Experimenting with Fine-Tuning-Based Small-Scale Code Summarization

Lorenzo Meninato
NYU, Computer Science
lm4244@nyu

Jack Gindi
NYU, Scientific Computing
jg6848@nyu

Abstract

With the viral proliferation of large language model (LLM) systems such as ChatGPT, Bard, and Bing over the past few months, many programmers are turning to them to for assistance with debugging, documenting, testing, and even writing code. Interestingly, the systems are extremely helpful despite the fact that they are not specifically trained to excel at programming tasks. In this paper, we use the CodeSearchNet dataset to run a selection of code summarization experiments exploring whether we can reproduce some of those emergent results at a smaller scale. In particular, we experiment with different pre-trained architectures and investigate whether models that are fine-tuned on more languages perform better than those that are pre-trained on fewer. While the models we fine-tuned were ultimately not usable, we present an analysis of what we believe went wrong, and suggest avenues that might improve our results.

1 Introduction

Given the centrality of language to human societies, as language models have become more and more capable, people have naturally started to ask whether we are on the cusp of developing human-level, or even superhuman, intelligence. No development has supercharged these conversations more than ChatGPT[1], a recent system based on (Brown et al., 2020) released by OpenAI that allows users to interact with a LLM via a chat interface. Although it is not perfect, ChatGPT has been shown to be capable of mathematical reasoning, programming, writing, test-taking, and other skills.

An interesting feature of many of the large language models that are being developed is that they seem to have emergent capabilities that they were not explicitly oriented toward. One area in which they seem to be especially capable is a variety of programming and programming-adjacent tasks, such as writing, explaining, documenting, and testing code in many different programming languages.

In this paper, we focus on one of those tasks: documentation. We use the CodeSearchNet data set (Husain et al., 2019) and a few different pre-trained sequence-to-sequence architectures attempt to translate code (functions) into a natural language descriptions (documentation strings). It turns out that this task is harder than we thought, and while the models we obtained did not produce useful outputs, the experiment results suggested a few insights that could prove interesting as directions for future work.

The rest of the paper is organized as follows. Section 2 briefly discusses prior work on code summarization and other recent models aimed at the intersection of code and natural language. We also provide some background about the pre-trained architectures we use in our experiments. Section 3 discusses our experiment setups. Section 4 discusses our results. In Section 5, we offer hypotheses that explain our results. Finally, in Section 6, we discuss potential avenues for future exploration.

2 Related Work

Since the release of OpenAI's GPT (Radford et al., 2018), many powerful models for code generation and summarization have been developed. One of the most powerful such models is Codex (Chen et al., 2021), a state-of-the-art code generation model that is gaining traction through use as part of GitHub Copilot. DeepMind has also developed a model called AlphaCode (Li et al., 2022), which is able to generate solutions to more complex competitive programming problems that were better than over half of human contest participants.

One of the early end-to-end neural code summarization algorithms was CODE-NN (Iyer et al., 2016), an LSTM-based recurrent architecture with attention used to produce summaries of SQL

[1] https://chat.openai.com

queries and C# code. This model became a subsequent benchmark against which further advances were compared. In order to better address long-range dependencies and introduce greater computational efficiency, (Ahmad et al., 2020), (Wang et al., 2020), and others built code summarization models using transformers. To incorporate greater information from the full code structure (abstract syntax trees), Leclair et al. (LeClair et al., 2020) uses graph neural networks on Java (code, docstring) pairs.

In this paper, we experiment with three different pre-trained models to try to build our own code summarization engine:

1. T5 (Raffel et al., 2019): an encoder-decoder model pre-trained on supervised (provided by GLUE (Wang et al., 2018) and SuperGLUE (Wang et al., 2019) benchmarks) and self-supervised tasks (e.g. MLM).

2. FLAN-T5 (Chung et al., 2022): instruction fine-tuned T5.

3. BERT (Devlin et al., 2018): bidirectional transformer-based encoder trained on unlabeled text.

3 Experimental Setup

To be able to compare results across experiments, we used the following fine-tuning setup. For each model we fine-tuned, we used a dataset of 30,000 pairs of code and corresponding docstrings as the training set, and evaluated the model using a separate validation set of 5,000 pairs. Initially, we only used Python code, but in a later experiment we used a mixtures of examples in different languages. The maximum sequence length for both code and docstrings was set to 256 tokens, and we fine-tuned the model for 15 epochs. We used a batch size of 8 for both training and evaluation with a 10 gradient accumulation steps, resulting in an effective training batch size of 80.

Every 500 training steps we computed the following metrics: BLEU score (Papineni et al., 2002), ROUGE-1 and ROUGE-2 scores (Lin, 2004). While both BLEU and ROUGE measure the degree of n-gram overlap from the machine generated text to the human reference text, BLEU is precision oriented while ROUGE is recall oriented. For code documentation generation it makes sense to use a recall-oriented metric since it is more important to avoid missing any key information, even if that means including some less relevant information in the summary. However, if it is crucial to be as concise as possible, a precision-oriented metric like BLEU could be more suitable[2].

