PyMT5: multi-mode translation of natural language and PYTHON code with transformers

Colin B. Clement[*]
Microsoft Cloud and AI
coclemen@microsoft.com

Dawn Drain
Microsoft Cloud and AI
dadrain@microsoft.com

Jonathan Timcheck[†]
Stanford University
timcheck@stanford.edu

Alexey Svyatkovskiy
Microsoft Cloud and AI
alsvyatk@microsoft.com

Neel Sundaresan
Microsoft Cloud and AI
neels@microsoft.com

Abstract

Simultaneously modeling source code and natural language has many exciting applications in automated software development and understanding. Pursuant to achieving such technology, we introduce PyMT5, the PYTHON method text-to-text transfer transformer, which is trained to translate between all pairs of PYTHON method feature combinations: a single model that can both predict whole methods from natural language documentation strings (docstrings) and summarize code into docstrings of any common style. We present an analysis and modeling effort of a large-scale parallel corpus of 26 million PYTHON methods and 7.7 million method-docstring pairs, demonstrating that for docstring and method generation, PyMT5 outperforms similarly-sized auto-regressive language models (GPT2) which were English pre-trained or randomly initialized. On the CODESEARCHNET test set, our best model predicts 92.1% syntactically correct method bodies, achieved a BLEU score of 8.59 for method generation and 16.3 for docstring generation (summarization), and achieved a ROUGE-L F-score of 24.8 for method generation and 36.7 for docstring generation.

1 Introduction

Software is a keystone of modern society, touching billions of people through services and devices daily. Writing and documenting the source code of this software are challenging and labor-intensive tasks; software developers need to repeatedly refer to online documentation resources in order to understand existing code bases to make progress. Developer productivity can be improved by the presence of source code documentation and a development environment featuring intelligent, machine-learning-based code completion and analysis tools.

Recent progress in natural language processing (NLP), especially encoder/decoder-based transformer models (Vaswani et al., 2017) and pre-training (Radford et al., 2018; Lewis et al., 2019), has led to state-of-the-art performance on language modeling, classification (Devlin et al., 2018), translation (Raffel et al., 2019), summarization (Liu and Lap-

[*]Corresponding author
[†]Work done during a Microsoft internship

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, pages 9052–9065,
November 16–20, 2020.

ata, 2019), grammar correction (Bryant et al., 2017), entity recognition, dialogue generation (Budzianowski and Vulić, 2019), and more. Along with these quantitative advances have come deeper understanding of the learned hidden representations which power transformers (Kovaleva et al., 2019; Voita et al., 2019; Clark et al., 2019; Ethayarajh, 2019). While they are arguably not 'natural,' programming languages are increasingly becoming modeling playgrounds for NLP modeling. Since these languages by definition have a grammar, syntax, and known relationships between entities, they offer enticing opportunities for an even deeper probing of NLP models and tasks. Beyond theoretical importance, many NLP tasks have practical utility in software development environments: language modeling or generation can be used for code completion (Raychev et al., 2014; Bruch et al., 2009; Svyatkovskiy et al., 2019, 2020), translation/summarization to generate documentation or natural language summaries (Moreno et al., 2013; Scalabrino et al., 2017; Wan et al., 2018; Alon et al., 2018) or even summarize a set of code changes (Moreno et al., 2014), translation and grammar error correction to patch and detect bugs (Zhai et al., 2019), and joint embedding of code and natural language for code search (Husain et al., 2019; Gu et al., 2018).

In this work we focus on jointly modeling both source code (PYTHON) and concomitant natural language documentation (docstrings) with transformers, through the study of dual tasks: generating method code bodies from signatures and docstrings, and generating docstrings from signatures and method code bodies. While previous work (Allamanis et al., 2015; Yin and Neubig, 2017) has leveraged the grammar of code to extract features like the Abstract Syntax Tree for modeling (treating code and natural language as separate modalities), we follow examples like Barone and Sennrich (2017) and treat PYTHON and its docstrings as fundamentally no different than other 'natural' languages, representing both source code and natural language docstrings as sequences of tokens sharing the same vocabulary. Here we present a multi-mode translation method resulting in PYMT5, the PYTHON method text-to-text transfer transformer (inspired by the text-to-text transfer transformer T5 (Raffel et al., 2019)). Our single model can both learn code/language generation and understand the relationships between them.

