument for our work, as there does not exist a valid reason for not writing complete documentation comments and developers would benefit from getting support in writing them, especially for the Python programming language. Interestingly, for the Java functions

Table 4 Summary Statistics for the Fraction of Missing Sections

Section	Mean	std	min	25%	50%	75%	max
Python							
Parameters	0.424	0.340	0.000	0.083	0.400	0.733	1.000
Return	0.565	0.368	0.000	0.218	0.625	0.941	1.000
Raises	0.859	0.282	0.000	0.888	1.000	1.000	1.000
Java							
Parameters	0.303	0.316	0.000	0.002	0.197	0.500	1.000
Return	0.243	0.292	0.000	0.000	0.125	0.397	1.000
Raises	0.868	0.181	0.000	0.000	0.000	0.857	1.000

```
def mnist_cross_entropy(images, one_hot_labels):
    """Returns the cross entropy loss of the classifier on images.

    Args:
        images: A minibatch tensor of MNIST digits. Shape must be
        [batch, 28, 28, 1].

    Returns:
        A scalar Tensor representing the cross entropy of the image
        minibatch."""

    logits = tfhub.load(MNIST_MODULE)(images)
    return tf.compat.v1.losses.softmax_cross_entropy(one_hot_labels,
            logits, loss_collection=None)
```

Listing 3 Example Python Function with Missing Parameters

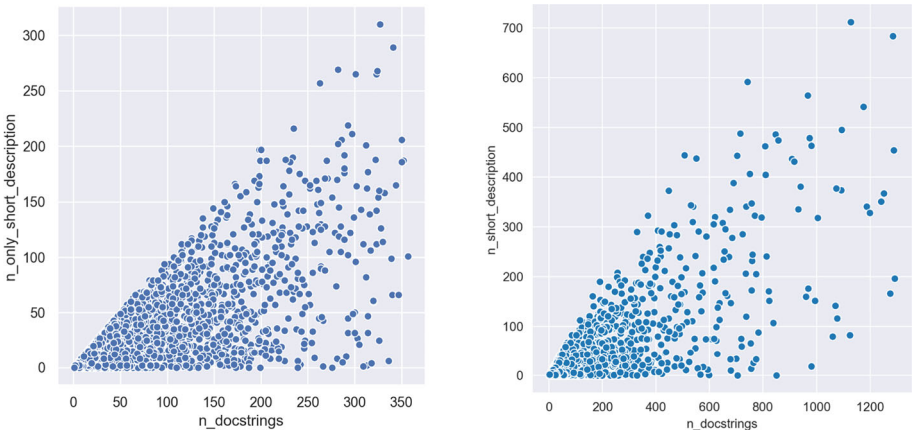
we analysed, this problem seems to be less pronounced. A possible reason could be that for Java projects it is more common to use IDEs which can create a documentation comment template automatically whenever a function is defined. If this is the case we expect to see more sections with empty descriptions. Indeed, the Java dataset contains more empty parameter descriptions at 10.03% vs. 6.59% in Python. Exception descriptions are also more frequently empty in Java at 18.23% vs. 2.85% in Python. However, return descriptions are empty in Java for 7.16% of functions, compared to 11.07% for Python. For those parameter, return and exception descriptions which are not empty, we provide a further empirical analysis regarding their size in Section 3.3.5.

From our preliminary analysis of missing sections for the Python docstrings, we notice that the *Raises* section is missing in a large number of cases. The *Return* section is also usually omitted, while the *Parameters* section seems to be the one developers do try to include most often. We can observe a similar pattern for the Java dataset, where the *Raises* section is most often neglected. However, it is less common for the *Parameters* section and *Return* tags to be entirely missing.

3.3.3 Distribution of Docstrings Sections

The minimal documentation comment includes only the *Short description* section, the short summary of the function. For Python this should fit on a single line, however as we observed developers write summaries that are longer than a single line, we extract the first sentence as the *Short description*. The short summary of a function has also been the focus of previous research that automatically generates summaries of code (Hu et al. 2018; Wan et al. 2018; Le Clair et al. 2019). In this section, we would like to understand how common it is for developers to only write a summary of the method for the projects in our datasets, therefore previous approaches only focusing on generating the short summaries of code are enough. Or if developers do provide more than just the first sentences for a documentation comment and additional approaches are needed to support them with this task.

To study how common it is for developers to only write this first sentence for the Python and Java projects in our datasets, we plot in Fig. 3 the number of extracted docstrings per project on the x axis and the number of corresponding number of *Short description* only docstrings on the y axis for the Python dataset. If the documentation comments in our dataset



(a) Python (b) Java

Fig. 3 Total Docstrings vs. Short Description only Docstrings per Project

would only contain short description, we should observe an almost linear relationship. On the contrary, if the comments would never contain only the short description, then the scatter plot would only include points on the x axis, resulting in a horizontal line. From the plots, we can observe that there is a large variety into the number of *Short description* only docstrings per project for both programming languages. Therefore there is no obvious correlation between the number of docstrings and how many of those are *Short descriptions* only.

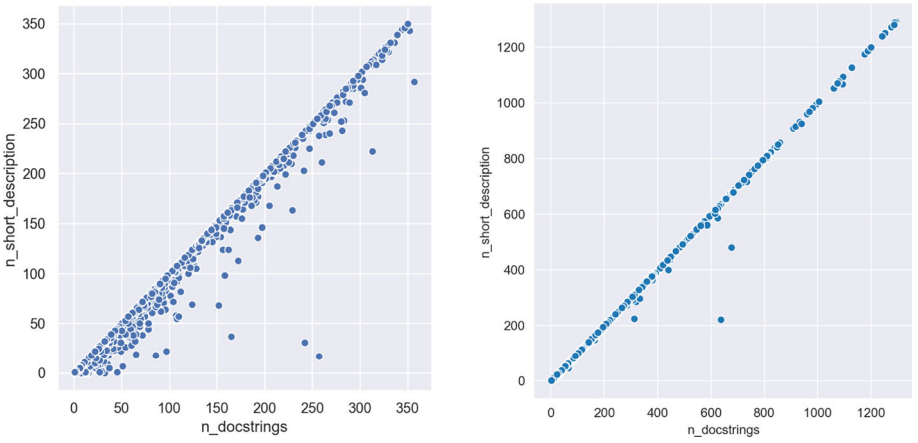
To better understand the distribution of these comments we compute the fraction of *Short description* only from the total docstrings per project and include the summary statistics for these values in Table 5. The average fraction of *Short description* only per project is around 0.38, while the median percentage has a close value of 0.31. We can notice that the Java projects have, in general, a higher number of documentation comments from the plots, while the average number of *Short descriptions* only per project is smaller with a value of 0.29 and the median value is only 0.18. This suggests that *Short descriptions* only comments are somewhat less frequent for Java documentation comments than for Python from the analysed dataset.

We would like to note that for all the scatter plots presented in the paper, we show only the projects that contain a number of docstrings less or equal to the 0.99 percentile, as showing all the points would lead to more dense and harder to read figures. As well, please note the differences in scales on the x and y axis for the Python and Java side-by-side plots. Using the same scale, would lead to figures that are more difficult to read.

Following, we analyse how common each of the studied sections: *Short description*, *Long description*, *Parameters*, *Return* and *Raises* are among the docstrings on a per project

Table 5 Summary Statistics for the Percentage of *Short Description* only Docstrings per Project

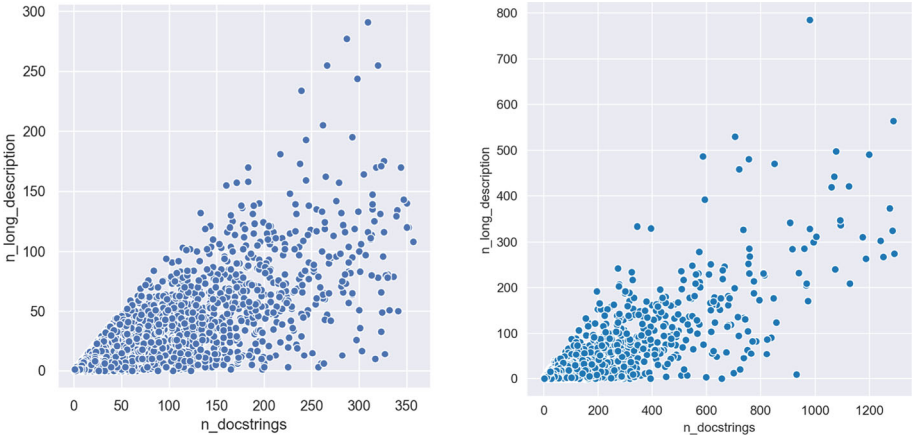
Dataset	Mean	std	min	25%	50%	75%	max
Python	0.38	0.34	0.00	0.04	0.31	0.62	1.00
Java	0.29	0.31	0.00	0.00	0.18	0.47	1.00



(a) Python (b) Java

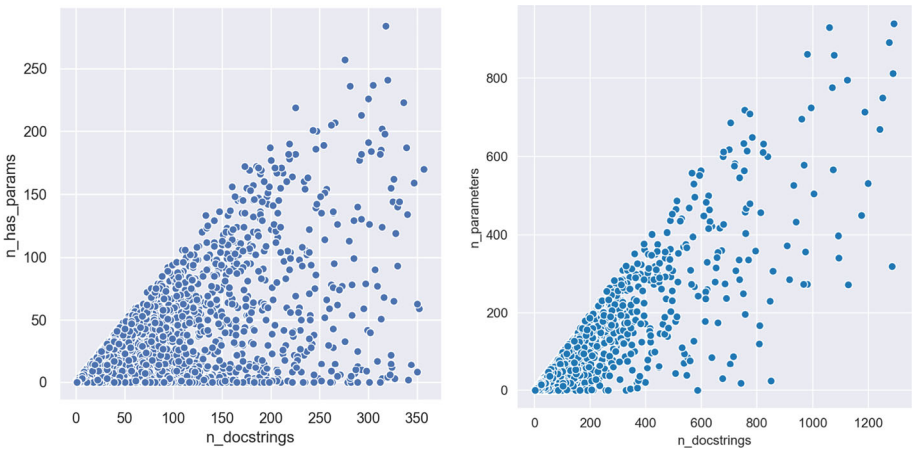
Fig. 4 Total Docstrings vs. Total Docstrings with Short Descriptions per Project

basis. We first look at how prevalent the *Short description* is over all the docstrings from the two datasets. We expect that most comments include a *Short description*, while it should be less common for a comment to include other sections (like *Parameters*, *Return* or *Raises*) but not this one. In Fig. 4 we plot the number of docstrings per project on the x axis and the number of extracted *Short descriptions* on the y axis. We can notice that there is almost a linear relationship between the number of docstrings and the ones that include a *Short description*. There is some variety for the Python dataset, while for the Java dataset this relationship between the number of docstrings per project and the corresponding number of *Short descriptions* is almost perfectly linear with the exception of a few projects. For both of our datasets it is clear that there is a strong inclination to include the summary as a *Short description* in a comment.



(a) Python (b) Java

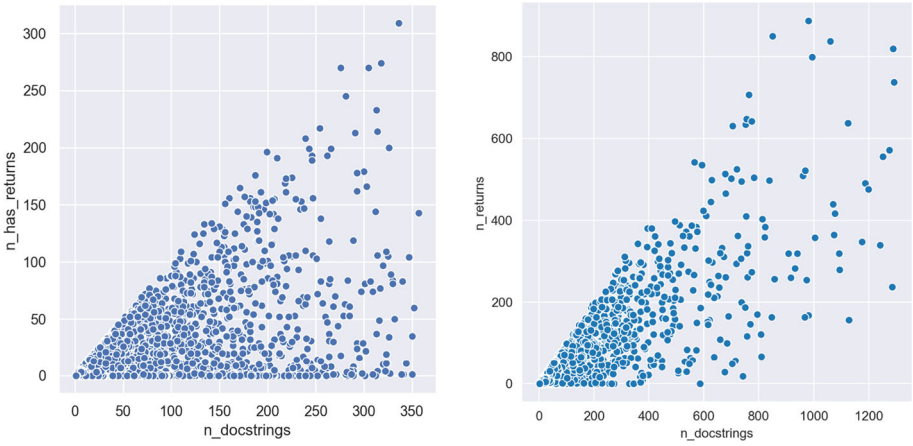
Fig. 5 Total Docstrings vs. Total Docstrings with Long Descriptions per Project



(a) Python (b) Java

Fig. 6 Total Docstrings vs. Total Docstrings with Parameters per Project

To analyze how common the *Long description* section is we plot in Fig. 5 the number of docstrings on the x axis and the number of extracted *Long description* per project on the y axis for both datasets. Here, we observe that there is a high variety on the frequency of this section, while still being relatively common. The goal of the *Long description* is to elaborate on the summary and can include different details that the developer finds important, for example dependencies with other functions, constraints that should be known when using the particular function or rationale for using a specific data structure or algorithm. While this section is less common than the *Short description*, developers use it frequently to describe more than just the one sentence summary of the function. We can observe that the frequency of this section is similar between Python and Java.



