DocuMint: Docstring Generation for Python using Small Language Models

Bibek Poudel*, Adam Cook*, Sekou Traore*, Shelah Ameli**
Min H. Kao Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN, USA
{bpoudel3,acook46,staore1,oameli}@vols.utk.edu

Abstract

Effective communication, specifically through documentation, is the beating heart of collaboration among contributors in software development. Recent advancements in language models (LMs) have enabled the introduction of a new type of actor in that ecosystem: LM-powered assistants capable of code generation, optimization, and maintenance. Our study investigates the efficacy of small language models (SLMs) for generating high-quality docstrings by assessing accuracy, conciseness, and clarity, benchmarking performance quantitatively through mathematical formulas and qualitatively through human evaluation using Likert scale. Further, we introduce DocuMint, as a large-scale supervised fine-tuning dataset with 100, 000 samples. In quantitative experiments, Llama 3 8B achieved the best performance across all metrics, with conciseness and clarity scores of 0.605 and 64.88, respectively. However, under human evaluation, CodeGemma 7B achieved the highest overall score with an average of 8.3 out of 10 across all metrics. Fine-tuning the CodeGemma 2B model using the DocuMint dataset led to significant improvements in performance across all metrics, with gains of up to 22.5% in conciseness. The fine-tuned model and the dataset can be found in HuggingFace[1] and the code can be found in the repository[2].

CCS Concepts: • **Software and its engineering → Automatic programming**; **Software prototyping**.

Keywords: Docstring generation, LLMs for documentation, Fine-tune LLMs for docstring generation.

1 Introduction

Code is at the frontier of Artificial Intelligence (AI). The emergence of Large Language Models for Code (Code LLMs), including foundation models such as CodeLlama [36], CodeGemma [18], and StarCoder [28], along with products like GitHub Copilot [2] and GhostWriter [5], has paved the way for even more advanced software development tools. The latest saga in this odyssey is the rise of CodeLLM-enabled AI agents such as Devin [3], Devika [1], and OpenDevin [4], which can automate entire software development pipelines, ushering us into an era where software is co-developed by humans and AI. Already in 2022, GitHub reported that on average, 46% of all code written across all programming languages was assisted by GitHub Copilot [17]. While the formal syntax, limited vocabulary, and deterministic outcomes in code provide an ideal (structured) environment for CodeLLMs to thrive [43], at the same time, their usefulness is challenged by the need to understand context, interpret natural language instructions, and convey the intent for generated code. As the CodeLLM landscape evolves and software development increasingly involves collaboration between human and AI, it becomes increasingly important for both parties to convey their objective and comprehend the code generated by their counterparts.

```python
def example_function(param1, param2):
    """
    This is an example of a docstring
    for a Python function.

    Docstrings are enclosed in triple
    quotes and appear immediately
    after the function definition.
    """

    # Function implementation
```

Figure 1. An example of docstring for a python function. Docstrings serve as documentation and help developers understand how to use the respective code.

Given the impressive capabilities of LLMs and CodeLLMs, there is growing interest in smaller, more efficient models called Small Language Models (SLMs) [34], such as Phi [6], Gemma [19] and smaller variants of Llama [7, 33]. SLMs are significantly more cost-effective in terms of training and inference (latency, memory, throughput, and energy consumption), and they are small enough for considerations on running locally; about 7B parameters for consumer-level GPUs and about 2B parameters for comsumer CPUs. **As language models continue to shape the landscape of software development, high-quality documentation becomes increasingly crucial for facilitating effective communication and collaboration between humans and AI**. Given that developers already spend 58% of their time on comprehension of unfamiliar code [41], writing clear and consistent documentation helps helps reduce this time while enabling both human developers and AI to convey the

*Authors contributed equally to this research.
[1] https://huggingface.co/documint
[2] https://github.com/Docu-Mint/DocuMint

intent behind the code and understand each other's contributions better. This not only enhances the developer experience but also improves overall software quality.

Although CodeLLMs are capable of generating docstrings (as docstrings are a part of their pre-training and fine-tuning data), to date, limited research has focused on evaluating the quality of the generated docstrings. Existing benchmarks, such as HumanEval [14] and MBPP [9], assess code generation performance, leaving a gap in measuring the quality of the accompanying documentation. Furthermore, traditional natural language generation metrics, such as Bilingual Evaluation Understudy (BLEU), fall short in evaluating docstring quality due to their emphasis on n-gram overlap between generated and reference sentences, disregarding the semantic meaning and context. In this work, we address these limitations by introducing DocuMint, a large-scale dataset and methodology for evaluating the quality of docstrings generated by SLMs. We make three key contributions:

- First, we breakdown the "quality" assessment of docstrings into three categories: accuracy, conciseness, and clarity with well defined metric for each category.
- Second, we benchmark the performance of leading code generation SLMs in generating docstrings using both the mathematical metrics and human evaluation.
- Third, we introduce DocuMint, a supervised fine-tuning dataset consisting of 100, 000 samples.