The paper is organized as follows: we begin in sec. 2 by presenting examples of the performance of our novel multi-mode PYMT5 —the PYTHON method text-to-text transfer transformer model—which we trained to translate between all pairs of combinations of method signatures, docstrings, and bodies which do not have the same feature in both the source and target. In sec. 2.1 we describe our training data and the pre-processing steps for source code and natural language we followed, and compared it to existing parallel docstring-method corpora like CODESEARCHNET (CSN)(Husain et al., 2019) and that presented by Barone et al (Barone and Sennrich, 2017). In sec.2.2 we explain our BART-like (Lewis et al., 2019) pre-training scheme, demonstrating a 25$\times$ speed-up in training time for docstring generation. Next, in sec. 2.3 we analyze and classify PYTHON docstrings, enabling style-conditioned docstring generation in PYMT5. In sections 3 and 4, we discuss PYMT5 results on method generation and docstring generation respectively and compare it to two GPT2 models randomly initialized and pre-trained on English.

2 Multi-mode training

Figure 1 shows examples of inputs and outputs of our model PYMT5 for 3 example tasks: (top, blue) predicting a body from a method

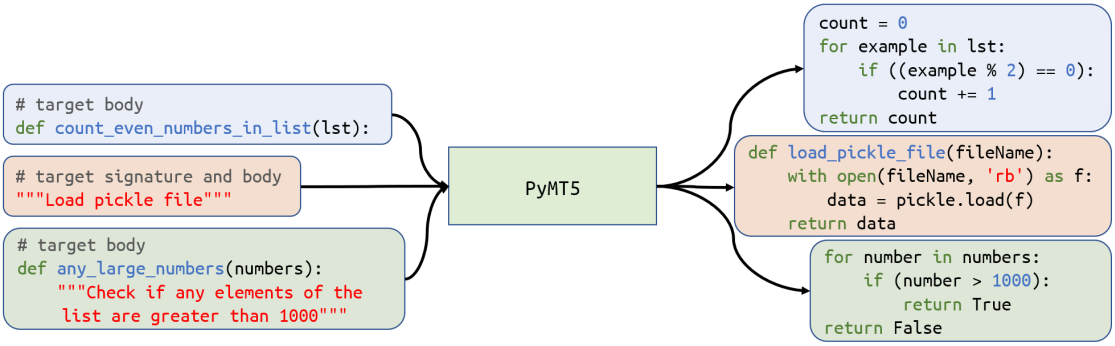

Figure 1: Real examples of PyMT5 performing method generation using combinations of signatures and docstrings. A leading comment in the input sequence instructs the model to output a particular combination of features, e.g. '`# target signature and body`' instructs PyMT5 to predict both a signature and body.

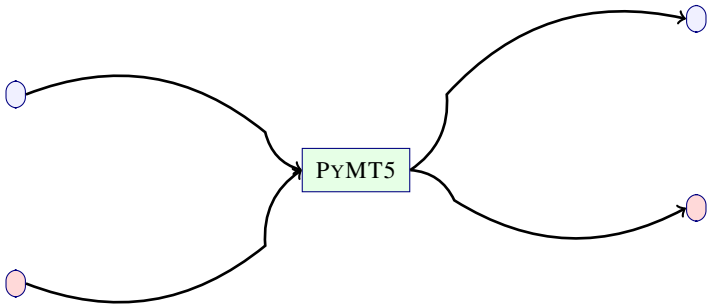

Figure 2: PyMT5 performing docstring generation on an example method, showing the output when the target prefix indicates one line (top, blue) and Numpydoc docstring (bottom, red) styles.

signature, (middle, red) predicting a whole method from a natural language docstring, and (bottom, green) predicting a body from a signature and docstring. Note that the comment '`# target <specification>`' instructs the model to choose a particular form of output. Further note that PyMT5 correctly learns to interpret natural language: it interprets '`even`' as being related to '`(example %2) == 0`', and '`greater than 1000`' as '`number > 1000`'. The model also produces syntactically correct code (as we will discuss later, we never show the model syntactically incorrect code), and correctly infers the types of '`lst`' and '`numbers`' to be iterables containing numbers.