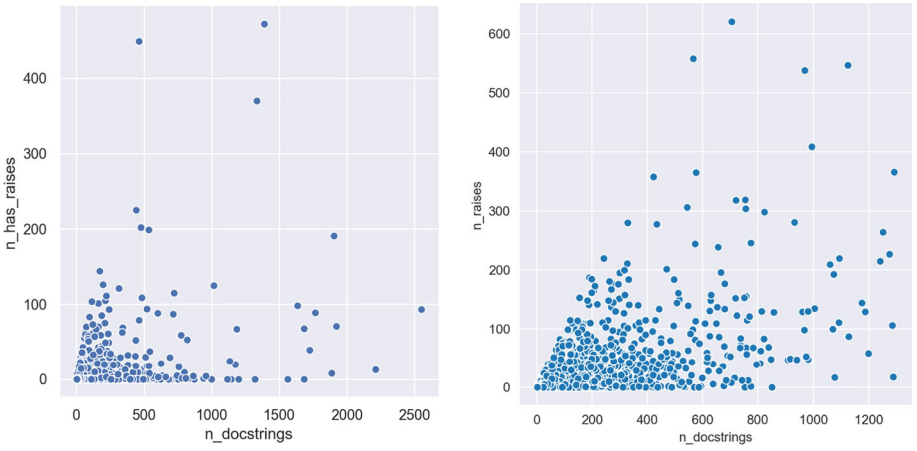
(a) Python (b) Java

Fig. 7 Total Docstrings vs. Total Docstrings with Return per Project

Next, we look at how common the *Parameters* section is. In Fig. 6 we plot the number of docstrings per project on the x axis, while on the y axis we plot the corresponding number of docstrings including this section for the Python and Java datasets. We can notice a higher prevalence of this section for the Java dataset, while the distribution between the number of *Long Descriptions* and the corresponding total number of docstrings has a linear tendency. The Python dataset seems to have several projects with various counts of docstrings, but without any extracted *Parameters* section, as indicated by large density of points on the x axis. These likely correspond to the projects for which we could not identify a style and classified them as N/A in Fig. 2. Here, we would like to note that it is possible for developers to have provided description for the parameters as free-text, while others include formatting errors which makes it impossible for the parser to correctly extract the *Parameters* section. In our results, we can only analyze sections that have been correctly formatted.

Finally, we analyse the *Return* and *Raises* sections. The plots in Fig. 7 include the number of docstrings on the x axis and the number of extracted *Return* sections on the y axis. We can observe similarities of this section with the *Parameters* section per programming language. For the Python dataset, both sections have a wide distribution without any clear correlation and a large density on the x axis. While for Java, there is a clear linear tendency while the projects plotted on the x axis are not very frequent. In Fig. 8 we plot the number of docstrings on the x axis and the number of well formed *Raises* sections on the y axis. Here we can notice a drastic difference in the frequency of this section for the Python dataset compared to the rest of the sections. Very few projects contain a large number of this section among the total number of docstrings. This difference is not as pronounced for the Java dataset with a larger variety of the frequency of this section among the various documentation comments.

Even though the scatter plots allow us to visualize the prevalence of the different sections per project, to gain a better quantitative understanding of how common these sections are, we perform a further analysis. We compute the fraction of docstrings including a particular section per project and then include the summary statistics for these fraction values in Table 6. From this table we can observe that for the Python dataset, the *Long description* is on average included in a third, while the *Parameters* section is included in a quarter



(a) Python (b) Java

Fig. 8 Total Docstrings vs. Total Docstrings with Raises/Throws per Project

Table 6 Summary Statistics for the fraction of included sections per project

Section	Mean	std	min	25%	50%	75%	max
Python							
Short description	0.977	0.104	0.000	1.000	1.000	1.000	1.000
Long description	0.328	0.301	0.000	0.042	0.271	0.500	1.000
Parameters	0.252	0.335	0.000	0.000	0.000	0.500	1.000
Return	0.181	0.295	0.000	0.000	0.000	0.284	1.000
Raises	0.025	0.103	0.000	0.000	0.000	0.000	1.000
Java							
Short description	0.996	0.434	0.000	1.000	1.000	1.000	1.000
Long description	0.250	0.242	0.000	0.333	0.200	0.375	1.000
Parameters	0.560	0.333	0.000	0.294	0.622	0.846	1.000
Return	0.448	0.327	0.000	0.150	0.443	0.709	1.000
Raises	0.156	0.230	0.000	0.000	0.500	0.222	1.000

of the docstrings of a project. The *Return* section shows up on average in 0.181 fraction of the docstrings per project, while the *Raises* section is extremely rare in the projects from the Python dataset. Here, we would like to note that the median value for the *Parameters*, *Return* and *Raises* sections is equal to 0, as around half of the projects do not have a majority styles, this means that such sections could not be extracted by the parser we used.

Interestingly, the *Long description* is less common in the Java dataset we studied, the average fraction of docstrings including this section per project is 0.25, 0.078 less than for the Python dataset, while the occurrence of the *Return* and *Raises* sections is significantly larger in the dataset we studied. The average fraction of docstrings per project including a *Parameters* section is 0.56, while for the *Return* and *Raises* section this value is equal to 0.448 and 0.156 respectively.

Following our analysis of the structure of documentation comments for the Python and Java dataset, we can make the following observations:

> Python docstrings include more frequently only a *Short description* as a documentation comment than Java. A large majority of documentation comments include the *Short description* with minor differences between the two programming languages. For the Python dataset, the *Parameters* and *Return* section are reasonably frequent, while the *Raises* section is extremely rare. The comments of the Java dataset are more complete and often include the *Parameters* and *Return* sections. However, the *Raises* section is also for this dataset the least frequent.

Finally, we can notice differences between the structure of Python and Java documentation comments. We speculate that this might be due to the more frequent use of IDEs for writing Java code, which generate the template of documentation comments automatically. Additionally, the existence of a single recommended and widely accepted formatting style might encourage developers to use this more often. Nevertheless, we observed for the sections that are included in Python documentation comments, they seem to be longer than the

Java counterpart. Further work is needed to understand the reasons behind these differences. However, the prevalence of these sections in both datasets we studied, indicates that both Java and Python developers can benefit from approaches that support them in writing fully formed comments and not only the summary.

3.3.4 Analysis of the *Parameters* Section

For the *Parameters* section, we extend our analysis by looking at how correct and complete this section is. We interpret precision as correctness in order to answer the following question: *"from the parameters that are described in the comment, how many do actually appear in the function signature?"*. And we interpret recall as completeness to answer the second question: *"from the parameters included in the function signature how many are described in the documentation comment?"*. We compute the precision (correctness) and recall (completeness) of the included parameters using the following two formulas:

$$Precision = \frac{|correctDocstringParams|}{|docstringParams|}$$

$$Recall = \frac{|correctDocstringParams|}{|functionParams|}$$

The *correctDocstringParams* represent the number of parameters described in the docstring that match parameters from the function signature. The *len(docstringParams)* represents the number of parameters included in the docstring, while *len(functionParams)* represent the number of parameters included in the function signature. For class instance methods we ignore the parameter `self`, as the recommendation is to not document it. As an example, in Listing 3 the description for the parameter `one_hot_labels` is missing and we obtain the following values for precision and recall: Precision = 1/1 = 1.0, Recall = 1/2 = 0.5.

We then compute the average of these values per project and include the summary statistics in Table 7. From this table we can observe that the average and median precision and recall per project for Python docstrings that do include parameters descriptions is around 0.5. This can happen if a developer does not document all the parameters in the docstring. Maybe they did not include them from the beginning, or maybe they neglected to update the corresponding comment when adding additional parameters at a later time. Another option is that the function parameter in the signature was renamed, but not in the docstring. Incomplete comments can be particularly problematic, as a developer, reading the comment might wrongly assume that it is complete and rely on incorrect information. Interestingly, the average precision per project of included parameters description is much higher for the Java projects at around 0.97, while the average recall per project is lower at 0.65. This further corroborates our previous observation, that the comments of the Java dataset are more complete. If parameters are included in the documentation comments these are correct, as

Table 7 Summary Statistics for the Parameter's Precision and Recall

Language	Metric	Mean	std	min	25%	50%	75%	max
Python	Precision	0.53	0.34	0.00	0.23	0.52	0.85	1.00
	Recall	0.51	0.33	0.00	0.20	0.50	0.81	1.00
Java	Precision	0.97	0.09	0.00	0.98	1.00	1.00	1.00
	Recall	0.65	0.35	0.00	0.38	0.77	0.98	1.00

named in the function signature, however also for the Java dataset sometimes parameters are missing as indicated by the lower average recall value.

> From our analysis of the correctness and completeness of the *Parameters* section we can observe that Java parameters are more often consistently documented than the ones in Python. However, for both programming languages developers often omit describing at least some of the parameters.

References in Short or Long descriptions In many cases, docstrings are present, but the expected parameters are not defined. We wanted to determine how often parameters may be documented simply as part of the *Short* or *Long descriptions*, rather than in their dedicated sections. For this investigation, we first filtered out all samples where the parameters are fully documented. For the remaining samples in which at least one parameter is undocumented, we (a) parsed the function body to extract the parameter names, (b) determined, whether each parameter name appears in the *Short* and/or *Long descriptions*, (c) calculated, for each project, the mean fraction of parameter names referred to in different parts of the description. We chose to consider the mean fraction for each project, rather than all fractions, as projects have a tendency for uniformity, and we wanted to avoid larger projects with more functions to bias the outcome. To parse the functions, we again used `javalang` for Java code, while using Python's own `ast` module for parsing Python functions. We were able to parse 496,248 Java methods. Since the CodeSearchNet dataset also contains older Python code, we used lib2to3 (Lib2to3 2022) to refactor any Python functions that could not be parsed initially. This allowed us to parse an additional 3,964, for a total of 442,980 Python functions. We were unable to parse 440 Java and 333 Python functions from the CodeSearchNet dataset, as they appear to be malformed. The mean fraction of mentioned parameters for each project is calculated via the following formula:

$$\frac{\sum_{i=0}^{|functions|} \frac{|mentionedParameters_i|}{|parameters_i|}}{|functions|}$$

Figure 9 visualizes these mean values. We can see that for both Python and Java, where docstring tags are missing, parameters are mentioned in about 10-40% of descriptions for most projects. Conversely, this means that parameters are undocumented in the majority of functions. The outliers indicate that in some projects, most parameters are documented, or at least mentioned, as part of the *Short* or *Long descriptions*.

> Our analysis reveals that if parameters, returns or exceptions are not documented in their dedicated section, they are likely not documented at all, although there exist some projects, which extensively cover parameters in the *Short* or *Long descriptions*.