To the best of our knowledge, DocuMint is the first comprehensive work designed to evaluate and improve the quality of docstrings generated by SLMs, providing a large-scale dataset, benchmarks, a fine-tuned model, and an evaluation methodology.

2 Related Work

There exist two mainstream code-related evaluations for LLMs: code generation and code comprehension.

Code Generation: The standard code-related evaluation benchmarks for language models, such as HumanEval [14], MBPP [9], and SWE-Bench [23], assess their abilities to solve problems through generated code. HumanEval presents carefully crafted programming problems and evaluates whether the generated code solutions pass on hidden test cases. MBPP tests the ability of LLMs to translate instructions into functional code. More recently, SWE-Bench tests their abilities to tackle real-world software issues sourced from GitHub, directly assessing a model's proficiency in understanding and resolving software problems.

Code Comprehension: Most of the prior research in this field has focused on using LLMs to comprehend human-written or AI-generated code, including code summarization [12, 39] and explanations [37]. These tasks leverage the abilities of LLMs to reason in a Chain-of-thought [40] or in a step-by-step [25] manner. Code summarization focuses on generating succinct, human-readable summaries of code snippets, whereas step-by-step explanations are more prominent in educational settings [11].

To date, limited research has focused on the evaluation of CodeLLMs in docstring generation, and prominent datasets for this task are scarce. Early works on code documentation generation, such as [10], introduced datasets of parallel code and natural language descriptions of 150, 370 Python function declarations, bodies, and docstrings scraped from open-source repositories. However, these efforts were limited by the size and diversity of the available data, and they were conducted before the emergence of language models, which questions their relevance. Similarly, line-by-line comment generation [22] and contextual function/method call and usage information [32] have been explored, but these works also predate the rise of LLMs. More recently, RepoAgent [29] has demonstrated the potential of LLMs for repository-level code documentation generation, generating and proactively maintaining high-quality documentation for entire projects.

While these existing works provide valuable insights, they differ from our approach, which is to evaluate the abilities of CodeLLMs themselves to generate documentation alongside the code they produce. We evaluate the efficacy of language models in documentation generation as a software development tool, i.e., **given a code function, is the CodeLLM able to describe its core functionality, input/output parameters, and intended usage in an accurate, concise, and clear manner?**

3 Preliminaries

In this section, we introduce the various training phases of a language model and the tasks that it is trained to perform.

3.1 Training

Creating a helpful Language Model (LM) involves three essential steps: pre-training, fine-tuning, and alignment.

Pre-training: Pre-training involves training a LM on vast amounts of unlabeled text data using self-supervised learning techniques such as masked language modeling or next token prediction. This process allows the model to capture the intrinsic patterns and statistical regularities present in natural language. Pre-training is computationally intensive and requires clusters of GPUs to process the internet-scale data efficiently, resulting in a "base model" that serves as a foundation for further fine-tuning.

Fine-tuning: Pre-training alone does not enable the model to understand or follow specific instructions. The utility of LMs is significantly improved by fine-tuning and depending on the nature of the task, fine tuning could be Supervised Fine Tuning (SFT) or Instruction Tuning (IT). SFT adapts a pre-trained base model to perform specific downstream tasks by training it on labeled data relevant to the task. Whereas IT is employed to turn the base model into a useful assistant capable of understanding and responding to user instructions expressed in natural language.

Alignment: Also known as human preference fine-tuning, this step aligns the behavior of an LM with human preferences, enhancing its friendliness, helpfulness, and safety. This process involves collecting human feedback on the model's outputs and using this feedback as a reward signal to guide the model's behavior. By optimizing the model to generate outputs that align with human preferences, this fine-tuning helps to mitigate undesirable behaviors and ensures more socially appropriate and beneficial responses.

3.2 Tasks

LMs can be useful for various tasks, such as text/code infilling and completion.

Infilling: is a specialized task designed for code-generation models, where the objective is to generate code snippets or comments that best fit within a given prefix and suffix. This task is particularly relevant for code assistants, which are trained to provide code suggestions based on the surrounding context at the current cursor position. Infilling models learn to capture the syntactic and semantic patterns present in code, allowing them to generate coherent and contextually appropriate code fragments.

Completion: refers to the fundamental capability of LMs to generate fluent and coherent text by predicting the next token in a sequence based on the preceding context. Completion allows LLMs to generate human-like text that follows the style, tone, and subject matter of the given prompt. This is the most common task of a LM.

4 Methodology

In this section, we provide an overview of the selected state-of-the-art models, the data used for inference and benchmarking them, the data extraction process for fine-tuning, and the applied fine-tuning technique.

4.1 Selected Models

Models were selected from the EvalPlus Leaderboard [27] at full precision, with their rank on the leaderboard as of April 25,2024 shown in Table1.