PyMT5 can also be prompted with source code to produce a docstring summary in various styles. Figure 2 shows the model prompted with one of the methods generated by PyMT5 in Fig. 1 (top, blue), in both a 'one line' (top, blue) style and a 'Numpydoc' (bottom, red) style. It infers the intent from the signature name and code, and even infers that type of the argument is a `list` and return type `int`. It produces the same terse one sentence summary of the function in both cases.

In order to teach PyMT5 to maximally relate the separate method features (signatures, docstrings, bodies), we trained it to translate

between all pairs of feature combinations in which the same feature does not appear in both the source and target. This scheme is also advantageous as our corpus is unbalanced, with only 1/5 methods featuring docstrings, and so the model can learn to leverage all the features whether they are present or not. Additionally, it has been shown that code is more 'predictable' than natural language (Hindle et al., 2012). If the method and argument names are a dominating signal due to their relatively rigid structure, the model may learn to ignore the content of docstrings. This multi-mode method overcomes that by training the model to generate method bodies from docstrings alone. See the appendix for a more detailed description of the multi-mode training scheme.

2.1 Dataset

Our data consists of 118k GITHUB repositories, which includes all public repositories labelled as containing primarily PYTHON source code, featuring at least 10 stars, and which have had a commit in the past 5 years. We successfully cloned 112k of these repositories, extracting 5.3 million PYTHON files from the default `HEAD` state of each repository. We then removed literal duplicate files, resulting in 2.3 million unique files, but did not remove finer-grained clones. After removing license from the files, the literal contents were used in the pre-training step, comprising about 27GB of raw text.

In order to extract method-level information for fine-tuning, we used the `python3.7` standard library `ast` to produce the file-level Abstract Syntax Tree (AST) for each PYTHON file, extracting every individual and class method. For each file which failed to parse, we used `2to3` and `autopep8` to overcome the issue of different styles and white space or tab conventions, successfully parsing 97.3% of the 2.3 million unique PYTHON files. We used the PYTHON module `astunparse` to take the AST for each method and unparse them back into source code, so that our fine-tuned model was never trained on syntactically incorrect code. The statistics of our method-docstring corpus are summarized in Table. 1. Our parallel method-docstring corpus is twice as large as the next largest irrespective of language and over 15× as large as the next largest PYTHON parallel corpus, both in CSN.

For each method, we ignored comments as they generally represent trivia and are not part of the normal language syntax. We cleaned the docstrings by removing non-ASCII characters, normalizing Unicode, and replacing commit hashes, file paths, and URLs with placeholder tokens. In all studies here, we randomly split the files at the repository level (to prevent data leakage) with 90% for training, 5% for validation, and 5% for a test set.

2.2 Pre-training

The majority of our PYTHON methods—over 20 million methods— do not possess docstrings. This imbalance is, in fact, an opportunity in light of the recent trend for NLP: unsupervised pre-training of language models on vast amounts of raw text (Devlin et al., 2018). Using these pre-trained models as starting points for downstream tasks—like classification, translation, summarization, and question answering—consistently yields state-of-the-art results (Lewis et al., 2019; Raffel et al., 2019).

Following this trend, we use a similar span-masking objective used by the recent text-to-text transfer transformer (T5) (Raffel et al., 2019). As shown in Figure 3, after tokenizing the inputs, we sample a random subset of the token spans up to length 3 to be replaced with, e.g. a `[MASK0]` token, and then teach the sequence-to-sequence model to replace the missing tokens. The training target is com-

Dataset	Methods	w/ docstring	Languages
PyMT5	2.6×10^7	7.7×10^6	Python
CSN (Husain et al., 2019)	6.4×10^6	2.3×10^6	Python, et al.
Ciurumelea et al. (2020)	1.6×10^5	1.6×10^5	Python
Barone and Sennrich (2017)	1.6×10^5	1.5×10^5	Python

Table 1: Summary statistics of our Python parallel corpus compared to others presented in the literature. CSN contains 500k Python methods with docstrings, among 6 other languages. Our parallel corpus is 3× as large as the next largest, and over 15× the size of the next largest Python parallel corpus.