3.3.5 Length of Tag Descriptions

Similar to parameters, also return values and exceptions are documented using *tags*, for example *@returns foo* or *@raises exception X*. To obtain an approximate understanding of how extensively this *tag-like* structures are documented (if they are present), we used NLTK's word tokenizer to count the number of words contained in their descriptions. For

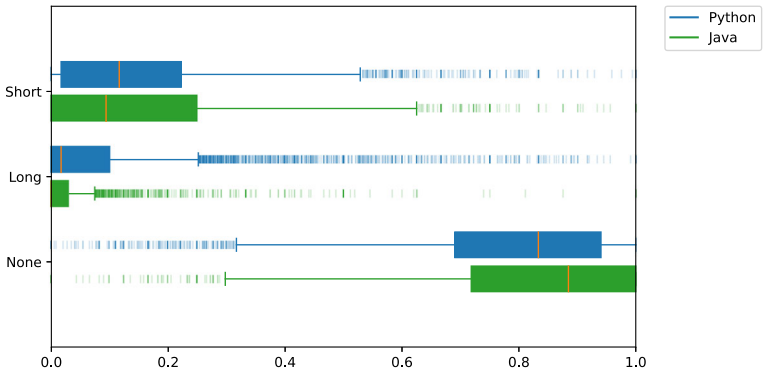


Fig. 9 For each project, the mean fraction of parameter names mentioned in the short or long descriptions (or not at all) for functions where not all code parameters have corresponding tags

parameters, we also check if the parameter name itself is mentioned as part of the description. We make sure to only identify parameter names not surrounded by other alphabetical characters (e.g., a parameter `i` would be identified in the sentence "Takes a number *i*." but not in "Not *i*mplemented!" by using the following regex:

rf"(?: ^ |[^ a-zA-Z])(re.escape(param))(?:[^ a-zA-Z]—$)

Figures 10, 11 and 12 display the word count distributions (not including zero-length descriptions), revealing the insights described next.

A quarter of all non-empty parameter descriptions appearing in Python and Java programs comprise just up to 5 or 3 words, respectively. From a qualitative inspection of our data, we conclude that these parameter tags are usually in the form of `@param thing the given thing`; i.e., the documentation provides barely any additional information over the source code. The same holds true for exceptions, where a quarter are described in 6 words or less in both languages. Furthermore, one in four parameter descriptions mention the parameter name itself, contributing to the word count, in both Python (27%) and Java (28%).

Tag descriptions in Python tend to be more verbose than in Java, this is true for both *Short* and *Long descriptions*. We identify at least one contributing reason: most Python code lacks type hints and Python code often leverages duck typing. For this reason, the parameter descriptions often describe default values or the ability to accept different types of values. Similarly, return values in Python are often complex values (such as dictionaries), which are not defined and documented in a separate source code location, so they are documented in the location where they are returned. It is also common for usage examples to be included as part of parameter or return descriptions. Listings 4, 5 and 6 illustrates a few such examples.

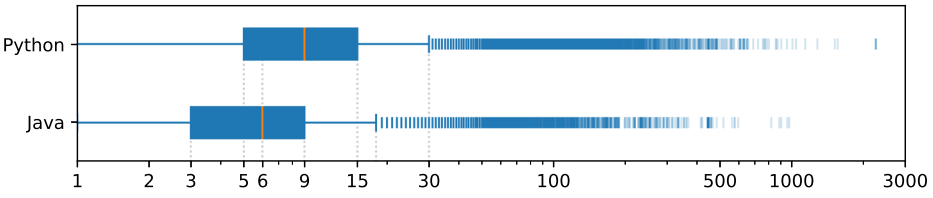


Fig. 10 Number of words contained in non-empty parameter descriptions, shown as quartiles

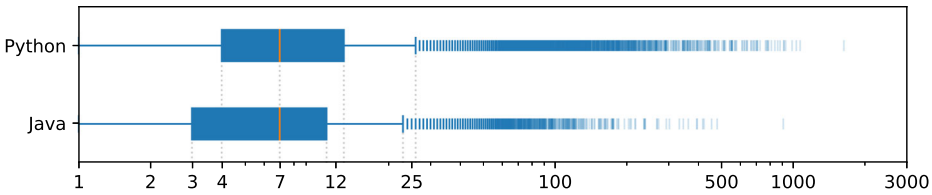


Fig. 11 Number of words contained in non-empty return descriptions, shown as quartiles

By contrast, parameters and return types in Java have a specific (or at least, generic) type which is documented elsewhere. Thus, return descriptions in Java are often similar to the short parameter and exception tags in the form of "@return the thing". Given the relative verbosity of Java code, it is reasonable to assert that Java code can be more self-documenting than Python code in this regard.

There are numerous outliers which provide extensive documentation for single parameters, return values, or even raised/thrown exceptions. Inspecting a random sample of long parameter descriptions reveals that these often accept a specific set of values (such as choosing a specific strategy or algorithm), in which case every value accepted by the function is described as part of the parameter documentation. Similarily, long return and exception descriptions often describe an extended set of possible values. We also observe that in some cases, the text provided as a description must have been placed there accidentally, for example duplicating documentation that belongs to the type of the parameter, possibly done so by an automated tool.

Table 8 gives an overview of all data analyzed in Sections 3.3.5 and 3.3.4.

To summarize: a quarter of parameter descriptions mention the parameter name itself, and although the majority of tag descriptions are very short, a significant number of very long descriptions exist. Python comments are generally more verbose than Java comments, possibly because of the weakly-typed nature of Python code.

4 Structure-based Comment Completion

The empirical study has shown that structural elements are indeed widely used in documentation comments of Python (docstrings) and Java (Javadoc). Motivated by this finding, we present novel language models in this section that leverage this structure information for the

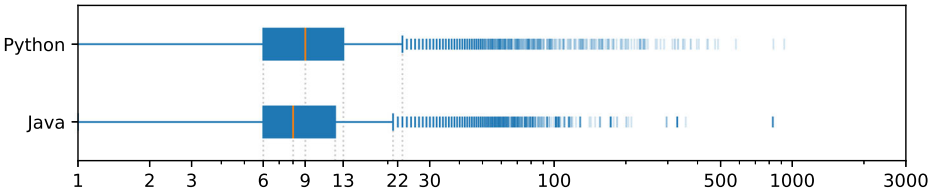


Fig. 12 Number of words contained in non-empty exception descriptions, shown as quartiles

```
    query_result: a list of dictionaries on the form
    {
        'p_value': float,
        'gene_id': str,
        'omim_id': int,
        'orphanet_id': int,
        'decipher_id': int,
        'any_id': int,
        'mode_of_inheritance': str,
        'description': str,
        'raw_line': str
    }
```

Listing 4 This return description from the robinandeer/puzzle Python project describes a complex return value

```
----------
der : int or list
How many derivatives to extract; None for all potentially nonzero
derivatives (that is a number equal to the number of points), or a
list of derivatives to extract. This numberincludes the function
value as 0th derivative.
```

Listing 5 This parameter description from the pandas-dev/pandas Python project illustrates that Python parameters may require more documentation due to weak typing

```
Returns:
    list of lists of str, one list for each Chapter or Appendix

>>> lines = get_lines(BOOK_PATH)
>>> next(lines)
('.../src/nlpia/data/book/Appendix F -- Glossary.asc',
 ['= Glossary\n',
  '\n',
  ""We've collected some ...])
```

Listing 6 This return description from the totalgood/nlpia Python project includes a usage example

Table 8 Overview statistics on the code and docstring analysis

Language	Java	Python
Total number of functions	496,688	443,313
Functions successfully parsed	496,248	442,980
Functions failed to parse	440	333
Parameter descriptions mentioning parameter	147,350	96,234
Functions with complete param tags	329,940	57,539
Functions with incomplete param tags	166,748	385,774
Parameters parsed	325,219	993,581
Parameters found in docstring tags	558,432	367,705
Parameters mentioned in short description	43,481	128,726
Parameters mentioned in long description	14,263	111,777
Parameters not mentioned at all	273,948	776,230
Empty parameter descriptions	33,636	11,111
Empty return descriptions	288,856	350,749
Empty exception descriptions	17,590	344

```
@classmethod
def as_view(cls, name, *class_args, **class_kwargs):
    """Converts the class into an actual view function that can be used
    with the routing system.  Internally this generates a function on the
    fly which will instantiate the :class:`View` on each request and call
    the :meth:`dispatch_request` method on it.

    The arguments passed to :meth:`as_view` are forwarded to the
    constructor of the class.
    """
```

Fig. 13 Docstring Completion Example

task of *comment completion*. For simplicity, we will refer to both types of structured documentation with the name *docstring*. The completion task that we are tackling is "*given a prefix sequence of a documentation comment, suggest the most likely next word to the developer*". We include an illustrative example in Fig. 13. Imagine that the developer is about to write a complete comment and is currently at the point of the cursor (where the darker text ends). The completion engine should assist the developer in finding the next words, i.e., `function that can`, and so on. We adopt and extend neural language models that were presented in previous work (Ciurumelea et al. 2020) as a baseline for generating completion suggestions.

4.1 Neural Language Models

The task of generating completion suggestions can be solved using language models (LMs), which are statistical models (Jurafsky and Martin 2000) that assign a probability to each possible next word and are also able to assign a probability to an entire sentence. For the example in Fig. 13 if the developer typed `Converts the class into an actual view` such a model can generate possible next words. Some potential ones are `function`, `method` and `object`, but not `the`, `refrigerator` or `view`. These kind of models have applications in a large variety of tasks, such as speech recognition, spelling or grammatical error correction and machine translation among others. Traditionally, this problem was tackled using n-gram language models, which estimated the probability of a sequence by extracting frequency counts from the training corpus. However, n-gram models have several problems due to data sparsity and require complex back-off and smoothing techniques. Additionally, using larger n-gram sizes is very expensive in terms of memory and these models cannot be generalized across contexts. For example, seeing sequences such as "blue car" and "red car" will not influence the estimated probability of "black car". Whereas a neural language model is able to learn that "blue", "red" and "black" all represent the same concept of color. Neural language models are better able to handle all the problems described above, and will in general, have a much higher prediction accuracy than an n-gram model for a particular training set (Jurafsky and Martin 2000).

To build neural language models we use LSTM-based models that process sequences of words one word at a time and return at each time step a fixed sized vector representing the processed sequence so far. Vanilla RNNs are notoriously difficult to train and have problems with exploding and vanishing gradients during training. For this reason they have been replaced by gated architectures such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) units.

Language models can be evaluated using an intrinsic metric called **perplexity**, the perplexity of a language model on a test set represents the inverse probability of the test set normalized by the number of words (Jurafsky and Martin 2000). A better language model is one that assigns a higher probability to the test set, therefore has lower perplexity values. For training and evaluation of such models it is common to use the log transformed version of perplexity, called cross-entropy. An intrinsic improvement in perplexity does not guarantee an improvement in the performance of the task for which a language model is used, therefore it is important to always evaluate such a model using extrinsic metrics, which reflect the performance of the application. In our case, we want to know how well we can predict completion suggestions while a developer is typing a comment, for this we use the Top-k accuracy metric with $k = [1, 3, 5, 10]$. We define the Top-k accuracy as the ratio of correct predictions among the k predicted words with regard to all predictions being made.

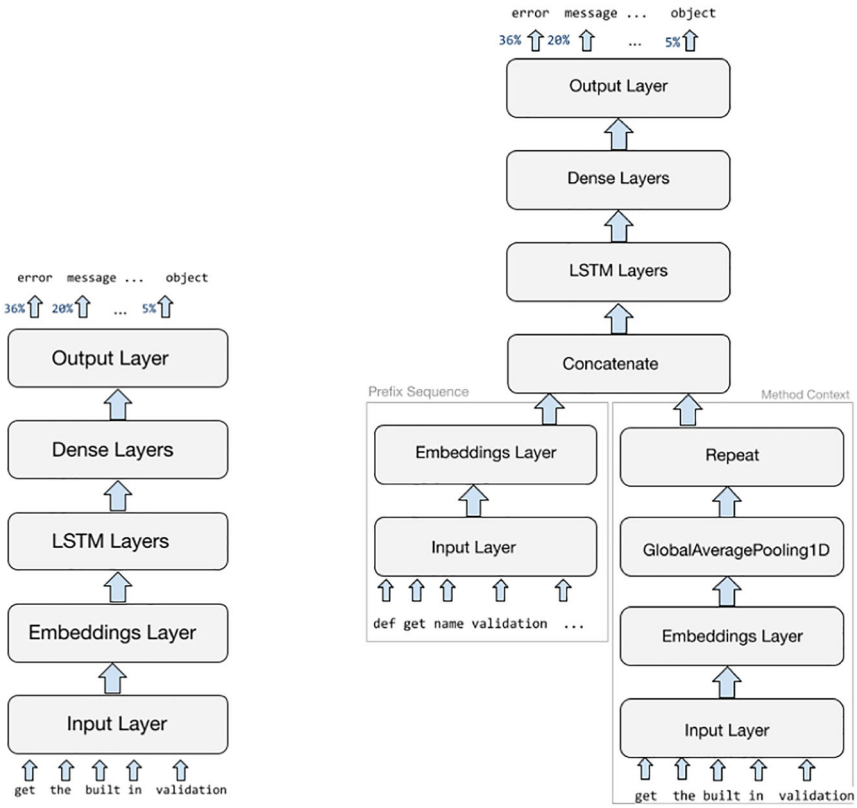
When training language models it is necessary to define a fixed vocabulary at the beginning. This vocabulary can be composed of full words, sub-words or characters, while each type has its advantages and disadvantages. In general, it is not possible to learn all possible words, as some will not appear in the training set at all or will not be frequent enough. The solution is to assign a general *¡UNK¿* token to words that are rare or were not seen during training. This is a reasonable solution for natural language text, as it is less likely to encounter unknown tokens during training and testing. However, in source code developers include new identifier names all the time (Hellendoorn and Devanbu 2017) and dealing with *¡UNK¿* words requires additional measures. For example, splitting identifiers using the snake or camel case convention into individual words.