4 Results

Our initial experiment was to fine-tune three commonly used models: BERT, T5, and Flan-T5. These models are often fine-tuned for more specific downstream tasks.

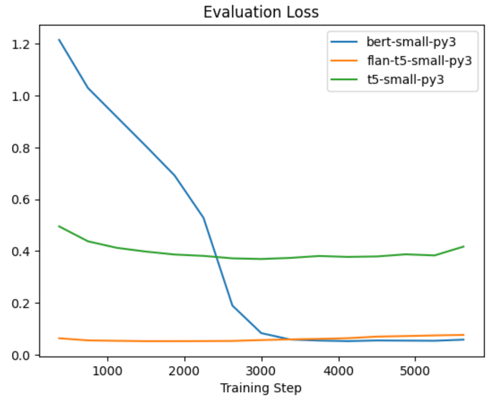

Figure 1: Evaluation loss for BERT, T5, and Flan-T5 trained on Python code.

Although BERT, and Flan-T5 had lower losses, we noticed that for Python examples the documentation for the code was embedded inside the code. This meant that models could score well on various metrics by learning to extract random parts of the function body[3].

To show that the model was indeed over-fitting for code with embedded docstrings, we ran two fine-tuning runs. In one, we used the same set-up as before, using unmodified code examples, and in the other we used pre-processed the code and used an AST parsing library to remove docstrings from as many examples as possible. (Some of the examples were written in Python 2, which the library could not parse. If an example could not be parsed, it was included as-is. About 83% of examples were stripped successfully.) In Figure 2, we see that loss is much higher when fine-tuning on examples without docstrings and conversely ROUGE-2 scores are much lower. This suggests that the scores prior to stripping were significantly attributable to the presence of the docstrings in the code. We also

[2] In either case we compute both metrics. For brevity we only display plots of ROUGE-2 scores.

[3] See `https://peps.python.org/pep-0257/` for examples

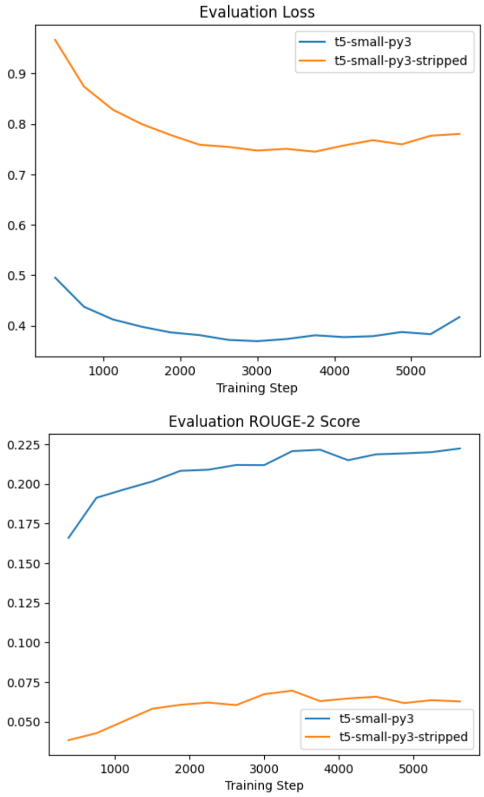

Figure 2: Evaluation loss and ROUGE-2 scores with and without docstring stripping when fine-tuning T5 model.

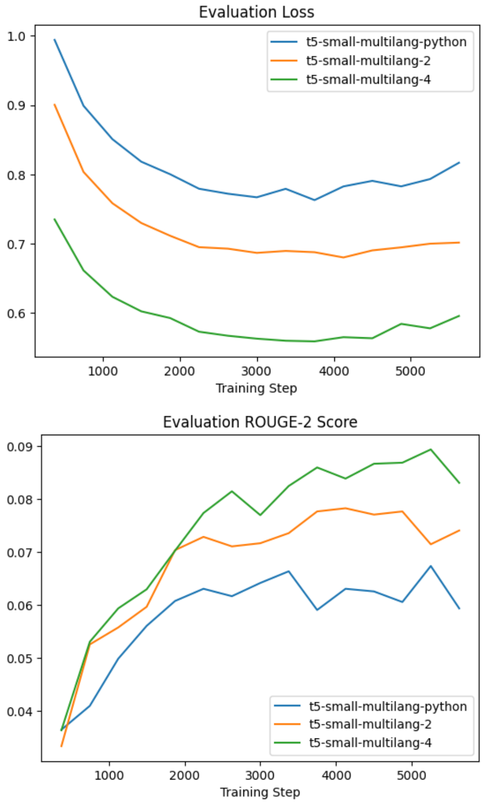

Figure 3: Evaluation loss and ROUGE-2 scores training on a even mixture of languages: (Python), (Python, Java), and (Python, Java, JavaScript, Go).

manually verified that the examples would often copy the docstring straight into the output.