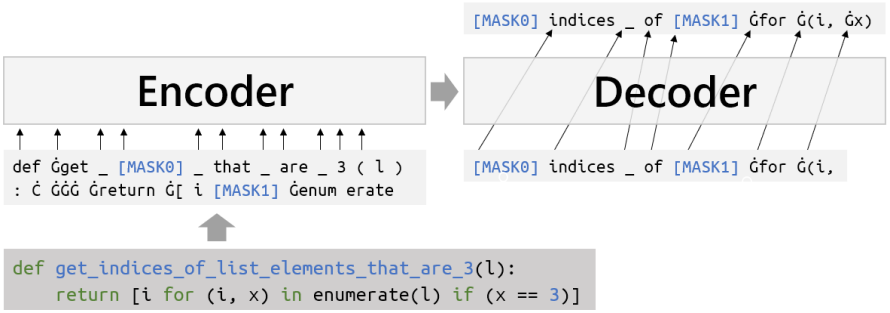

Figure 3: Denoising auto-encoder pre-training for sequence-to-sequence tasks, based on the span-masking objective used by the T5 (Raffel et al., 2019). Python files are first tokenized with spaces replaced by the character Ġ, which is 256 in ordinal above the space character (similarly for newlines, tabs, etc.). Note that indentation is a token of multiple Ġ's. We replace random sub-sequences of tokens with numbered masks, and train the model to return each mask followed by the tokens it replaced.

prised of numbered mask tokens followed by the tokens that mask represents.

The architecture of PyMT5 is an encode-decoder transformer with a vocabulary of 50181 (byte-pair BPE encoder trained on raw python files), 6 self-attention encoder/decoder layers in each encoder layers, and a hidden dimension of 1472, totaling 374 million parameters. All the experiments in this paper, including GPT2 were done using this same extended GPT tokenizer. We pre-trained PyMT5 on 27GB of raw source code in total, for 3 weeks on sixteen 32GB Tesla V100 GPUs, or 73 epochs total. When training on docstring generation alone, we observed 25× faster convergence to a lower loss when starting with this pre-trained model as compared to a random initialization. See the appendix for details. In all experiments PyMT5 is trained starting with this pre-trained model.

2.3 Docstring analysis

When examining docstring samples from our corpus, one of the most salient features is the different styles of documentation. The Python community has no prescribed or de facto style for docstrings, but Python enhancement protocol 257 (Goodger and van Rossum, 2001) does describe one-line and multi-line docstrings, and mandates indentation as well. Most modern large-scale projects utilize docstring styles which are parseable, allowing the automatic creation and synchronization of source code and documentation websites, see, e.g. `sphinx`. Therefore, a number of standard styles have evolved in the community.

The currently dominant parseable docstring

styles (and the ones supported by `sphinx`) are reSTructuredText (reST) (Jones, 2013), the official Google style (Google, 2020), Numpy style (also technically satisfies reST) (Maintainers, 2020), and Javadoc style (jav, 2011). The difference between each style is mainly in the syntax of denoting sections (if they exist) and the name/type/description annotation of the method arguments and returned/yielded quantities (if they exist). We defined, in addition to these styles, one-line (containing only one line), one-paragraph (containing no empty lines), and 'other' to label any docstring not described so far, which includes informal user docstring styles and a few project-specific styles like the SAGE mathematics toolkit library.

Table 2 shows the breakdown of the fraction of each of these styles in our corpus. The plurality of docstrings (44%) are one-line. The next most common style is one-paragraph at 14%. The next four most-common styles are the machine parseable styles discussed above, comprising 26.2% of the total number of docstrings. The appendix contains detailed distributions of method signature, docstring, and method body character and line lengths.

Style	Fraction of methods
One line	44%
One paragraph	14%
reST	13%
Google	7.3%
Numpy	4.8%
Javadoc	1.6%
Other	15%

Table 2: Docstring style statistics from 7.7 million Pythondocstrings.

To visualize the space of these styles, we used FastText vector embeddings of the docstrings, obtaining 100-dimension continuous vector representations of each. We then used PCA to reduce the dimensionality to 50 and applied the t-distributed stochastic neighbor embedding (t-SNE) to obtain a two-dimensional visualization. Figure 4 shows 1/10th of our corpus (700k docstrings) embedded, colored by docstring style as defined above. We can see clear clustering of styles, indicating that similar docstrings use the same style (for the parseable styles). There is also a natural dichotomy between parseable and non-parseable styles: the left side is dominated by 'one line,' 'one paragraph,' and 'other' styles, and the four parseable styles are largely on the right side. This observation can be used to generate documentation consistent with the style of a given project, or it could be used to translate methods into more informal descriptions useful for search indices.