4.2 Language Models for Documentation Comments

We train five different language models (LM) for the experiments in this paper. We first train a Sequential and a Context-based Language Model on the extracted sections to establish a baseline. For this, we use the same models that have been introduced in previous work (Ciurumelea et al. 2020).

Sequential LM The sequential language model is presented in Fig. 14(a) and is an LSTM model that includes an Input layer, an Embeddings layer, one or more LSTM layers followed by one or more Dense layers interleaved with Dropout layers. The input layer receives a word sequence of length k representing the prefix, for which a completion suggestion should be generated. Then this is followed by an Embeddings layer, this associates with each word of the vocabulary a dense fixed-size real valued vector that incorporates semantic and syntactic information. This layer is necessary, as deep learning models cannot receive as inputs words, only numeric values. What happens in practice is that words are one-hot encoded, that is each word is represented as a sparse vector with a 1 on a single position. The Embeddings layer is added on top of the Input layer, to learn a dense distributed representation for each word from the vocabulary in a predefined vector space. This representation of the words, also called word embedding is learned based on the usage of the words. Words that are used in a similar way will result in having similar representations that can be projected close in the geometric space they are represented. For example, colors or animals will have similar vector representations.

The LSTM layer will process the input sequence word-by-word and keep an internal state during this process. At the end of the sequence it will return a vector representation of the received sequence. This is passed through the Dense layers, to extract further useful features until the final Dense layer, which outputs for each word in the vocabulary the



(a) Sequential Language Model. (b) Context Language Model.

Fig. 14 Language Models from Previous Work

probability that this is the next word. This is a regular language model that only uses the text of the documentation comments in our dataset for training, but as mentioned in (Ciurumelea et al. 2020) documentation comments are accompanied by context information, such as the method body they are documenting.

Context LM Following, we build a Context LM represented in Fig. 14(b), which is a multi-input model that additionally to the prefix sequence also receives the method body as input. The method body is passed through a shared Embeddings layer, then the obtained representation of the method body is passed through a GlobalAveragePooling1D layer. This averages the Embeddings vector representing the method body to obtain a single vector representation of the method body context, which is then repeated and concatenated with each word embedding representation of the prefix sequence. The rest of the model follows the structure of the Sequential LM.

Next, we want to investigate how to leverage the structure information of the documentation comments to improve the suggestions results. For this we have two options: either build a model that receives the section type as additional input, or train section specific

models, that is train a separate model for each of the sections we analysed: *Short* and *Long description*, *Parameters*, *Return* and *Raises* sections, respectively.

Section LM To pursue the first option, we extend the Sequential LM and build a Section LM model represented in Fig. 15, that also receives the section information as input, which can take one of the following categorical values: *Short description*, *Long description*, *Parameters*, *Return* or *Raises*. The section input is passed through an Embeddings layer, with different weights and dimensions than the ones used for the prefix sequence input. This allows the model to learn a more meaningful representation for the sections, for example, it might learn for the *Short description* a vector representation that is more similar to the one learned for the *Long description* than the one for the *Raises* sections. The embedding representation of the section input is repeated and concatenated with each embedding vector for the prefix sequence words which are then passed through the layers described previously.

Section-Context LM Our fourth model is a Section-Context LM model that receives as input the prefix sequence, the method body context and the section type for which it should

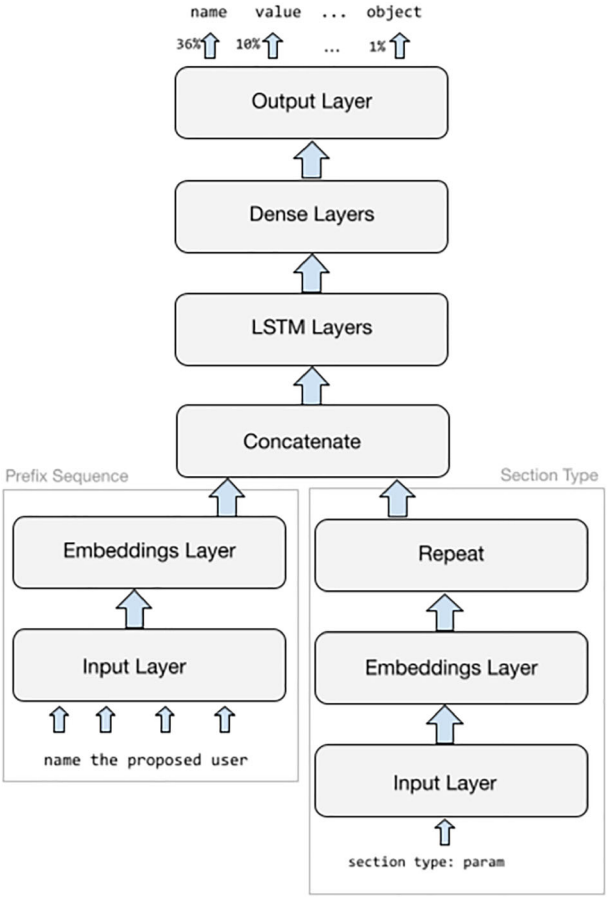


Fig. 15 Section Language Model

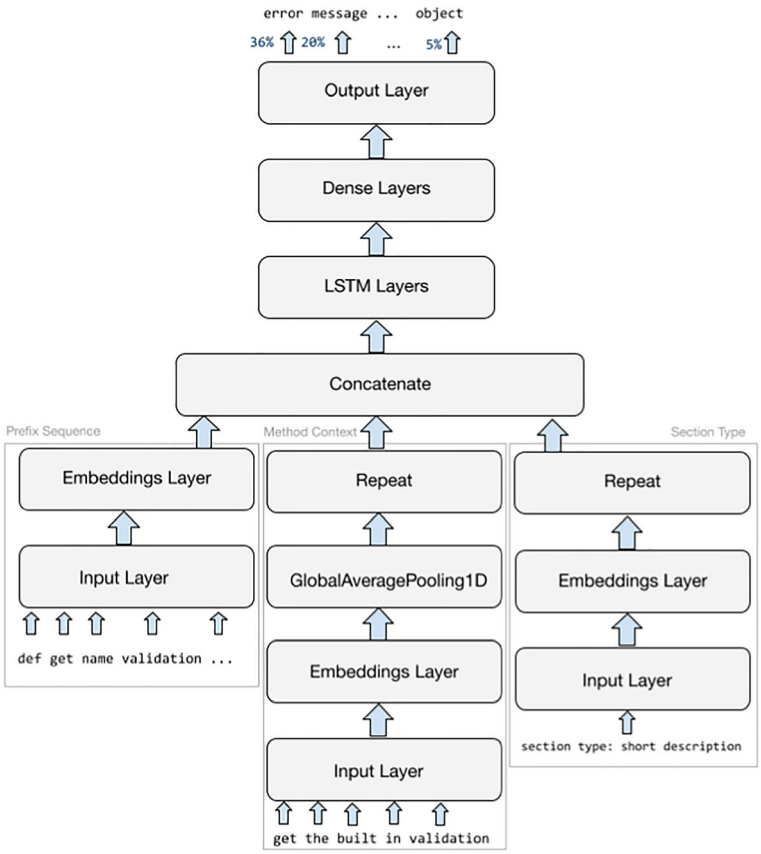


Fig. 16 Section-Context Language Model

learn to generate a completion suggestion. This model is included in Fig. 16. It concatenates the embedding representation for each word in the prefix sequence with the obtained method context and the section types representations. This information is then passed through the LSTM layers for each word of the prefix sequence, and the representation at the end of this sequence is passed through the Dense layers up to the final Output layer.

Section specific Context LMs Additionally, we train section-specific models using the Context LM architecture 14(b) using as training data only the corresponding sections. For example, for the *Short description* we train a model that only receives as input the prefix sequences generated from the extracted *Short description* sections together with the method body context. As a result, we obtain 5 Section models that are then evaluated on the corresponding section test sets. We will refer to these models as *Section specific* Models in the rest of the paper.

5 Dataset Creation and Training Details

We selected Python and Java as two popular representatives for two different programming paradigms to investigate the performance of our completion engine in different

programming languages. Additionally, both languages are known for recommending a formatting style when writing documentation comments. In the following, we describe which datasets we use for training and evaluating the models, how we extracted the training instances and finally provide some additional details about the models themselves. For Python, we collected a larger and more diverse dataset for training the language models. However, for Java we keep the CodeSearchNet dataset as this should be sufficient for our goals, to understand if our results can be generalized to other programming languages.

5.1 Documentation Comments Datasets

Python Dataset Collection We build a new Python dataset from the GitHub dump available on the BigQuery platform through the Google Cloud Public Datasets Program (Google cloud public datasets 2021). At the time we queried BigQuery for this particular dataset, it had last been updated on the 20^{th} of March 2019. We collect all Python files that contain at least one instance of triple quotes, which could potentially represent a docstring. Here, we would like to note that some of the projects were relatively old and might not use the newer recommendations. Additionally, by only looking at triple quotes, we might have missed the comments that uses single triple quotes. However, this was only common for older projects. This resulted in a very large number of files and projects of varied quality and characteristics. To ensure that the quality of our dataset is sufficient, we queried GitHub using the REST API (Github rest api 2022) for additional commit information and filter the projects based on the following characteristics:

- no forks (not marked as a fork of another project);
- include at least 10 Python files;
- commits data: it includes at least 10 commits, and these commits include at least 2 different author emails
- include at least 100 docstring - function pairs.

The filtering criteria have been chosen as minimal characteristics for excluding toy projects. For example, we have chosen at least 10 commits with at least 2 different emails as criteria to ensure a minimum amount of activity in the project without excluding too many projects. However, it is possible that a GitHub user uses two different emails when committing changes to a repository, thus resulting in a false positive. We also ignore projects which have been removed from GitHub or for which we were not able to obtain the commit information. After performing all these filtering steps we obtain a list of 14,507 projects, nevertheless we noticed that there was some duplication among the projects. While we filter out the forks, some users copy a specific project then create a new repository on GitHub using this project and give it a different name. By using the git root hash we remove these cloned projects and obtain a final list of 9,743 projects that comprise the Python dataset. By looking at the git root hash, we were able to filter out a substantial number of cloned projects. These were projects that copied another popular project and contained minimal changes to it. However, it might still be possible that we missed some of them, or that a project includes the source code of another project.

Java As we previously mentioned, for Java we use the dataset, that we analysed in Section 3 of the paper. This is the one provided as part of the CodeSearchNet challenge (Husain et al. 2019a).

5.2 Training Data Extraction

To train and evaluate machine learning models, it is necessary to prepare the data and convert it to training instances in the format expected by the models. Additionally, to ensure that our evaluation results are valid, we should follow procedures such as splitting the dataset into a training, validation and testing sets, and use cross-fold validation. Employing k-fold cross-validation for training and evaluation mitigates the risk that the results are caused by spurious patterns in the data partitioning.