Model	Rank
DeepSeek Coder 6.7B Instruct	14
Meta Llama3 8B Instruct	38
CodeGemma 7B Instruct	43
StarCoder2 7B	66
CodeGemma 2B	83

Table 1. Selected models and their rank.

Since the selected models are CodeLLMs, use instruct variants whenever available.

4.2 Benchmarking Data

For benchmarking the selected models, we extract seven python functions from three widely established NLP datasets [13, 24, 26, 35, 36, 40, 44] and perform model inference.

The Mostly Basic Python Problems **MBPP** dataset [9] consists of around 1,000 crowd-sourced Python programming problems, designed to be solvable by entry level programmers, covering programming fundamentals, standard library functionality, and so on. Each problem consists of a task description, code solution and 3 automated test cases. The dataset is split into a train and test set, we select functions from the test set.

The Hand-Written Evaluation Set **HumanEval** dataset [14] consists of 164 hand written problems with a function signature, a docstring describing the task, reference solution, and tests. These problems are our medium level functions, they contain functions and problems that are notably more complex than the MBPP dataset but less algorithmic intensive when compared to our selected APPS functions.

The Automated Programming Progress Standard **APPS** dataset [20] comprises 10,000 coding examples, evenly distributed between train and test sets. These examples were sourced from various open-access coding websites like Codeforces and Kattis. Each problem within the dataset includes a question description, solutions, input/output details, difficulty level, and source URL, with problems categorized into Introductory (simple and easy), Interview (technical interview questions), and Competition Level (advanced programming competitions).

We focused on functions categorized under the 'interview level' section of the test set. From each problem, we extracted solutions contained within functions defined by the 'def' keyword, excluding class objects from our inference dataset for analysis.

4.3 Fine Tuning Data

As the human annotation process is expensive and relatively slow, we leverage existing code from the Free and open-source software (FOSS) ecosystem. We utilize World of Code (WoC) [30, 31], an efficient and flexible project analysis framework that provides an abstracted interface to the intricacies of this ecosystem, enabling a reliable and straightforward approach to research. Our goal is to ensure quantity, quality, and diversity of data for fine-tuning.

We query the repositories in WoC and filter them based on metrics such as number of contributors (> 50), commits ($> 5k$), stars ($> 35k$), and forks ($> 10k$). This ensures that we focus on well-established and actively maintained projects. To extract functions from the filtered repositories, we employ a parser that utilizes the GitPython and abstract syntax tree modules to navigate through repository files and identify function definitions. The parser extracts 100,000 Python functions along with their respective docstrings. The data

Category	Parameter	Value
CodeGemma Config	vocabulary size	256000
	max pos. embeddings	8192
	num_hidden_layers	18
	num_heads	8
Fine Tuning	max_seq_len	128
	LoRA rank	64
	LoRA alpha	128
	LoRA dropout	0.1
	batch size	8
	grad accumulation steps	16
	epochs	4
	optimizer	8 bit adam
	Learning rate (initial)	2e−4

Table 2. Detailed configuration of the CodeGemma model and hyper-parameters used in fine-tuning.

was formatted into a JSON file with the Alpaca instruction format [38] shown below.

```
{
"instruction": "<function definition>",
"response": "<ideal docstring>"
}
```

4.4 Fine Tuning

The full model fine-tuning for language models, even for modest sizes, is memory and compute-intensive; hence, we resort to parameter-efficient fine-tuning, updating only a small subset of model parameters, among which Low-Rank Adaptation (LoRA)[21] is a popular technique. We perform supervised instruction fine-tuning on the Gemma-2B base model, which has been trained to predict the next token on internet text without any instructions, using LoRA. This fine-tuning process involves using examples that demonstrate how the model should respond to specific instructions. The total fine-tuning time was 48 GPU hours, with 78, 446, 592 LoRA parameters at full precision (32 bit) and 185, 040, 896 total training tokens. The full set of parameters used in fine-tuning are provided in Table2.

5 Experiments

In this section, we define our experiments, describe the metrics used to assess the generated docstrings, provide summary statistics of the human participants, and detail the experiment setup.

Our research objectives are contained in the three experiments described below.

Experiment 1 quantitatively evaluates the docstrings generated by SLMs using the three metrics: Accuracy, Conciseness, and Clarity, as described in Section 5.1.

Experiment 2 qualitatively evaluates a subset of the docstrings generated by SLMs using a human questionnaire

System Prompt

```
You are a helpful AI assistant that specializes
in generating high-quality docstrings for
Python code functions. Your task is to create
docstrings that are:

Accurate: Cover functionality, parameters,
return values, and exceptions.

Concise: Brief and to the point, focusing
on essential information.

Clear: Use simple language and avoid ambiguity.

Generate docstring in this format:
"""<generated docstring>""".
```

Figure 2. System prompt that is attached before the function definition when querying the model for inference.

where participants with prior Python programming experience score the Accuracy, Conciseness, and Clarity of generated docstrings on a Likert scale.