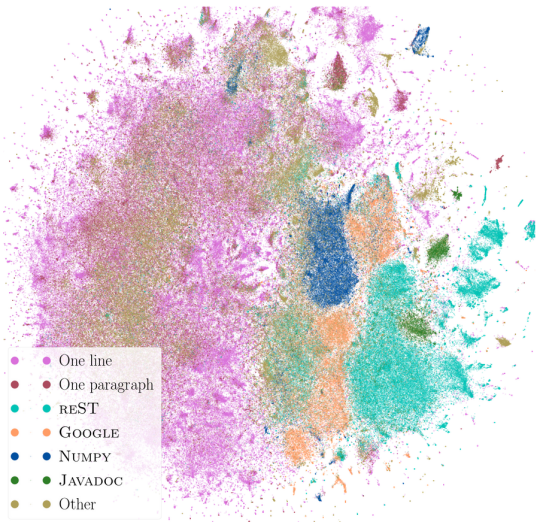

Figure 4: Visualization of continuous embeddings of 1/10th of our docstring corpus (770k docstrings), colored by docstring style. Embeddings were obtained using FastText, and the two-dimensional embedding was obtained via PCA (for dimensionality reduction and initialization) and t-SNE.

Model	Ppl	BLEU	Syntax	Stat.	R1	R2	RL
GPT2-med random	2.36	5.60	85%	Prec.	25.8	12.3	26.8
				Rec.	26.7	12.1	25.9
				F1	21.8	10.6	22.5
GPT2-med english	**2.09**	5.63	86%	Prec.	25.4	12.1	26.3
				Rec.	26.9	12.2	26.1
				F1	21.7	10.6	22.5
PyMT5	2.36	**10.6**	**93.6%**	Prec.	**33.8**	**21.5**	**33.6**
				Rec.	**44.1**	**25.0**	**43.8**
				F1	**35.1**	**21.5**	**32.2**
CSN test:							
GPT2-med random	–	2.8	77.2%	Prec.	**32.3**	11.8	**33.7**
				Rec.	19.6	7.0	19.3
				F1	20.9	7.6	21.9
PyMT5	–	**8.59**	**92.1%**	Prec.	25.6	**12.5**	25.3
				Rec.	**40.2**	**18.3**	**39.6**
				F1	**28.4**	**13.5**	**24.8**
Barone and Sennrich (2017) test:							
PyMT5	–	20.2	91.1%	Prec.	41.3	28.5	40.7
				Rec.	52.2	34.7	51.3
				F1	43.2	29.8	39.7
Barone et al.	–	10.9	–	–	–	–	

Table 3: Comparing 3 models–GPT2 with a random weight initialization, GPT2 pre-trained on English, and PyMT5–on the task of method generation from a signature and natural language docstring. The first three rows use our test set consisting of 1,285,794 methods. The fourth and fifth rows compare the performance of PyMT5 and GPT2-medium on the CodeSearchNet Python test set. The final rows compare the performance of PyMT5 on the parallel corpus test set of Barone and Sennrich (2017). Syntax is the fraction of predicted methods which had correct syntax using the Python 3.7 grammar.

3 Method generation

Now we turn our attention to method generation: predicting a whole method code body from either a method signature, a natural language docstring, or both. We first discuss a benchmark of this task using a GPT2-medium model (345 million parameters, see the appendix for details), training from scratch and starting with the publicly released OpenAI English pre-trained checkpoint with weights from HuggingFace(Wolf et al., 2019). In all experiments we used an extended GPT2 tokenizer—including white-space (one tab, two tabs, etc.) tokens—for a total vocabulary size of 50337, and we used beam decoding with a beam width of 5.

The third row of tab. 3 shows PyMT5 has more than double the BLEU score, overall better recall, and significantly better ROUGE-2 and ROUGE-L F-scores than our GPT2 baselines. Further, 93.6% of the methods generated by PyMT5 were syntactically correct Python 3.7, whereas only 86% of GPT2 methods were syntactically correct. PyMT5 was trained on 16 Tesla V100 16GB GPUs for 62 epochs, or 5 weeks training time (see the appendix for its hyper-parameters) and the GPT2 baselines were trained on the same hardware for 1 week training time (achieving the same or better validation loss/perplexity as PyMT5).