The dataset is split into 5 folds, while 4 folds are used for training, and the fifth fold is equally split into validation and test sets. We are using 5 folds instead of the more traditional 10 folds because of limited computational resources. We follow the guidance from Le Clair and McMillan (2019) and split the data on a repository level granularity. This means that repositories that appear in the training set will not appear in the validation/test sets and vice-versa. Splitting the data on a function level granularity would lead to a boost of our results as functions from the same project would be found both in the training and test set, therefore making it easier for the neural language models to generated completions for them. Please note, that the partitioning on a project level is different to previous work (Ciurumelea et al. 2020), which split the dataset at a function level.

For the collected Python dataset, we check and eliminate function-docstring exact duplicates pairs between the train and validation/test sets. In this way we are ensuring that the results are not inflated because of auto-generated code and comments or snippets of code that are copy-pasted between projects. For the Java dataset this has already been done (Husain et al. 2019a).

In Table 9 we include the number of repositories, files and extracted function-docstring pairs averaged over the 5 folds for the Python and Java datasets. We would like to note, that as some of the projects have a disproportionate number of docstrings, we only keep a maximum of 500 docstrings per project for Python that were randomly selected. Additionally, for each documentation comment we extract the sections and obtain the total counts as included in Table 10 averaged over the 5-folds for the separate dataset partitions.

We can observe by comparing the number of repositories and extracted function-docstring pairs for the Python and Java datasets described in Table 9, that the Java dataset is smaller. From the statistics describing the number of extracted sections included in Table 10 for the Java dataset, we notice that these are better balanced and we expect that this will have an effect on the training of the neural language models. This observation confirms the

Table 9 Training/Validation/Testing Datasets Statistics

Dataset	Repositories	Files	Method-Docstrings Pairs
Python			
Train	7,439	270,498	1,415,297
Validation	960	40,806	211,868
Test	963	40,528	213,181
Java			
Train	3,674	93,171	327,112
Validation	457	9,862	34,980
Test	461	13,453	46,797

Table 10 Extracted Sections Statistics (5- fold average)

Section type	Train count	Validation count	Test count
Python			
Short description	1,370,770	205,576	207,041
Long description	394,098	62,170	62,786
Parameters	790,204	115,192	113,115
Raises	38,392	5,438	5,484
Returns	233,090	33,435	32,960
Java			
Short description	327,111	34,980	46,797
Long description	103,044	10,852	14,908
Parameters	407,753	44,293	57,645
Raises	95,768	7,584	16,357
Returns	163,547	17,096	23,790

findings from the Section 3, that Java comments include more often the different sections and rarely contain only a summary of the method as a documentation comment.

Preprocessing and Generation of Training Instances For creating the training instances we use the sections extracted from the docstring-function pairs. For each of the *Short description*, *Long description*, *Parameters*, *Return* and *Raises* sections we perform the following preprocessing steps:

1. remove non-English docstrings, which were detected using the Python langdetect library (Langdetect 2021) as mentioned in Le Clair et al. (2019);
2. remove doctests (Doctest — test interactive python examples 2021) from the section content, these correspond to interactive Python sessions and contain snippets of code and the expected results, they are not very common and would likely lead to worse results. This step is only applied for the Python dataset.
3. split identifiers based on the camel and snake case conventions, this has been shown in previous work to reduce the necessary vocabulary size (Hu et al. 2020) for training neural language models and as a reasonable solution for the problem of Out-Of-Vocabulary words for source code comments;
4. lowercase all characters;
5. replace any characters that do not satisfy the following regular expression: [A-Za-z0-9.,!?:; \n] with an empty space;
6. replace the punctuation signs with a corresponding special symbol (e.g.,the dot character is replaced with <punct.>), we keep the punctuation signs as we believe they are important signals for the LMs;
7. multiple whitespaces are replaced by a single one;
8. add a <sos> (start of sequence) and an <eos > (end of sequence) token to each extracted section.

The summary statistics for the obtained lengths in tokens of the preprocessed sections for the datasets are included in Table 11.

Table 11 Preprocessed Sections Lengths Statistics

Section	Mean	Std	Min	25%	50%	75%
Python						
Full Comment	41.04	76.63	1.00	9.00	18.00	45.60
Short description	11.30	9.85	1.00	7.00	9.00	13.00
Long description	41.04	62.15	1.00	12.60	23.00	46.00
Parameters	15.58	27.57	0.00	7.00	11.00	17.00
Returns	13.31	49.65	0.00	4.00	8.00	14.00
Raises	15.45	15.58	0.00	9.00	12.00	18.40
Java						
Full Comment	50.67	68.75	1.00	17.00	34.00	62.00
Short description	12.19	9.44	1.00	7.00	10.00	15.00
Long description	37.95	53.14	1.00	13.00	22.00	43.00
Parameters	10.40	10.72	1.00	6.00	9.00	12.00
Returns	10.26	9.46	1.00	5.00	9.00	13.00
Raises	11.60	14.73	0.00	6.00	10.00	14.00

Next, we describe how the extraction, preprocessing and training instance generation works using as an example the function-docstring pair illustrated in Listing 1. We include in Table 12 the extracted sections, while in Table 13 we include the corresponding preprocessed sections. The *Short* and *Long descriptions* will be extracted as free text from the docstring if they exist, while the *Parameters* are represented as a list of tuples. Each tuple will contain the identified name, the description, the type if it is included, and, for Python, a boolean flag that indicates whether the parameter is optional. The *Return* section is a tuple containing the type and the corresponding description, while the *Raises* section can contain a list of tuples containing for each extracted exception the type and the description. The method body is split into tokens, which are then preprocessed using steps 3 to 7 from above and the results are then fed as context input for the Context-based LMs presented earlier.

Using the preprocessed section, we generate the training instances, these contain sequences of length k, the preprocessed method body and the corresponding section type. The neural language models will be trained to predict the k^{th} word, given the prefix sequence of length k - 1, and additional input depending on the model type. We apply a sliding window

Table 12 Extracted Docstring Sections Example

Section type	Section content
Short Description	Returns the cross entropy loss of the classifier on images.
Long Description	None
Parameters	[('images', 'A minibatch tensor of MNIST digits. Shape must be [batch, 28, 28, 1].', None, None), ('one_hot_labels', 'The one hot label of the examples. Tensor size is [batch, 10].', None, None)]
Returns	(None, 'A scalar Tensor representing the cross entropy of the image minibatch.')
Raises	None

Table 13 Preprocessed Docstring Sections Example

Section type	Section content
Short Description	returns the cross entropy loss of the classifier on images <punct.>
Long Description	None
Parameters	[('images', 'a minibatch tensor of mnist digits <punct.> shape must be batch <punct,> 28 <punct,> 28 <punct,> <newline> 1 <punct.>', None, None), ('one hot labels', 'the one hot label of the examples <punct.> tensor size is batch <punct,> <newline> 10 <punct.>', None, None)]
Returns	(None, 'a scalar tensor representing the cross entropy of the image minibatch <punct.>')
Raises	None

approach to extract the sequences of length k for the *Short* and *Long description* sections. However, the other sections are lists of tuples or tuples, therefore we first assemble them into strings before extracting the sequences of length k. For example, for each extracted parameter we concatenate the identifier name, the type and optional/not optional qualifier if it exists, a special token <sod > (start of description) and finally the description. A similar process is used for the *Return* and *Raises* sections.

In Table 14 we include several of the resulting training instances corresponding to the current example. The first column includes the preprocessed method body, the second one includes the section type and the final one includes the docstring sequence. For these examples, the model would receive the sequence without the last word as input and it would learn to predict the last word. In practice, we train the model on all possible subsequences that we can generate from a comment. We follow the previous work (Ciurumelea et al. 2020) and use a value of 5 for k, as this allows us to predict words early on. The size of the preprocessed method body is limited to the first 100 tokens, which also follows the example of previous work (Hu et al. 2020) in automated summary generation, as neural networks generally have problems with learning long sequences.

5.3 Model Configuration and Training Details

We trained the described models for 20 epochs with early stopping, and the models were evaluated after each epoch on the validation set. The model that performed best on the validation set was then evaluated on the test set by computing the Top-k accuracy. Here, we

Table 14 Training Instances Example

Method body	Section type	Docstring sequence
def mnist cross entropy images...	Short Description	<sos> returns the cross *entropy*
def mnist cross entropy images...	Short Description	returns the cross entropy *loss*
def mnist cross entropy images...	Parameters	<sos> images <sod> a *minibatch*
def mnist cross entropy images...	Parameters	images <sod> a minibatch *tensor*
def mnist cross entropy images...	Returns	<sos> <sod> a scalar *tensor*
def mnist cross entropy images...	Returns	<sod> a scalar tensor *representing*

would like to mention that testing instances for which the predicted word is a special token, like the $< UNK >$ or any of the punctuation tokens, are ignored during the evaluation on the test set. We used the Tensorflow and Keras libraries to implement our models and trained them using a GPU. For the training, we use the following hyperparameters for the neural language models:

- a size of 512 for the prefix and method body Embedding layers, while a size of 64 is used for the section type Embedding layer;
- an LSTM layer with size 256, followed by a Dense layer with size 128 and finally the Output layer with dimension equal to 30,000, as the vocabulary size, we also add a Dropout layer between the Dense and Output layers with a dropout probability value of 0.2;
- training is done using the Adam optimizer with a learning rate of $3e-4$;

This model configuration is similar to the previous work, we have only set the dimensionality slightly higher to accommodate for our larger dataset. We have not performed further hyperparameter optimization because of time and computational resources constraints. This should not affect our findings, since the goal of our work was to investigate whether the structure of documentation comments could be leveraged for the problem of generating completion suggestions. However, further hyperparameter optimization could results in better results.

6 Can Structural Information Improve Completion Accuracy? (RQ2)

The focus of this research question is to understand the effect of using the section information on the completion performance. We will investigate two aspects: *"1) does using the section information improve the generated completion suggestions?"* and *"2) are there significant differences between the completion predictability among the sections?"*. To limit the scope of the research question, we first focus only on Python docstrings. The comparison to the Java results will then be performed in the next research questions.

Methodology To evaluate the trained language models we compute the Top-1, Top-3, Top-5, Top-10 accuracy of the generated suggestions for the test sets averaged over the 5 folds. There are several options for evaluating machine learning models, and language models in particular. We choose the Top-k accuracy as it allows us to directly measure how well the models work for our task. This metric has been used in a paper (Zhou et al. 2022) that evaluates deep learning models for the task of code completion, which is similar to our task. We are interested in understanding if the results returned by the model are correct, the most important value is the Top-1 accuracy, as likely developers will not spend time analysing a list of 10 words to chose the correct next word. However, looking at the Top-3, Top-5 and Top-10 accuracy allows us to understand the potential of the models, as additional post-processing steps could allow us to improve the prediction results. For example, we might decide to suggest a parameter or variable name on the first position if this appears in the method body and the model suggests it in the Top-5, but not as the Top-1 prediction.

