Multi-step Automated Generation of Parameter Docstrings in Python: An Exploratory Study

Vatsal Venkatkrishna*
Australian National University

Durga Shree Nagabushanam*
Australian National University

Emmanuel Iko-Ojo Simon
Australian National University

Melina Vidoni
Australian National University

ABSTRACT

Documentation debt hinders the effective utilisation of open-source software. Although code summarisation tools have been helpful for developers, most would prefer a detailed account of each parameter in a function rather than a high-level summary. However, generating such a summary is too intricate for a single generative model to produce reliably due to the lack of high-quality training data. Thus, we propose a multi-step approach that combines multiple task-specific models, each adept at producing a specific section of a docstring. The combination of these models ensures the inclusion of each section in the final docstring. We compared the results from our approach with existing generative models using both automatic metrics and a human-centred evaluation with 17 participating developers, which proves the superiority of our approach over existing methods.

CCS CONCEPTS

• **Software and its engineering → Documentation**; • **Computing methodologies → Machine learning**; **Natural language generation**.

KEYWORDS

Docstrings, Pre-trained models, Code summarisation, Scientific software, Documentation debt

ACM Reference Format:
Vatsal Venkatkrishna, Durga Shree Nagabushanam, Emmanuel Iko-Ojo Simon, and Melina Vidoni. 2024. Multi-step Automated Generation of Parameter Docstrings in Python: An Exploratory Study. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3639478.3643110

1 INTRODUCTION

Documentation is a vital aspect of software development but is often neglected by developers because it can be tedious and time-consuming [11]. As a result, documentation worsens over time, leading to increased maintenance costs, reduced software quality, and lower user satisfaction [1]. This is especially prevalent in scientific software, often coded by scientists and mathematicians unfamiliar with coding best practices. In recent years, there have been numerous studies that employ generative neural models for code summarisation – producing a high-level summary of a code snippet [7]. However, a "detailed docstring" with all the technical details of a function's parameters is essential for understanding and reusing code . Generating such a docstring is significantly more challenging since it requires an in-depth understanding of each parameter's role in the function, and is exacerbated by a lack of well-written human-written documentation [8].

In this work, we present an approach to generate documentation catering to the specific needs of scientific software. We based our approach on a taxonomy of parameter-level documentation directives[1] developed by Vidoni and Codabux [12]. Our proposed framework is an amalgam of multiple modules, each specialised in generating a distinct directive. We compared our results to those obtained from using a single transformer-based model through automatic scoring methods and further validated our approach through a survey with 17 participating developers.

2 RELATED WORKS

Deep Learning for Docstring Generation. Code summarisation is the task of generating a short natural language description for a code snippet describing its purpose and functionality. Language models for code have shown promising results on the task [10]. However, few studies investigate their efficacy in generating a detailed docstring. Cui et al. [4] attempted to generate an informative and detailed summary of the code, but their approach does not guarantee the generation of vital directives. Moreover, we were unable to compare our results with theirs due to missing dataset files and minimal documentation. Clement et al. [3] modified the T5 architecture and achieved promising results on generating good quality docstrings, but similarly lack details on reproducibility.

2.1 Methodology

To maintain simplicity in our work, we selected four directives from the taxonomy by Vidoni and Codabux [12] based on the availability of high-quality training data, namely – short description (PD), datatype (PT), default value (PV), and acceptance of "None" (PN).

Data Gathering and Processing: We filtered the set of data science repositories provided by Biswas et al. [2] to contain relevant

*Both authors contributed equally to this research.

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04
https://doi.org/10.1145/3639478.3643110

[1]A documentation directive is a natural-language statement to inform developers of constraints and guidelines related to the correct usage of a section of code; in our case, a function's parameter

repositories with open-source licenses using GitHub's REST API[2], resulting in 873 repositories fitting our criteria. To broaden the scope of our work to all forms of scientific software, we mined repositories of science and statistics using GitHub's Advanced Search . Two authors manually filtered the top 200 results, agreeing to include 109 of them with a Cohen's Kappa score of 0.779, indicating high inter-rater reliability. The final dataset was thus composed of 982 repositories. We extracted $111k$ function-docstring pairs from these repositories using `function-parser`[3]. Further, we discarded all functions with minimal or partial documentation and standardised the docstring format across all remaining instances, resulting in a dataset of $41k$ functions, hereby denoted as the *Formatted* dataset.

Model Preparation: Our approach aims to simplify the task of docstring generation by generating parameter-wise docstrings using an amalgam of multiple models best suited to each target directive.