Experiment 3 evaluates the docstrings generated by a SLM that has been fine-tuned on data extracted from World of Code and quantitatively compares (similar to Experiment 1) the performance against the original model.

5.1 Experiment Metrics

Classic NLP metrics such as Bilingual Evaluation Understudy (BLEU) are often used to evaluate machine translation systems by comparing the machine-generated translations to human-generated reference translations. However, they are not reliable in the context of code [16]. Two dramatically different code snippets can be semantically equivalent, and since the same vocabulary is often used with docstrings attached to the code, we refrain from using BLEU. We consider the primary concern when it comes to LLM-generated docstring is accuracy (correctness); to further get a comprehensive measurement, we consider conciseness, and clarity.

Accuracy: measures the correctness in the description of the function logic and coverage of the generated docstring on associated code elements such as input and output variables. We measure accuracy using BERTScore [42] given by

$$\text{Accuracy} = \frac{v_g \cdot v_e}{\sqrt{v_g^2} \cdot \sqrt{v_e^2}},$$

where v_g and v_e are the Bidirectional Encoder Representations from Transformers [15] encodings of the SLM generated docstring and an "expert" docstring generate by the Claude-3 Opus [8] LLM, respectively. A higher accuracy score indicates a greater similarity between the generated docstring and expert docstring and vice versa.

Model	Accuracy	Conciseness	Clarity
CodeGemma 7B	0.609	0.569	76.49
DeepSeek Coder 6.7B	**0.679**	0.734	**64.44**
Llama3 8B	0.668	**0.605**	**64.88**
StarCoder2 7B	0.626	0.510	69.74

Table 3. Experiment 1 results i.e., comparative evaluation of various CodeLLMs in generating docstring as measured by Accuracy, Conciseness, and Clarity. Instruct variants used for CodeGemma, DeepSeek, and Llama3. Overall, Llama 3 performs the best across all three metrics. DeepSeek achieves the best Accuracy score, CodeGemma achieves the best Conciseness score, and both DeepSeek and Llama 3 share the best Clarity score.

Conciseness: measures the ability of the generated docstring to convey information succinctly without unnecessary verbosity. We measure conciseness using a compression ratio defined as:

$$\text{Conciseness} = \frac{\text{Text}_C}{\text{Text}_O},$$

where Text_C and Text_O are the sizes of the compressed text and the original text respectively. The compression ratio is bounded between 0 and 1 with a lower ratio indicating more conciseness. An ideal score is between 0.5 and 0.6 i.e., if the score is too low, it indicates that the generated docstrings are extremely short.

Clarity: measures the readability of the generated docstring i.e., uses simple language and is unambiguous. We measure clarity using the Flesch-Kincaid reading score:

$$\text{Clarity} = 206.835 - (1.015 * \frac{w}{l}) - (84.6 * \frac{s}{w})),$$

where w is the number of words generated, l is the number of sentences or lines, and s is the number of syllables. A score between $70 - 100$ indicates that the text is very easy to read, at $50 - 70$ text slowly becomes more difficult to read (high school level), and $0 - 50$ is regarded as difficult to read and best understood by university graduates. Ideally, for docstrings a score of $50 - 70$ is desired.

5.2 Summary statistics of the Human Population

We selected participants to perform a 'qualitative' measure of the docstrings generated by each of the four Small Language Models (SLMs). Each participant evaluated five questions from the inference dataset, with three persons per SLM. A sample of the questionnaire is included in appendix A.1 for reference.

The selected population of this study comprised of human programmers, with a median age of 25.5 years. The cumulative years of Python Programming experience is 47 years, averaging 4 years per participant. We had a diverse group of 8 males and 4 females. Figure 3 highlights the habits of the population with reading and engaging with docstrings.

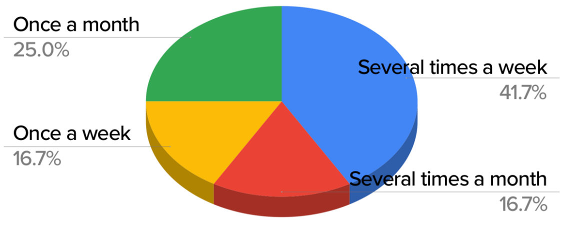


Figure 3. Survey participants' frequency in reading docstrings. About 42% read docstrings several times a week, ~ 17% read docstrings once a week and several times a month each, while a quarter of them reads docstring once a month.

5.3 Experiment Setup

The data and model exploration phase was conducted using Lightning AI with A-10G GPUs. For inference by zero-shot prompting in experiment 1, we employed an Intel 14900K CPU, an Nvidia RTX-A5000 GPU, and 32GB RAM. Fine-tuning experiments were performed using an Intel 12900K CPU, an Nvidia RTX-3090 GPU, and 64GB RAM. All models used in this study were obtained through HuggingFace.