We include the results of our evaluation in Table 15, describing the results obtained for the Sequential, Section, Context, Context-Section Language Models and for the Section specific Models. We select the Context LM as the base model and include for all the other models in parentheses the relative change, in percentages compared to this model. The relative change is computed as the difference between the new value and the reference value,

Table 15 Top-k Accuracy for Python Docstrings

Model	Short Description	Long Description	Parameters	Return	Raises
Top-1					
LM	0.180 (−12.7%)	0.182 (−6.3%)	0.251 (−4.9%)	0.247 (-6.6%)	0.307 (−4.9%)
Section LM	0.183 (−11.0%)	0.182 (−6.2%)	0.256 (−3.1%)	0.241 (-9.0%)	0.335 (+4.0%)
Context LM	0.206	0.194	0.264	0.264	0.323
Context-Sec LM	0.210 (+2.0%)	0.194 (+0.1%)	0.270 (+2.5%)	0.273 (+3.3%)	0.348 (+8.0%)
Sec specific Models	0.221 (+7.3%)	0.202 (+4.1%)	0.284 (+7.6%)	0.290 (+9.6%)	0.349 (+8.3%)
Top-3					
LM	0.286 (−13.6%)	0.286 (−7.2%)	0.373 (−5.7%)	0.361 (−7.9%)	0.422 (−6.5%)
Section LM	0.294 (−11.2%)	0.287 (−6.8%)	0.380 (−4.0%)	0.356 (−9.2%)	0.456 (+1.1%)
Context LM	0.331	0.308	0.396	0.392	0.451
Context-Sec LM	0.339 (+2.6%)	0.308 (−0.1%)	0.405 (+2.3%)	0.400 (+1.9%)	0.483 (+7.1%)
Sec specific Models	0.352 (+6.5%)	0.315 (+2.0%)	0.414 (+4.7%)	0.402 (+2.6%)	0.458 (+1.5%)
Top-5					
LM	0.341 (−13.8%)	0.346 (−7.4%)	0.435 (−6.2%)	0.418 (−8.5%)	0.480 (−6.9%)
Section LM	0.349 (−11.7%)	0.350 (−6.6%)	0.443 (−4.4%)	0.417 (−8.7%)	0.512 (−0.6%)
Context LM	0.395	0.374	0.463	0.457	0.515
Context-Sec LM	0.404 (+2.3%)	0.375 (+0.3%)	0.473 (+2.0%)	0.463 (+1.4%)	0.546 (+6.0%)
Sec specific Models	0.416 (+5.4%)	0.381 (+1.8%)	0.478 (+3.2%)	0.456 (−0.1%)	0.509 (−1.3%)
Top-10					
LM	0.427 (−13.4%)	0.436 (−7.6%)	0.523 (−6.4%)	0.502 (−8.4%)	0.562 (−7.0%)
Section LM	0.433 (−12.2%)	0.440 (−6.9%)	0.530 (−5.1%)	0.500 (−8.8%)	0.586 (−3.0%)
Context LM	0.493	0.472	0.558	0.548	0.604
Context-Sec LM	0.500 (+1.4%)	0.474 (+0.4%)	0.565 (+1.3%)	0.554 (+1.1%)	0.633 (+4.7%)
Sec specific Models	0.509 (+3.1%)	0.476 (+0.8%)	0.564 (+1.0%)	0.533 (−2.8%)	0.578 (−4.4%)

divided by the reference value, where the new value is the Top-k accuracy obtained for a specific model and the reference value is the Top-k accuracy obtained for the base model, that is the Context LM model. The values included in the table are rounded for reasons of space, however the relative change is computed on the exact values.

For completion, we also include a metric traditionally used for evaluating summarization models, the ROUGE-L score (Lin 2004). ROUGE-L is part of the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) set of metrics used for evaluating automatic summarization and machine translation software in natural language processing. This metric measures how similar a candidate (predicted) sequence is to the reference (expected) sequence and includes several variants based on the length of the n-grams it considers for measuring the similarity. We focus on the ROUGE-L score that measures the longest common subsequence(LCS) between a reference and the predicted output and computes the corresponding precision, recall and F1-Score. The formulas for computing these scores are the following:

$$Precision = \frac{|LCS|}{|prediction|}$$

$$Recall = \frac{|LCS|}{|reference|}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

For example, for a reference sequence such as *"helper function to start the server"* and the prediction *"helper function to return the server"* the word *"return"* is incorrect and the LCS length is equal to 5 words and the corresponding F1 = 0.833.

We would like to note that our use case is different than the one for which the ROUGE metric was developed, as we focus on generating completion suggestions on a word level and not the full sequences, therefore we could not directly compare our approach to others that focus on generating full summaries. The ROUGE metric is typically used to evaluated summarization and machine translation models such as the more recent T5 Model (Raffel et al. 2019) which are more complex than our models. In our current work we address the more classical recommender systems use case and recommend one word at a time, and plan on addressing longer completions in the future.

The F1 score is computed for each method on a section level, we concatenate the predictions of the model on a word level for a particular section and compare it with the reference section and use the rouge-score (Python rouge implementation 2022) open source package to calculate the scores. This is similar to the scenario where the developer is typing the comment and the model suggests a complete word and the developer either accepts it or types the correct one. We compute these scores per section, which we then average for each model and section and report the final results in Table 16. We only present the scores for the Context, Context-Section and the Section specific Models, as these are expected to have the more promising results.

Results From the evaluation results included in Table 15 we can observe that there are significant differences between the obtained accuracy of the different sections. For the base model, the Context LM Model, we note that it obtains the lowest Top-1 accuracy for the *Long* and *Short descriptions* sections. The evaluation results for the *Parameters* and *Return* sections are equal, while the results obtained for the *Raises* section is significantly higher. This indicates that some sections are easier to complete than others, as they are more predictable. For example, the most challenging to complete is the *Long description*, followed by the *Short description*. This is likely due to the fact that the content of the *Long description* is quite heterogeneous and longer, developers might expand the *Short description*, include additional explanations about how a particular function or algorithm works,

Table 16 ROUGE-L F1 Scores for Python Docstrings and Javadocs

Model	Short Description	Long Description	Parameters	Return	Raises
Python					
Context LM	0.232	0.229	0.327	0.279	0.340
Context-Sec LM	0.238 (+2.3%)	0.228 (−0.5%)	0.337 (+3.0%)	0.285 (+2.1%)	0.368 (+8.1%)
Sec specific Models	0.252 (+8.3%)	0.229 (+0.3%)	0.347 (+6.1%)	0.266 (−4.7%)	0.358 (+5.2%)
Java					
Context LM	0.265	0.229	0.403	0.302	0.362
Context-Sec LM	0.257 (−2.9%)	0.231 (+1.3%)	0.423 (+5.0%)	0.296 (−1.9%)	0.407 (+12.5%)
Sec specific Models	0.251 (−5.0%)	0.216 (−5.5%)	0.415 (+2.9%)	0.267 (−11.4%)	0.374 (+3.4%)

while sometimes they might either include the parameters, raises or returns information in a free-text form or using a custom formatting style. The *Parameters*, *Return* and *Raises* sections are easier to complete. This is probably caused by the more homogeneous content of these sections, as they typically explain what a particular parameter, return or raises values represent.

The Rouge-L F1 scores confirm our initial observations, that there are significant differences between the values obtained for the different sections of the Python dataset. The section for which we observe the highest scores is the *Raises* one, while also for this metric the hardest to predict is the *Long Description*. The Sec specific Models seem to lead to better results, however this does no apply for the *Return* section, and further analysis is needed to investigate why.

Qualitative Analysis For the Python dataset, we observe that we obtain the largest gain by training Section specific Models, however this gain is more pronounced for the Top-1 results. To improve our understanding of these differences, we include several examples from the test set where the Section specific Models are able to return the correct word on

Table 17 Comparison Between Section specific and Context LM Predictions for Python (The grey part of the docstrings is provided as context information to the reader, but is not part of the input, the expected target word is highlighted in bold)

Nr.	Partial Docstring	Input Sequence	Section Models Top-10 Predictions	Context LM Top-10 Predictions
1	Return "True" if all items in this **«sequence»** are integers or non-negative integers.	all items in this	**«sequence»**, list, array, set, integer, range, <punct'>, seq, object, <newline>	list, **«sequence»**, sentence, set, object, word, index, <newline>, generator, array
2	If the logger has no **«handlers»**, call basicConfig()	logger has <newline> no	**«handlers»**, effect, content, <punct'>, name, such, children, handler, spaces, arguments	effect, arguments, **«handlers»**, exception, error, longer, message, errors, args, parameters
3	:param filepath: the file path to be **«opened»**	file path to be	**«opened»**, written, read, moved, copied, added, closed, created, open, compressed	written, read, **«opened»**, created, deleted, uploaded, <newline>, used, moved, <unk>
4	:return: A hash object of the same **«type»**	object of the same	**«type»**, size, length, format, subtype, shape, order, as, space, structure	class, object, **«type»**, hash, <newline>, name, instance, data, connection, as
5	TypeError - when any of the parameters are of the **«wrong»** type	parameters are of the	**«wrong»**, expected, invalid, <newline>, required, correct, parameters, os, provided, given	same, form, correct, **«wrong»**, format, <newline>, signature, following, type, expected

the first position, while the Context LM model is not, in Table 17. We include on each row an example belonging to the individual sections in the following order *Short description*, *Long description*, *Parameters*, *Return* and *Raises*. We omit the method body that both the Section specific and the Context LM models receive as input for reasons of space. The Section specific Models are Context LM models trained on the dataset specific to a section, while the Context LM model is trained on all the available data.

The first row includes an example from a *Short description*, interestingly the docstring starts with the word "Return". The developer tried to summarize what the method does, by indicating what values are returned and in what conditions. Such a description would be more appropriate for the *Return* section, however the docstrings in our dataset are of mixed quality and it is a positive sign that the models still work in such cases. The *Short description* specific Section model was able to return the correct word sequence as the first prediction, while the Context LM model only returned this word on the second position. It is difficult to find an intuition, for why the *Short description* specific model was able to predict the correct word on the first position, while the other model did not. Likely, for this section the word "sequence" is more common than the more specific word "list".

In all the included examples, we can notice that the Section specific Models predict the correct word on the first position, nevertheless the completions recommended by the Context LM are reasonable completions although they are not an exact match. However, the correct word is included in the 10 predictions also for the Context LM. This observation indicates that the Section specific Models learn a better ranking of the results, as opposed to the general Context LM, by being trained on the section data. Likely, for the individual sections, the models learn that some completions are more likely than others. For example, for the *Raises* section example included in row 5, the *Raises* specific Model learns to predict the word "wrong" to complete the prefix sequence "parameters are of the". When describing error conditions, it is common that developers use negative words more often, than for the other sections. Therefore the model learned to predict the correct word, instead of the word "same" as the Context LM model did.

To answer our second research question (RQ2), we can observe that there are significant differences for the prediction accuracy of the various sections of a documentation comments for the Python dataset. Additionally, using the section information when training neural language models for generating suggestion completions increases the prediction accuracy, especially for the first predicted token. For Python, the Section specific Models achieve the highest improvement in the accuracy of the generated suggestions.

7 How Language-Dependent Are The Evaluation Results? (RQ3)

This section focuses on completion suggestions for Java docstrings from the second dataset we introduced in Section 3. We train several language models for the task of *documentation comment completion* and evaluate them separately on the different docstring sections. We would like to investigate if we can observe similar findings as for the Python dataset: differences between the prediction accuracy of the various sections on the test set and if using the section information can improve the prediction results.

The *methodology* of this research questions is the same that we have applied in the previous section for Python.

Results We trained and evaluated a Context LM, Context-Sec LM and Section specific Models for the Java dataset and include the evaluation results in Table 18. We do notice, as for our Python Context LM Model, that the obtained Top-k accuracy values vary across the different sections. The *Long description* is the most difficult to complete, while the *Short description* is the second hardest. The best results are obtained for the *Parameters* section, followed by the *Raises* and *Return* sections. Surprisingly, the Section specific Models obtain worse evaluation results than the base model. We believe, this is caused by the smaller dataset, especially when the section datasets are used separately for training. This can be confirmed also for the ROUGE-L F1 scores that we included in Table 16, as the Section specific Models have the lowest values. Additionally, using the section information only helps for the *Parameters*, *Raises* and slightly for the *Long description* sections.