PDs: We split the *Formatted* dataset into N data points for each existing function, where N is the number of parameters in the corresponding function. We truncated each sample to the first sentence to prevent extra details from being generated. To avoid different outputs for the same code snippet, we prepended the phrase "parameter N: " to the code string in each entry to indicate the parameter being documented in the docstring. This dataset is hereby referred to as the *Exploded* dataset. We fine-tuned CodeBERT [5], CodeT5 [13] and UniXcoder [6] on the resulting dataset for 10 epochs.

PTs: To our knowledge, Type4Py [9] is the only model that can predict datatypes in the absence of available documentation. We used its API to obtain the datatypes of all variables in a given function and mapped them to the corresponding parameters.

PVs: Identifying default values is a relatively simple task in Python since defaults are mostly declared in the function header itself. Thus, we implemented a rule-based algorithm to parse the source code and infer defaults using Python's inbuilt `ast` module[4].

PNs: We modelled PNs as a binary classification problem, modifying the *Exploded* dataset to include a label. If the gold description of the parameter contained the "None" token, a label of 1 was assigned, 0 otherwise. To counter a large number of negative samples, we sampled an equal amount of both samples. We fine-tuned UniXcoder and CodeBERT on the resulting dataset for 10 epochs.

2.2 Results and Discussion

Firstly, we evaluated the performance of each directive-generating module. For PDs, we observed that CodeBERT significantly outperforms the remaining models with a BLEU score of 0.310, followed by CodeT5 at 0.278. This is surprising since CodeT5 has a pre-trained decoder, while CodeBERT's decoder is fine-tuned from scratch, suggesting CodeT5's pre-training does not help in generating parameter-wise descriptions. For PNs, UniXcoder achieves an F1-score of 0.704, outperforming CodeBERT at 0.669 as anticipated from its superior performance on other code-based classification tasks. For PVs and PTs, we did not have any labelled ground truths, prompting us to manually evaluate a representative sample with 95% confidence and 5% error. Two authors found PDs to have a high accuracy of 93.7% with a Cohen's Kappa of 0.903, falling short in cases such as ***def sample3(x, y=None): if y is None: y=5***, where the default value is 5 and not "None". However, the accuracy on PTs was much lower at 20%, indicating the need for better annotation and methodologies for type prediction in the absence of documentation.

Secondly, we compared our generated PDs to those generated by a single generative model – obtained by fine-tuning the same models on the *Formatted* dataset. CodeBERT saw a 46% improvement in BLEU in our multi-step approach compared to the generative-only approach. Moreover, among the best-performing models, CodeBERT in our setting achieved a 0.310 BLEU, outperforming the best generative-only model, CodeT5 at 0.281. However, to measure the impact of the other directives (PV, PT, PN), we conducted a human-centred evaluation. We designed a survey that received responses from 17 graduate students in software engineering who majorly use Python in their projects. They rated 10 randomly sampled docstrings generated by the best-performing models in both approaches on a six-point Likert scale on measures of understandability (U), grammaticity (G), completeness (C) and technical nature (T). We found developers to prefer our approach in terms of U, C and T over the generative-only approach, and no significant differences in G. We also found that CodeT5 is better ranked by developers despite CodeBERT having a higher semantic overlap with current documentation standards. This is an indication for more human-in-the-loop evaluation metrics to be used for documentation-related tasks to ensure that AI models learn to cater to the needs of developers.

In conclusion, we present a multi-step approach to generating detailed parameter docstrings for Python that outperforms neural generative models. It can be extended to include a wide variety of directives and can help reduce future documentation debt.

[2]https://docs.github.com/en/rest?apiVersion=2022-11-28
[3]https://nathancooper.io/function_parser/
[4]https://docs.python.org/3/library/ast.html

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *41st International Conference on Software Engineering*.

[2] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa Meets Python: A Boa Dataset of Data Science Software in Python Language. In *16th International Conference on Mining Software Repositories*.

[3] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. *arXiv:2010.03150* (2020).

[4] Haotian Cui, Chenglong Wang, Junjie Huang, Jeevana Priya Inala, Todd Mytkowicz, Bo Wang, Jianfeng Gao, and Nan Duan. 2022. CodeExp: Explanatory Code Document Generation. *arXiv:2211.15395* (2022).

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. (2020).

[6] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. (2022).

[7] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *17th Working conference on reverse engineering*.

[8] Timothy C Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE software* (2003).

[9] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*.

[10] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. (2023).

[11] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. 2015. To document or not to document? An exploratory study on developers' motivation to document code. In *CAiSE: Advanced Information Systems Engineering Workshops*.

[12] Melina Vidoni and Zadia Codabux. 2023. Towards a Taxonomy of Roxygen Documentation in R Packages. *Empirical Software Engineering* (2023).

[13] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR* abs/2109.00859 (2021).