The English pre-trained initialization of GPT2 only slightly beats the random initialization of GPT2, which could indicate that the learned biases of English are not particularly beneficial for writing Python code; the metrics are almost all within our margin of error. Note that Barone and Sennrich (2017) also modeled methods from docstrings, obtaining a similar BLEU score of 10.9 on their own Python parallel corpus. On the Barone et al. test set, PyMT5 obtains nearly double these scores at 20.2; such a large discrepancy could be explained by data leaking from their test set

Model	Ppl	BLEU		R1	R2	RL
GPT2-med	2.36	19.4	P	32.6	19.3	33.6
random			R	36.2	19.4	34.7
			F1	31.0	18.2	31.6
GPT2-med	**2.15**	19.6	P	33.1	19.4	33.9
English			R	36.4	19.5	34.8
			F1	31.4	18.3	31.8
PyMT5	3.74	**25.2**	P	**42.1**	**23.7**	**41.3**
			R	**50.4**	**27.0**	**49.3**
			F1	**43.3**	**24.4**	**39.8**
CSN test:						
GPT2-med	–	9.5	P	30.6	13.3	31.4
random			R	31.1	12.9	29.8
			F1	26.3	11.5	27.2
PyMT5	–	**16.3**	P	**38.0**	**19.2**	**36.8**
			R	**52.7**	**24.5**	**51.0**
			F1	**41.3**	**20.4**	**36.7**
Barone test:						
PyMT5	–	17.4	P	39.6	26.0	38.7
			R	53.6	33.7	52.1
			F1	43.1	27.8	39.1
Barone et al.	–	13.84	–	–	–	–

Table 4: Comparing 3 models–GPT2 with a random weight initialization, GPT2 pre-trained on English, and PyMT5–on the task of natural language docstring generation from a signature and method body. The first three rows are evaluated on our test set of 383695 methods. The fourth and fifth rows shows performance of PyMT5 and GPT2-medium on the CSN Python test set, and the last two rows compare our model to Barone et al. on their test set.

into our training set. Barone's test set is also 200× smaller than ours and may not be a representative sample of the whole Python code domain.

The third and fourth rows of tab. 3 show the performance of PyMT5 using the publicly available CSN Python test set, from which we find notably worse results than on our own test set. CSN curated their whole set by removing any methods with 'test' in the name and any methods with fewer than 3 lines of code. We calculated the performance of PyMT5 only on a subset of our test set curated the same way as CSN, observing F-scores for R1, R2, and R-L on our test set of 29.7, 17.2, and 26.1, which is lower than our nominal test set performance of 35.1, 21.5, and 32.2 and closer to the CSN performance of 28.4, 13.5, and 24.8. We believe this curating choice explains the difference between our test set and the CSN test set. We also conclude that tests and short methods are 'easier' to complete, which is plausible, and bodes well for automatic code completion applications.

4 Docstring Generation

We now examine results from the docstring generation task, which for evaluation purposes were conditioned on both signatures and method bodies. As in method generation, we set a GPT2 benchmark with random initialization and pre-trained English initialization as well as the same hyperparameters. Table 4 shows that the ROUGE scores of the GPT2 baselines are within the margin of error; a somewhat surprising result given the English domain of docstrings. The third row shows PyMT5 to be superior to GPT2-medium in terms of BLEU and all of the ROUGE metrics.

We again present the results from the publicly available CSN test set. Similar to the method generation task, PyMT5 performs worse on the CSN data than our own, likely for the same reasons we discussed in sec. 3. We also evaluated PyMT5 on the Barone et al. parallel test set, as shown in the second to last row of tab. 4, and find PyMT5 performs notably worse on Barone's test set than our own test set, contradicting the hypothesis that our doubling of the method generation BLEU score is due to data leakage. PyMT5 has a much higher BLEU score than that reported by Barone et al, perhaps indicating real progress in the code summarization field.

Docstring generation is similar to code summarization, though the domains are different as docstrings also contain structured annotations of arguments, return values, raised exceptions, and even in-line unit tests (`doctest`). TranS^3 by Wang et al. (Wang et al., 2020) reports a best ROUGE-L of 51.27 on the same test set for code summarization, but does not specify

which statistic they are reporting, so we cannot make strong conclusions about the performance of PyMT5 compared to the state of the art.