Next, we wanted to see if fine-tuning on a mixture of programming languages, rather than just Python, would improve performance.

Although the size of the training and validation data stayed the same (30k and 5k examples, respectively), Figure 3 shows that the loss *decreased* and BLEU/ROUGE-1/ROUGE-2 scores *increased* when increasing the number of languages. This result that suggests training on multiple languages could improve code documentation generation. While results did improve with additional languages, it was still difficult to obtain results that would be useful for humans at the scale of our training experiments. For instance, the model fine-tuned on 4 languages produces the following when asked to summarize a simple function:

```
def sum_list_of_numbers(x):
  result = 0
  for num in x:
    result += num
  return result
```

The model simply repeats back the prompt and the function text, which is clearly not helpful to a human user.

5 Discussion

Two aspects of our work are worth discussing in greater detail. The first is why adding languages seems to have improved performance. The second is a broader discussion of why code summarization might be difficult at small scale using existing pre-trained models not geared explicitly toward code. We will discuss each of these in turn.

5.1 Why do more languages work better?

The primary reason we think more languages lead to better performance is that by being exposed to different code patterns and styles, the model learns more general patterns that can be more reliably translated into documentation strings.

An interesting way of thinking about this effect is as a kind of regularization introduced by the dataset itself. By including data of different languages, the model cannot as easily overfit to patterns of a single language, and must come up with more robust ways of representing code that work well across languages.

5.2 Why were our results poor?

One unfortunate, inescapable fact of our work is that the models we fine-tuned did not produce useful output. In this section we discuss a few reasons for why our results, both qualitative and quantitative, might have been so poor.

Data Quality Upon manual inspection of the Python docstrings in the early phases of the project, we noticed that many docstrings did not contain much useful information. For example, we found that many docstrings only contained names of parameters, without any description of what the function does. Others contained descriptions, but ones which were very short and not very useful. These types of examples may have confused the model and made it difficult to learn useful relationships between functions and their docstrings.

Scale Another issue might have been scale in a number of different senses. Many of the general purpose language models such as ChatGPT that excel at code-related tasks are relatively large, and are trained on massive datasets, for very long periods of time. We did not have access to enough time or compute to run our experiments on larger models fine-tuned on more data, but suspect that if we did, our results would have improved.

Code vs. Natural Language We believe that one of the things that made our fine-tuning task difficult is that the pre-trained models we used were pre-trained on natural language. As anyone who has studied programming knows, programming languages differ significantly from natural language in a variety of aspects, including structure, syntax, the role of ambiguity, and vocabulary, among others. These differences may make it less straightforward to fine-tune a model that has captured features of natural language toward understanding and effectively summarizing code.

6 Conclusion and Future Work

In this paper, we discussed experiments we ran to try to produce a small-scale code summarization model by fine-tuning existing models pre-trained on natural language. Developing smaller machine learning models is an important goal for a variety of reasons, ranging from reducing environmental impact to enabling more people to be able to take advantage of the technology. While our effort did not end up producing a useful model, there are a few avenues for further investigation that we think might be fruitful.

First, we think it would be worth spending time to curate a large, high quality dataset that contains data representing different languages and paradigms (e.g., declarative vs. imperative, object-oriented vs. functional). Many of the (open) datasets in the literature contain a small number of languages. Careful thought would be required to design the types of filters and checks to ensure high quality, but the effort might be well worth it. In addition to GitHub and StackOverflow[4], data could also be sourced from programming interview preparation websites such as LeetCode[5], HackerRank[6], which have a plethora of problems with lengthy descriptions. (The challenge with the latter would be getting access to correct code solutions, but they might be amenable to sharing for academic purposes. For LeetCode specifically, users often post their solutions to the discussion sections for a given question, so it might be possible to scrape solutions that way as well.)

Second, it would be interesting to study how much further scale, in terms of training data size and model size, would help model quality. We would want to figure out how much of the bad performance is attributable to small scale, rather than the inherent difficulty of the problem. As noted earlier, many of the state of the art LLMs are very large with parameter counts in the billions, some even approaching trillions.

Finally, the T5 family of models we tried were trained on a variety of natural language tasks. It is possible that performance on code summarization is suffering because it is being polluted by the other tasks whose knowledge the pre-trained model is based on. We would be interested in further investigating whether training a code summarization model from scratch would produce a more useful model.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *CoRR*, abs/2005.00653.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie

[4] https://stackoverflow.com/

[5] https://leetcode.com/

[6] https://www.hackerrank.com/

Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *CoRR*, abs/2005.14165.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2022. Scaling instruction-finetuned language models.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. *CoRR*, abs/2004.02843.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. Superglue: A stickier benchmark for general-purpose language understanding systems. *CoRR*, abs/1905.00537.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *CoRR*, abs/1804.07461.

Ruyun Wang, Hanwen Zhang, Guoliang Lu, Lei Lyu, and Chen Lyu. 2020. Fret: Functional reinforced transformer with bert for code summarization. *IEEE Access*, 8:135591–135604.