For the completion suggestions predicted on the first position (Top-1 accuracy), the Context-Sec LM model performs best, except for the *Return* section where it shows a relative decrease in accuracy of 0.6%. However, the results are improved for the results returned on the first 3 positions (Top-3 accuracy), where the Context-Sec LM model outperforms the Context LM models for all sections. The differences of the results between the Python and Java models can either be caused by specifics of the programming languages or by the characteristics of the two datasets used for training. We plan on further investigating the cause as future work. A reasonable hypothesis is that the training instances are more balanced among the different sections for Java. Therefore the Context-Sec LM benefits from being trained on more data including all the sections but is also able to leverage the section information and improve the accuracy results compared to the Context LM model. For Python, the data is enough for the different sections to obtain good results when training section specific

Table 18 Top-k Accuracy for Javadocs

Model	Short Description	Long Description	Parameters	Return	Raises
Top-1					
Context LM	0.222	0.192	0.325	0.270	0.317
Sec specific Models	0.213 (–4.1%)	0.187 (−3.0%)	0.326 (+0.5%)	0.244 (−9.5%)	0.310 (−2.3%)
Context-Sec LM	0.223 (+0.2%)	0.197 (+2.3%)	0.335 (+3.3%)	0.268 (−0.6%)	0.342 (+7.8%)
Top-3					
Context LM	0.353	0.311	0.463	0.402	0.469
Sec specific Models	0.345 (−2.4%)	0.300 (−3.4%)	0.456 (−1.5%)	0.367 (−8.6%)	0.439 (−6.3%)
Context-Sec LM	0.364 (+2.9%)	0.321 (+3.3%)	0.480 (+3.6%)	0.408 (+1.5%)	0.495 (+5.6%)
Top-5					
Context LM	0.421	0.376	0.530	0.467	0.541
Sec specific Models	0.408 (−3.2%)	0.360 (−4.2%)	0.517 (−2.5%)	0.430 (−7.8%)	0.500 (−7.6%)
Context-Sec LM	0.432 (+2.6%)	0.386 (+2.7%)	0.545 (+2.8%)	0.476 (+2.0%)	0.565 (+4.4%)
Top-10					
Context LM	0.516	0.466	0.613	0.556	0.633
Sec specific Models	0.498 (−3.5%)	0.444 (−4.9%)	0.592 (−3.5%)	0.515 (−7.3%)	0.575 (−9.1%)
Context-Sec LM	0.526 (+2.0%)	0.474 (+1.6%)	0.625 (+1.9%)	0.569 (+2.4%)	0.650 (+2.8%)

models. And likely, the imbalanced sections make it harder for the Python Context-Sec LM to learn to take advantage of the section information.

Qualitative Analysis We include for the Java dataset a couple of examples that compare the results returned by the Context LM and the Context-Sec LM models included in Table 19. The rows correspond in order to a *Short description*, *Long description*, *Parameters*, *Return* and finally *Raises* examples. We include examples for which the model using the section information returns better results than the base model and interestingly the section information helps the models to improve the ranking of the results. While, it is often difficult to explain the predictions of large neural language models, as the ones we train, we believe that some completions are more common for certain sections than for others. For instance, for the example number 3, the Context-Sec LM model completes the prefix "the class of the" with the word «object», while the Context LM model predicts the word "given". While

Table 19 Comparison Between Section and Context LM Predictions for Java(The grey part of the docstrings is provided as context information to the reader, but is not part of the input, the expected target word is highlighted in bold)

Nr.	Partial Docstring	Input Sequence	Context-Sec LM Top-10 Predictions	Context LM Top-10 Predictions
1	Determines number of **«differences»** (substrings that are not equal) between two strings.	<sos> determines number of	**«differences»**, results, difference, diff, elements, all, strings, steps, two, values	elements, strings, difference, patterns, occurrences, **«differences»**, characters, values, partitions, words
2	If this button isn't attached to any parent **«view»**	attached to any parent	**«view»**, of, <punct.>, link, folder, <punct,>, and, page, item, parent	<punct.>, **«view»**, <punct,>, component, and, of, link, group, or, <eos>
3	@param cl the class of the **«object»** that needs to be serialized.	the class of the	**«object»**, class, serialized, serializer, type, json, data, result, bean, jdbc	given, specified, **«object»**, class, link, input, value, data, requested, property
4	@return an array of validation error **«messages»**	array of validation error	**«messages»**, <eos>, message, objects, information, codes, types, names, <punct.>, <punct,>	<punct.>, <eos>, message, **«messages»**, to, <punct,>, is, occurs, codes, occurred
5	@throws JsonParseException if there is an IOException or if the specified text is not **«valid»** JSON	specified text is not	**«valid»**, a, in, supported, an, found, readable, available, compatible, present	a, null, **«valid»**, supported, found, an, in, the, possible, present

over all the sections, the completion "given" is more likely, for the *Parameters* section the word «object» is more common.

To answer our third research question (RQ3), we can conclude that there are significant differences between the accuracy of the generated completion suggestions belonging to different sections even for the Java dataset. While using the section information can improve the prediction accuracy of the suggested completions, the best way of achieving this seems to be language specific and likely depends on the characteristics of the programming language and the training dataset.

8 Related Work

Work related to our research can be split into two main categories: *studies related to the content of comments* to understand the types of comments developers write, what kind of programming constructs they accompany and their frequency and *approaches for automatically generating comments*, mainly as summaries of the corresponding source code.

8.1 The Content of Source Code Comments

The content of source code comments was studied by Haouari et al. (2011), who conducted two studies: the first one looked at the distribution and frequency of comments depending on different programming constructs and noticed that method comments are the most common. While during the second study, participants looked at the content and relevance of comments and observed that most of them refer to the subsequent code and that method declarations are well explained. Developers seem to prioritize such comments, this supports our decision to focus on method and function comments. The quality of comments was investigated by Steidl et al. (2013) through a semi-automatic method. One of their metrics computes the coherence between code and comments based on the Levenshtein distance between the words in the method name and in the comment: if the two are too similar then the comment is likely trivial and not useful; if they are too different then either the identifier name needs to be refactored or the comment is not sufficient.

Pascarella and Bacchelli (2017) built a detailed taxonomy to identify the purpose of Java comments by analysing 6 open-source Java projects. They built and evaluated an automated approach for classifying comments according to their taxonomy and noticed that although *summary* comments are quite common, they only represent 24% of the overall comments. Two of the identified categories: *expand* which provides details on the code itself and *rationale* which explains the reason behind some choices, patterns and options, confirm our belief that comments contain more than just the summaries of the accompanying source code.

A more recent paper (Aghajani et al. 2019) analysed the issues that developers face with documentation by qualitatively evaluating documentation related artifacts collected from mailing lists, StackOverflow discussions, issue repositories and pull requests. They identified as important issues the lack of completeness and outdated documentation. In a subsequent work (Aghajani et al. 2020) the authors conduct surveys with practitioners

to understand issues with documentation from their perspective and identify the types of documentation that are considered to be most relevant. The majority of study participants identified *lack of time to write documentation* as an important issue that they frequently encountered. Additionally, code comments were described as highly useful for development and testing tasks. They also mentioned that keeping comments updated and consistent with the source code to be very important. Differently from previous work, we study the structure of documentation comments to understand if they follow the formatting styles recommended for a specific programming language. We investigate if such comments include the sections that we expect them to include, however we do not look at the content as this would be very difficult to do manually for the large datasets we consider.

8.2 The Automated Generation of Source Code Comments

Researchers have long identified that developers need support in writing comments and focused on the automated generation of comments in the form of short summaries from the accompanying source code. Initially, researchers have tried to solve the task of source code summarization by applying text retrieval methods, such as Vector Space Model (Salton et al. 1975) and Latent Semantic Indexing (Landauer et al. 1998), to generate term-based summaries (Haiduc et al. 2010) of Java methods and classes from code and inline comments. A subsequent work (Moreno et al. 2013) generated human readable summaries of Java classes based on fixed templates. Another paper (Sridhara et al. 2010) presents an approach to produce descriptive summary comments for Java methods using a set of heuristics to select important statements from a method body and then templates to generate corresponding natural language summaries. While this initial work on the automated generation of documentation was promising, the generated summaries were of limited quality and usefulness. More recent approaches are data-driven and take advantage of the large number of open source projects available online.

One of the earliest works that try to summarize code snippets using full data driven approaches is the one by Iyer et al. (2016). The authors build a model called CODE-NN, which is a neural network trained on StackOverflow C# and SQL code snippets to predict the corresponding title. They are the first to try to summarize code using neural networks, nevertheless they do not use actual source code, but StackOverflow data. In Hu et al. (2018) the authors used a machine translation based approach to automatically generated code comments for Java methods. Their approach, DeepCom, translates the source code to a high-level description of the method. They used the ASTs of the methods as code representation and convert this to a structured-based traversal (SBT) representation before passing it to the machine translation model to generate the method summary. Their results are promising and improve on the ones obtained in Iyer et al. (2016). Wan et al. (2018) develop an approach to generate summaries for Python methods by combining a sequence and AST representation of the source code snippets into a deep reinforcement learning framework using an actor-critic network. In Le Clair et al. (2019) the authors develop a neural machine learning model that receives separately as input and then combines two types of information for representing the source code: a word representation that treats the code as text and an AST representation. Using this model they train and evaluate it on a large dataset of Java methods to generate summaries. In a follow-up work (Haque et al. 2020) the authors use the file context, the other subroutines in the same file, as additional input to the summarization model to improved the generated comments, while in Le Clair et al. (2020) develop a more sophisticated model for handling the AST input by using a graph neural network.

In Chen et al. (2021) the authors observe that comments are written with different intentions, and these can influence the results of comment summarization approaches. To verify their observation, they classify comments according to the following six categories: *"what"*, *"why"*, *"how-to-use"*, *"how-it-is-done"*, *"property"* and *"others"* and evaluate existing comment summarization approaches on these. They notice that the different approaches lead to different results, depending on the comment category, while all approaches have trouble with the *"why"* and *"property"* categories. Finally, they develop a composite approach, that uses a predicted comment category to decide which model to use for comments summarization and obtain results that outperform current state-of-the-art approaches.

The previously mentioned research papers use increasingly sophisticated Deep Learning based approaches to convert the source code for a function body to a short natural language summary and evaluate them automatically using a machine translation metric called the BLEU score. While promising and potentially useful, these kind of approaches only address a small part of a comment, the first sentence, and cannot support the developer to generate information that is not part of the source code. By focusing on a semi-automated approach to support developers with comment writing, we address the full comment and our approach can potentially be used in combination with one that is able to automatically generate the summary of a function.

9 Future Work

For this paper, we have selected the Python and Java languages as these are the most popular languages for which sufficient open source projects exist that are known to use formatting styles for their documentation comments. Both languages have very different styles, so we believe that they provide a fair impression on commenting practices across languages. Nevertheless, we plan on extending our study to further programming languages to investigate this assumption.

Furthermore, we plan on investigating how using different kind of neural language models could improve the accuracy of the generated suggestions. This paper focused mainly on understanding the structure of documentation comments and whether this information can be leveraged to improve the completions. Newer neural language models can also lead to improved results, however they come at additional computational costs for training and inference. However, as shown in a recent work (Mastropaolo et al. 2021) using a newer and more complex model like the Text-To-Text Transfer Transformer (T5) (Raffel et al. 2019) Model for the code summarization task does not necessarily lead to clear improvements. Another potential direction for future work is understanding if other types of contextual information, for example the file context, can improve the predicted completion suggestions.

Finally, while code completion is a widely available and popular feature of many IDEs (Intellij idea - code completion 2021) and many researchers have developed improved approaches over the years (Bruch et al. 2009; Alon et al. 2018; Svyatkovskiy et al. 2020) for completing source code, there are currently no studies investigating the perceived acceptance and usefulness of *source code comment completion*. We plan on conducting such a study in the future. While we expect that accurate completion suggestions will help developers with writing comments faster, we would like to investigate how such a tool influences the content and completeness of the comments.

10 Threats to Validity

Possible threats to the validity of our results are related to the selected dataset and the automated parsing of the docstrings. However, we try to mitigate this threat by applying filtering criteria to the Python dataset we used to ensure a minimal quality of the selected projects. While, for Java we use an available dataset with strict filtering criteria that was made available for the CodeSearch Challenge (Husain et al. 2019a).

While analysing the structure of the documentation comments in Chapter 3 we studied all the methods included in the CSN Dataset (Husain et al. 2019a) for Python and Java. Public API and private non-API methods have different documentation requirements and ideally we should not mix them during our analysis. However, it was not possible to reliably and automatically distinguish between the two types of methods and this issue remains a threat to our study. We tried to mitigate this by selecting the CSN Dataset (Husain et al. 2019a) for the analysis, which only includes library projects, but we were not able to completely eliminate the threat.

Another threat is the use of an automated parser for extracting the different sections. For Python we extended an already existing parser while for Java we developed our own. We wrote automated tests for both parsers and manually evaluated a sample of the extracted sections, nonetheless it is still possible that because of formatting errors in the docstrings or errors in the parsers, we were not able to extract all the sections. However, this means that our findings are more conservative than in reality.