6 Experiment Results

In this section, we present the results of our quantitative benchmarks, qualitative human questionnaires, and fine-tuning experiments on the selected SLMs.

6.1 Experiment 1 Results

Table 3 shows the quantitative measurements of accuracy, conciseness, and clarity for the docstrings generated by SLMs, all of which have roughly 7 billion parameters. Accuracy scores were led by DeepSeek at 0.679, with Llama 3 in second place at 0.668. This indicates that the docstrings generated by DeepSeek and Llama 3 were the most similar to the control model (Claude 3 Opus, whose parameter count is in the hundreds of billions). On the other hand, CodeGemma scored the lowest accuracy at 0.609, indicating the most dissimilarity with the control model.

For conciseness, CodeGemma had the best score at 0.569, while StarCoder2, with a score of 0.510, generated extremely

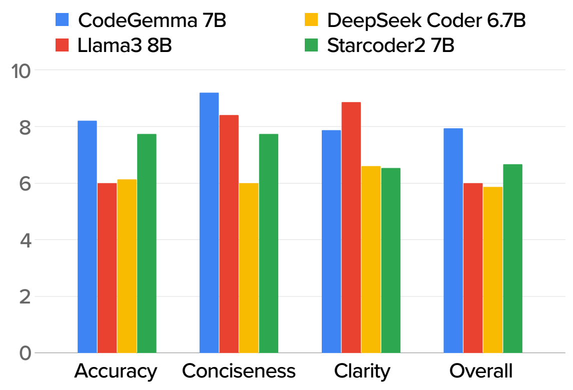


Figure 4. The average weighted scores from the human questionnaire. Instruct variants used for CodeGemma, DeepSeek, and Llama3. CodeGemma performs the best overall surpassing all other models in terms of accuracy and concisenes. Llama3 achieves the best clarity score. StarCoder2 was the second-best-performing model from a human perspective.

short docstrings that may not convey enough information. Surprisingly, DeepSeek Coder exceeded the ideal threshold of $0.5 - 0.6$ with a score of 0.734, implying that the generated docstrings were verbose. Further, DeepSeek, Llama3, and StarCoder2 have clarity scores within the desired range of $50 - 70$, with StarCoder2 at the upper threshold at 69.74. However, CodeGemma exceeds the ideal threshold with a score of 76.49. This means that while CodeGemma is deemed "easy to read," it uses much simpler language and syntax and may not be considered of professional quality.

6.2 Experiment 2 Results

Figure 4 shows the average accuracy, conciseness, clarity, and overall scores from the human questionnaires. For accuracy, CodeGemma achieves the overall best performance with 8.2 out of 10. Surprisingly, Starcoder2 performed well with an accuracy of 7.7 from a human perspective despite having the worst results in the math benchmarks (Section 3). Llama3 and DeepSeek Coder have the lowest accuracy scores, which varies from the results achieved with the quantitative benchmarks. This suggests that humans consider CodeGemma and DeepSeek Coder as more reliable in generating correct docstrings compared to the other two models.

For conciseness, CodeGemma takes the lead with a high score of 9.2 out of 10, indicating its ability to generate more succinct and compact responses, this is consistent with the quantitative metric where CodeGemma also had the best conciseness score. Llama3 and Starcoder2 show similar levels of conciseness, while DeepSeek Coder generates the least concise score (5.8) among the four SLMs. Whereas for Clarity, humans rank Llama3 as being the clearest with a clarity score of ~ 9, with CodeGemma coming closely behind with 7.8 clarity score demonstrating comparable performance. DeepSeek Coder and Starcoder2 exhibit lower clarity scores,

Model	Accuracy	Conciseness	Clarity
CodeGemma 2B	0.516	0.425	91.69
CodeGemma 2B Fine-tuned	**0.582**	**0.521**	**58.75**

Table 4. Experiment 3 results i.e., comparative evaluations between CodeGemma 2B base model and it's fine-tuned version. The fine-tuned model displays improvements in all metrics compared to the original model.

suggesting that humans perceived their generated responses to be less understandable compared to the top two models.

Overall, CodeGemma and Starcoder2 stand out as the best-performing models, with CodeGemma having an overall score of 7.9 on the Likert scale. These two models consistently score higher across the three metrics compared to Llama3 and DeepSeek Coder. The results indicate that CodeGemma and Starcoder2 are more effective in generating accurate, concise, and clear responses.

6.3 Experiment 3 Results

Table 4 presents the comparative evaluations between the base CodeGemma 2B model and its fine-tuned version. The fine-tuned model shows notable improvements across all metrics: accuracy increased by 12.7%, conciseness by 22.5%, and clarity by three reading levels. Despite the fine-tuned model being a 2B parameter model, it achieves competitive performance compared to the larger Llama3 8B model shown in Table 3. The fine-tuned CodeGemma 2B model has 14% lower accuracy than Llama3 8B, but its conciseness falls within a desirable range, and its clarity is only slightly below the threshold. These results validate the effectiveness of our fine-tuning process using the data obtained from World of Code. The fine-tuning loss curve is shown in Figure 5.

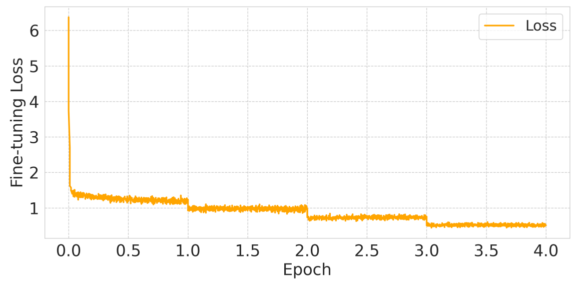


Figure 5. The loss curve during fine-tuning of the CodeGemma-2B model. The loss decreases sharply during the first epoch after which the curve saturates with marginal decrements per epoch.

7 Conclusion and Future Work

In this project, we introduced DocuMint, a dataset and methodology for evaluating the ability of Small Language Models

(SLMs) to generate useful Python docstrings. Our experiments indicate that SLMs are capable of generating docstrings that are accurate, concise, and clear to a reasonable degree. Some of the results are surprising. First, Llama3 8B, a general-purpose SLM, performed the best overall in calculated metrics, despite the fact that its training data only contains a small fraction of code samples compared to code-specific models (although the exact number is not available, it was reported that Llama3 had four times more code samples than the dataset used to train Llama 2, which likely improved its abilities). Second, CodeGemma 7B was perceived as the most accurate by humans, despite having lower accuracy scores in calculated metrics. Further, to the performance of SLMs in generating docstrings, we fine-tuned CodeGemma 2B model on data extracted from open source repositories using World of Code which leads to improvements across all metrics (upto 35% improvement in docstring clarity).

In the future, there are many interesting directions to extend this work. Expanding DocuMint with more diverse code samples and human-annotated reference docstrings would increase the quality of fine-tuning dataset. Developing better automated metrics to measure docstring quality that better align with human judgement could be another crucial area for future research. To facilitate future efforts, we have release the DocuMint dataset, the fine-tuned CodeGemma 2B model, and the code we used in this work in public repositories on GitHub and HuggingFace.

8 Reflections and Lessons Learned

Insights into the FOSS Ecosystem: The process of analyzing the Free and Open Source Software (FOSS) ecosystem through World of Code provided some crucial insights. Initially, we encountered a significant reduction in usable data—from 148,000 extracted functions to only 66,000 actually usable-due to syntactic errors that affected the correct parsing by the AST module. This outlines the challenge that the most popular projects are not always the best written.

Subjectivity of Docstring Quality: Docstring quality in projects varies widely and tends to be subjective, based on what developers deem important to document. For our fine-tuning data, the goal was to find an ideal balance in docstring clarity, conciseness, and accuracy. However, there is no single strategy that helps us achieve this across repositories.

Challenges Encountered During Sampling: The sampling process itself presented numerous challenges:

- Syntactic errors in repositories hindered AST parsing.
- Difficulty in identifying Python-specific files in multi-language repositories.
- Computational and time expenses associated with parsing and analyzing large codebases.
- Discrepancies in repository sizes posed challenges in maintaining uniformity in sampling.
- Subjective differences in the overall quality of the docstrings.
- Difficulty in avoiding repetitions in such a large dataset (many implementations of the same algorithms might have the same docstrings).
- Finding a sufficiently diverse set of data.

Further, the human questionnaire survey that we conducted for Experiment 2 also provided some qualitative insights behind the numerical scores. While the participants had praises for the SLMs output, they also highlighted issues that reduced the quality of the generated docstrings. The most common feedback for all models are listed below:

- Inconsistencies in spelling, description, and examples.
- Incorrect references to arguments and return type.
- Raises errors and exceptions not present in code.
- Weak explanations for nested loops.
- Too long Docstrings.

9 Contributions

All authors contributed equally to the work.

Bibek Poudel: Researched model precision and parameter efficient fine tuning. Introduced HuggingFace and Lightning AI. Secured required compute. Led the fine-tuning effort and technical report writing. Assisted in model inference and evaluation metrics setup.

Adam Cook: Researched available Small Language Models to implement for generating docstrings; developed a Python file that created an output text file containing docstrings generated from an input SLM and JSON data file; calculated the metrics depicted in Table 2.

Sekou Traore: Educated the group on tokenization and data related tasks. Developed a data strategy for fine tuning, familiarized with World of Code and performed data operations such as scraping, filtering, and json formatting as required for fine-tuning.