Additionally, the results of the neural language models could be affected by data used for training and evaluation. To mitigate this threat we perform 5-fold cross-validation and report the averaged results over the 5 folds for both the Python and Java datasets. We also ensure the partitioning of the data on a project level, not on a function level, therefore our results are more conservative than could otherwise be obtained.

11 Summary

The current recommendation for function documentation comments for the Python and Java programming language is to follow a specific formatting style, therefore they should have a well defined structure. In this work, we analysed the structure of documentation comments of two large datasets, a Python dataset including 13,590 open source projects and a Java dataset including 4,769 projects open source projects offered as part of the CodeSearchNet challenge (Husain et al. 2019a). For this we extended an existing parser (Docstring parser 2021) for Python and built a new one for Java to extract the following sections: *Short description*, *Long description*, *Parameters*, *Return* and *Raises* sections from the function documentation comments of our datasets. We could observe that developers do include these sections quite often and structure their comments according to a recommended formatting style. Therefore docstrings are not simple natural language text descriptions, but have a well-defined format that should be taken into account when analysing their content or when building tools to help developers with writing comments.

During the second part of the paper we train and evaluate neural language models on training instances generated from the extracted sections. Our results show that there are significant differences in the prediction accuracy depending on the considered sections. When evaluating our base model, the Context LM Model, we obtain a Top-1 prediction accuracy of 0.194 for the *Long Description*, while for the *Parameters*, *Return* and *Raises* sections we obtain a prediction accuracy between 0.264 and 0.323 for the Python dataset. We obtain

similar values for the Java dataset, the prediction accuracy for the *Long description* is 0.192, while for the *Parameters*, *Return* and *Raises* sections we obtain a prediction accuracy between 0.270 and 0.325. Leveraging the section information leads to improved suggestions, as we obtain a relative increase of up to 9.6% for the Python models compared to the base model and for the Java models we obtain a relative increase of up to 7.8% when considering the Top-1 accuracy results. In conclusion, function documentation comments for Python and Java often follow a specific formatting style, thus have a well defined structure, which can be leveraged for approaches that support developers with comment writing.

Acknowledgements This research work was partially funded by the H2020 grant 825328 (FASTEN). We thank the anonymous reviewers for their valuable feedback and comments.

Funding Open access funding provided by University of Zurich.

Data Availability All data generated or analysed during this study are included in this published article (and its supplementary information files) (Replication package 2022).

Declarations

Competing interests The authors declare that they have no competing interests.

References

Aghajani E, Nagy C, Linares-Vásquez M, Moreno L, Bavota G, Lanza M, Shepherd DC (2020) Software documentation: the practitioners' perspective. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, ICSE '20, Association for computing machinery, New York, pp 590–601. https://doi.org/10.1145/3377811.3380405

Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M (2019) Software documentation issues unveiled. In: Proceedings of the 41st international conference on software engineering, ICSE '19, IEEE Press, pp 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

Allamanis M (2018) The adverse effects of code duplication in machine learning models of code. CoRR arXiv:1812.06469

Alon U, Levy O, Yahav E (2018) code2seq: generating sequences from structured representations of code. CoRR arXiv:1808.01400

Bruch M, Monperrus M, Mezini M (2009) Learning from examples to improve code completion systems. In: Proceedings of the 7th joint meeting of the european software engineering conference and the ACM sigsoft symposium on the foundations of software engineering, ESEC/FSE '09, Association for computing machinery, New York, pp 213–222. https://doi.org/10.1145/1595696.1595728

Chen MX, Lee BN, Bansal G, Cao Y, Zhang S, Lu J, Tsay J, Wang Y, Dai AM, Chen Z, Sohn T, Wu Y (2019) Gmail smart compose: Real-time assisted writing. In: Proceedings of the 25th ACM sigkdd international conference on knowledge discovery & data mining, KDD '19, association for computing machinery, New York, pp 2287–2295.. https://doi.org/10.1145/3292500.3330723

Chen Q, Xia X, Hu H, Lo D, Li S (2021) Why my code summarization model does not work: Code comment improvement with category prediction. ACM Trans Softw Eng Methodol, vol 30(2). https://doi.org/10.1145/3434280

Ciurumelea A, Proksch S, Gall HC (2020) Suggesting comment completions for python using neural language models. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER), pp 456–467. https://doi.org/10.1109/SANER48275.2020.9054866

Docstring parser (2021). https://pypi.org/project/docstring-parser/ . Accessed 01 June 2021
Doctest — test interactive python examples (2021). https://docs.python.org/3/library/doctest.html. Accessed 12 Apr 2021
Example numpy style python docstrings (2022). https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html#example-numpy. Accessed 16 Mar 2022
Fluri B, Wursch M, Gall HC (2007) Do code and comments co-evolve? on the relation between source code and comment changes. In: 14th Working conference on reverse engineering (WCRE 2007), pp 70–79. https://doi.org/10.1109/WCRE.2007.21
Github rest api (2022). https://docs.github.com/en/rest. Accessed 16 Mar 2022
Google cloud public datasets (2021). https://cloud.google.com/public-datasets. Accessed 01 June 2021
Google docstring style (2022). https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings. Accessed 15 Mar 2022
Haiduc S, Aponte J, Moreno L, Marcus A (2010) On the use of automated text summarization techniques for summarizing source code. In: 2010 17th Working conference on reverse engineering, pp 35–44. https://doi.org/10.1109/WCRE.2010.13
Haouari D, Sahraoui H, Langlais P (2011) How good is your comment? a study of comments in java programs. In: 2011 International symposium on empirical software engineering and measurement, pp 137–146 https://doi.org/10.1109/ESEM.2011.22
Haque S, Le Clair A, Wu L, McMillan C (2020) Improved automatic summarization of subroutines via attention to file context. CoRR arXiv:2004.04881
Hellendoorn VJ, Devanbu P (2017) Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017, ACM, New York, pp 763–773. https://doi.org/10.1145/3106237.3106290
How to write doc comments for the javadoc tool (2021). https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html. Accessed: 22 Mar 2021
Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: Proceedings of the 26th conference on program comprehension, ICPC '18, association for computing machinery, New York pp 200–210. https://doi.org/doi.org/10.1145/3196321.3196334
Hu X, Li G, Xia X, Lo D, Jin Z (2020) Deep code comment generation with hybrid lexical and syntactical information. Empir Softw Eng, vol 25. https://doi.org/10.1145/3196321.3196334
Husain H, Wu H, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: Evaluating the state of semantic code search. CoRR arXiv:1909.09436
Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) CodeSearchNet, challenge: evaluating the state of semantic code search. arXiv:1909.09436
Intellij idea - code completion (2021). https://www.jetbrains.com/help/idea/auto-completing-code.html. Accessed 01 June 2021
Javalang (2021). https://github.com/c2nes/javalang. Accessed 29 Mar 2021
Iyer S, Konstas I, Cheung A, Zettlemoyer L (2016) Summarizing source code using a neural attention model. In: Proceedings of the 54th annual meeting of the association for computational linguistics, Association for computational linguistics, Berlin (vol 1: long papers), pp 2073–2083. https://doi.org/10.18653/v1/P16-1195. https://www.aclweb.org/anthology/P16-1195
Jurafsky D, Martin JH (2000) Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 1st edn. Prentice Hall PTR, Upper Saddle River
Landauer TK, Foltz PW, Laham D (1998) An introduction to latent semantic analysis. Discourse Process 25(2-3):259–284. https://doi.org/10.1080/01638539809545028
Langdetect (2021). https://pypi.org/project/langdetect/. Accessed: 01 June 2021
Le Clair A, Haque S, Wu L, McMillan C (2020) Improved code summarization via a graph neural network. In: Proceedings of the 28th international conference on program comprehension, ICPC '20, association for computing machinery, New York, NY pp 184–195. https://doi.org/10.1145/3387904.3389268
Le Clair A, Jiang S, McMillan C, A neural model for generating natural language summaries of program subroutines (2019). In: Proceedings of the 41st international conference on software engineering, ICSE '19. IEEE Press pp 795–806. https://doi.org/10.1109/ICSE.2019.00087
Le Clair A, McMillan C (2019) Recommendations for datasets for source code summarization. CoRR arXiv:1904.02660
Lib2to3 (2022). https://github.com/python/cpython/tree/3.10/Lib/lib2to3/. Accessed 12 Mar 2022
Lin CY (2004) ROUGE: a package for automatic evaluation of summaries. In: Text summarization branches out, association for computational linguistics, Barcelona, pp 74–81. https://aclanthology.org/W04-1013
Mastropaolo A, Scalabrino S, Cooper N, Nader-Palacio D, Poshyvanyk D, Oliveto R, Bavota G (2021) Studying the usage of text-to-text transfer transformer to support code-related tasks. CoRR arXiv:2102.02017

Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: 2013 21st International conference on program comprehension (ICPC), pp 23–32. https://doi.org/10.1109/ICPC.2013.6613830

Nltk sentence tokenizer (2021). https://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.punkt. Accessed 01 June 2021

Numpy docstring style (2022). https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard. Accessed 15 Mar 2022

Numpydoc docstring guide (2021). https://numpydoc.readthedocs.io/en/latest/format.html#method-docstrings. Accessed 01 June 2021

Pascarella L, Bacchelli A (2017) Classifying code comments in java open-source software systems. In: Proceedings of the 14th international conference on mining software repositories, MSR '17. IEEE press, Piscataway, pp 227–237. https://doi.org/10.1109/MSR.2017.63

Pep 257 – docstring conventions (2021). https://www.python.org/dev/peps/pep-0257/. Accessed 01 June 2021

Pep 287 – restructuredtext docstring format (2021). https://www.python.org/dev/peps/pep-0287/. Accessed 01 June 2021

Python rouge implementation (2022). https://pypi.org/project/rouge-score/. Accessed 09 Sept 2022

Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2019) Exploring the limits of transfer learning with a unified text-to-text transformer. CoRR arXiv:1910.10683

Replication package (2022). - completing function documentation comments using structural information. https://www.dropbox.com/sh/vi2zl3shvntp8p7/AAC5uwwDV973h2L1Ou7A4hcYa?dl=0. Accessed 16 Dec 2022

Salton G, Wong A, Yang CS (1975) A vector space model for automatic indexing. Commun ACM 18(11):613–620. https://doi.org/10.1145/361219.361220

Sridhara G, Hill E, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM international conference on automated software engineering, ASE '10, ACM, New York, pp 43–52. https://doi.org/10.1145/1858996.1859006

Steidl D, Hummel B, Juergens E (2013) Quality analysis of source code comments. In: 2013 21st International conference on program comprehension (ICPC), pp 83–92. https://doi.org/10.1109/ICPC.2013.6613836

Svyatkovskiy A, Deng SK, Fu S, Sundaresan N (2020) Intellicode compose: Code generation using transformer. CoRR arXiv:2005.08025

The epytext markup language (2022). http://epydoc.sourceforge.net/manual-epytext.html. Accessed 15 Mar 2022

Wan Y, Zhao Z, Yang M, Xu G, Ying H, Wu J, Yu PS (2018) Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018, association for computing machinery, New York, pp 397–407. https://doi.org/10.1145/3238147.3238206

Wen F, Nagy C, Bavota G, Lanza M (2019) A large-scale empirical study on code-comment inconsistencies. In: 2019 IEEE/ACM 27th International conference on program comprehension (ICPC), pp 53–64. https://doi.org/10.1109/ICPC.2019.00019

Writing system software: code comments (2021). http://antirez.com/news/124. Accessed 01 June 2021

Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2018) Measuring program comprehension: a large-scale field study with professionals. IEEE Trans Softw Eng 44(10):951–976. https://doi.org/10.1109/TSE.2017.2734091

Ye D, Xing Z, Li J, Kapre N (2016) Software-specific part-of-speech tagging: An experimental study on stack overflow. In: Proceedings of the 31st annual ACM symposium on applied computing, SAC '16, association for computing machinery, New York, pp 1378–1385. https://doi.org/10.1145/2851613.2851772

Zhou W, Kim S, Murali V, Aye GA (2022) Improving code autocompletion with transfer learning. In: 2022 IEEE/ACM 44th international conference on software engineering: Software engineering in practice (ICSE-SEIP), pp 161–162. https://doi.org/10.1145/3510457.3513061