Shelah Ameli: Initial research on available compute within the ISAAC cluster. Studied standard datasets such as MBPP, HumanEval, and APPS; scraped and filtered a curated dataset for inference. Provided a JSON structure file for setting up inference experiment. Facilitated and compiled the human benchmark experiments.

Acknowledgments

The authors would like to thank Dr. Audris Mockus for his guidance on the project and help with World of Code. The authors would also like to thank the Fluidic City Lab for providing the compute resources.

References

[1] 2024. devika. https://github.com/stitionai/devika.

[2] 2024. GitHub Copilot. https://github.com/features/copilot.

[3] 2024. Introducing Devin. https://www.cognition-labs.com/introducing-devin.

[4] 2024. OpenDevin. https://github.com/OpenDevin/OpenDevin.

[5] 2024. Replit AI. https://replit.com/ai.

[6] Marah Abdin, Sam Ade Jacobs, and et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. *arXiv preprint arXiv:2404.14219* (2024). https://arxiv.org/abs/2404.14219 Accessed 2024 April.

[7] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[8] Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family Accessed: 2024-05-13.

[9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[10] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[11] Peter Brusilovsky, Arun-Balajiee Lekshmi-Narayanan, Priti Oli, Jeevan Chapagain, Mohammad Hassany, Rabin Banjade, and Vasile Rus. 2023. Explaining code examples in introductory programming courses: Llm vs humans. *arXiv preprint arXiv:2403.05538* (2023).

[12] Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song Dong. 2024. On-the-Fly Adapting Code Summarization on Trainable Cost-Effective Language Models. *Advances in Neural Information Processing Systems* 36 (2024).

[13] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* (2023).

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[16] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* 203 (2023), 111741.

[17] GitHub. 2023. GitHub Copilot for Business is now available. https://github.blog/2023-02-14-github-copilot-for-business-is-now-available/.

[18] Google. 2024. CodeGemma: Open Code Models Based on Gemma. https://storage.googleapis.com/deepmind-media/gemma/codegemma_report.pdf.

[19] Google. 2024. Gemma: Open Models for Developers. https://blog.google/technology/developers/gemma-open-models/.

[20] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[22] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[24] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[25] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[27] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct. *Rigorous evaluation of large language models for code generation. CoRR, abs/2305.01210* (2023).

[28] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

[29] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. *arXiv preprint arXiv:2402.16667* (2024).

[30] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 143–154.

[31] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering* 26 (2021), 1–42.

[32] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[33] Meta. 2023. LLaMA. https://llama.meta.com.

[34] Microsoft Research. 2024. Phi-2: The Surprising Power of Small Language Models. https://www.microsoft.com/en-us/research/blog/\phi-2-the-surprising-power-of-\small-language-models/.

[35] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[36] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[37] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.

[38] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3, 6 (2023), 7.

[39] Yuki Wang, Gonzalo Gonzalez-Pumariega, Yash Sharma, and Sanjiban Choudhury. 2024. Demo2code: From summarizing demonstrations to synthesizing code via extended chain-of-thought. *Advances in Neural Information Processing Systems* 36 (2024).

[40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[41] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

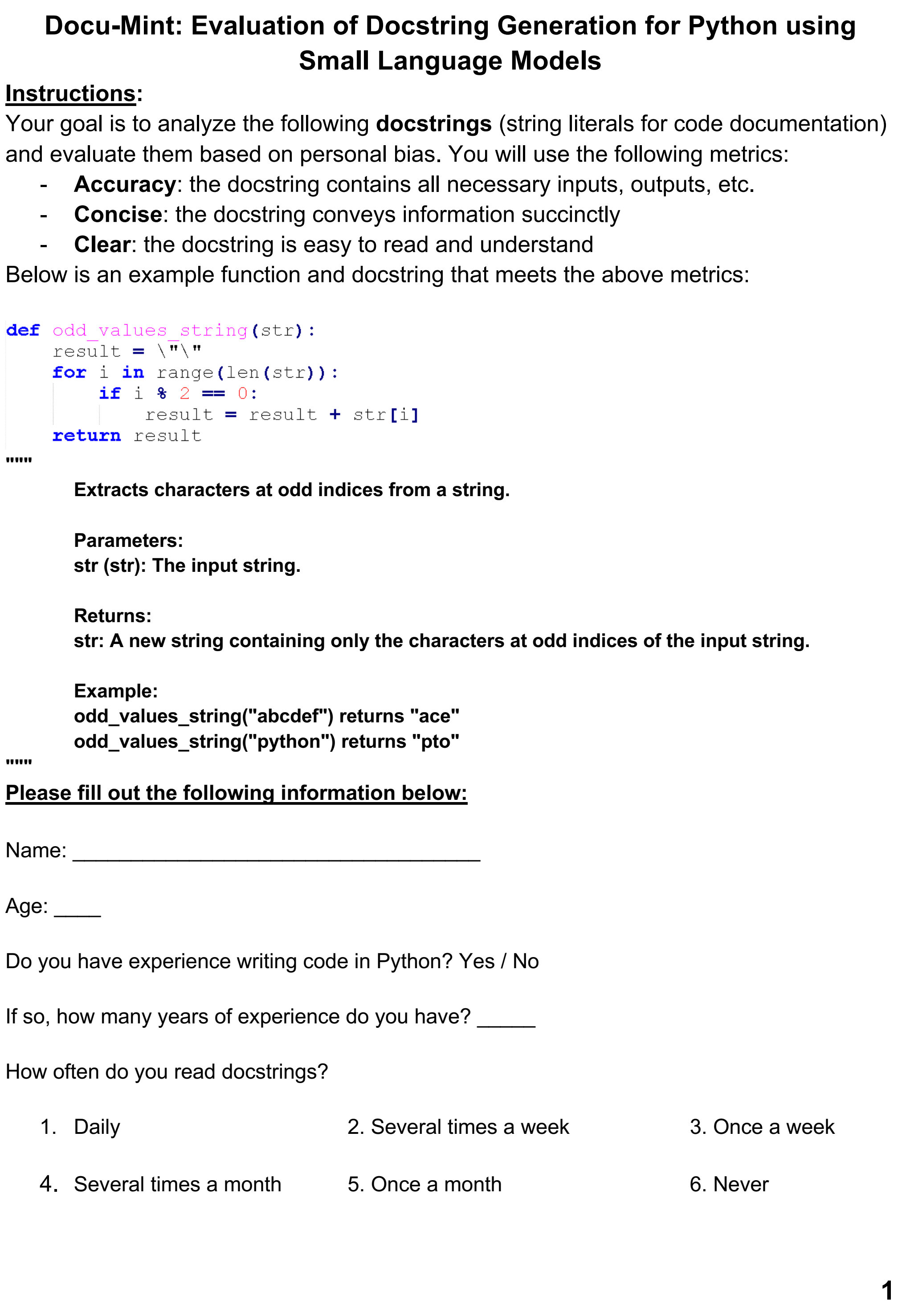
[42] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[43] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989* (2023).

[44] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

A Appendix

A.1 Experiment 2 Human Questionnaire Sample



Docu-Mint: Evaluation of Docstring Generation for Python using Small Language Models

Instructions:

Your goal is to analyze the following **docstrings** (string literals for code documentation) and evaluate them based on personal bias. You will use the following metrics:

- **Accuracy**: the docstring contains all necessary inputs, outputs, etc.
- **Concise**: the docstring conveys information succinctly
- **Clear**: the docstring is easy to read and understand

Below is an example function and docstring that meets the above metrics:

```
def odd_values_string(str):
    result = \"\"
    for i in range(len(str)):
        if i % 2 == 0:
            result = result + str[i]
    return result
"""
      Extracts characters at odd indices from a string.

      Parameters:
      str (str): The input string.

      Returns:
      str: A new string containing only the characters at odd indices of the input string.

      Example:
      odd_values_string("abcdef") returns "ace"
      odd_values_string("python") returns "pto"
"""
```

Please fill out the following information below:

Name: ____________________________________

Age: ____

Do you have experience writing code in Python? Yes / No

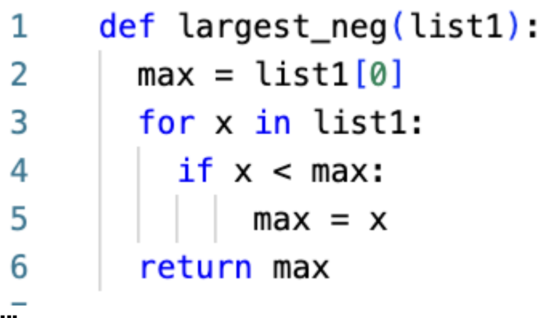
If so, how many years of experience do you have? _____

How often do you read docstrings?

1. Daily	2. Several times a week	3. Once a week
4. Several times a month	5. Once a month	6. Never

1

Question 1:



```python
def largest_neg(list1):
  max = list1[0]
  for x in list1:
    if x < max:
        max = x
  return max
```

"""

Finds the largest negative number in a list.

Args:
   list1: A list of numbers.

Returns:
   The largest negative number in the list.

Raises:
   ValueError: If the list is empty.
"""

On a scale of 1 (poor) to 10 (excellent):

Rate the docstring based on how **accurate** it is:

1 2 3 4 5 6 7 8 9 10

Rate the docstring based on how **concise** it is:

1 2 3 4 5 6 7 8 9 10

Rate the docstring based on how **clear** it is:

1 2 3 4 5 6 7 8 9 10

How would you rate the overall quality of the docstring?

1 2 3 4 5 6 7 8 9 10

Please provide any additional comments