5 Conclusion

In this work, we presented a novel multi-mode PYTHON method text-to-text transfer transformer model PyMT5as well as the largest parallel corpus of PYTHON source code and docstrings reported in the literature to date. We have trained PyMT5 to translate between all pairs of combinations of method signatures, docstrings, and method bodies which do not have the same feature in both the source and target. Further, we introduced control token prefixes for docstring generation to facilitate docstring generation of various styles. Focusing on two modeling tasks – predicting PYTHON methods from docstrings and summarizing PYTHON source code methods into docstrings of various commonly occurring styles – we have compared this new approach to the auto-regressive GPT2 baselines trained on individual docstring or method generation tasks. On the CODESEARCHNET test set PyMT5 achieves a BLEU score of 8.59 for method generation and 16.3 for docstring generation, and a ROUGE-L F-score of 24.8 for method generation and 36.7 for docstring generation. We have demonstrated the effectiveness of dynamic masked pre-training, reducing docstring generation training time by 25×. Looking forward, we plan to leverage PyMT5 for various downstream automated software engineering tasks—including code documentation and method generation from natural language statements—and develop more model evaluation criteria to leverage the unique properties of source codes.

Acknowledgements

We would like to thank the Microsoft Cloud and AI SmartML engineering team for help in preparing the data, Shao Kun Deng for the development of compelling user experiences leveraging PyMT5, and Christian Bird for useful discussions.

References

2011. Java doc. Technical report.

Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 2123–2132. JMLR.org.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

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*.

Marcel Bruch, Martin Monperrus, and Mira Mezini. 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*, pages 213–222.

Christopher Bryant, Mariano Felice, and Edward Briscoe. 2017. Automatic annotation and evaluation of error types for grammatical error correction. Association for Computational Linguistics.

Paweł Budzianowski and Ivan Vulić. 2019. Hello, it's gpt-2–how can i help you? towards the use of pretrained language models for task-oriented dialogue systems. *arXiv preprint arXiv:1907.05774*.

Adelina Ciurumelea, Sebastian Proksch, and Harald Gall. 2020. Suggesting comment completions for python using neural language models. In *27th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341*.

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*.

Kawin Ethayarajh. 2019. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. *arXiv preprint arXiv:1909.00512*.

David Goodger and Guido van Rossum. 2001. Docstring conventions. PEP 257.

Google. 2020. Google python style guide. Technical report.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA. Association for Computing Machinery.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE.

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*.

Richard Jones. 2013. A restructuredtext primer. *docutils. sourceforge. net, March*.

Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. Revealing the dark secrets of bert. *arXiv preprint arXiv:1908.08593*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Yang Liu and Mirella Lapata. 2019. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*.

Numpydoc Maintainers. 2020. Numpydoc docstring guide. Technical report.

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE.

Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*.

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. *arXiv preprint arXiv:1910.10683*.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428.

Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys

Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Elena Voita, Rico Sennrich, and Ivan Titov. 2019. The bottom-up evolution of representations in the transformer: A study with machine translation and language modeling objectives. *arXiv preprint arXiv:1909.01380*.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. Transˆ 3: A transformer-based framework for unifying code summarization and code search. *arXiv preprint arXiv:2003.03238*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2019. Cpc: automatically classifying and propagating natural language comments via program analysis.

A Appendix

A.1 Docstring statistics

Figure 5 shows the distributions of various features of docstrings in our corpus. The top row is the distribution of total character-level length of the method signatures (left), docstrings (center), and code bodies. The blue lines are for methods possessing a docstring, and we can see that the vast majority of these methods have docstrings with more than 10 characters. The bottom row shows the distribution of line lengths of the concomitant features from the top row. While the most common line length of docstrings is 1 (comprising 41%), the vast majority of docstrings have multiple lines.

A.2 Pre-training details

Figure 7 is the complete training script, using the Facebook AI Research Sequence (FAIRSEQ) modeling library, with which we pre-trained PYMT5. The data was pre-noised and processed using the `fairseq-preprocess` command, and placed in the directory indicated by `$DIR`. The architecture and training hyper-parameters are set in this script. PYMT5 was trained with the same hyperparameters, but with data described in sec.A.4.

Figure 7 shows learning curves of a single seq2seq model of the same architecture as PYMT5 trained only on docstrings, starting from random initializations, and starting from our pre-trained model. As the figure shows, the

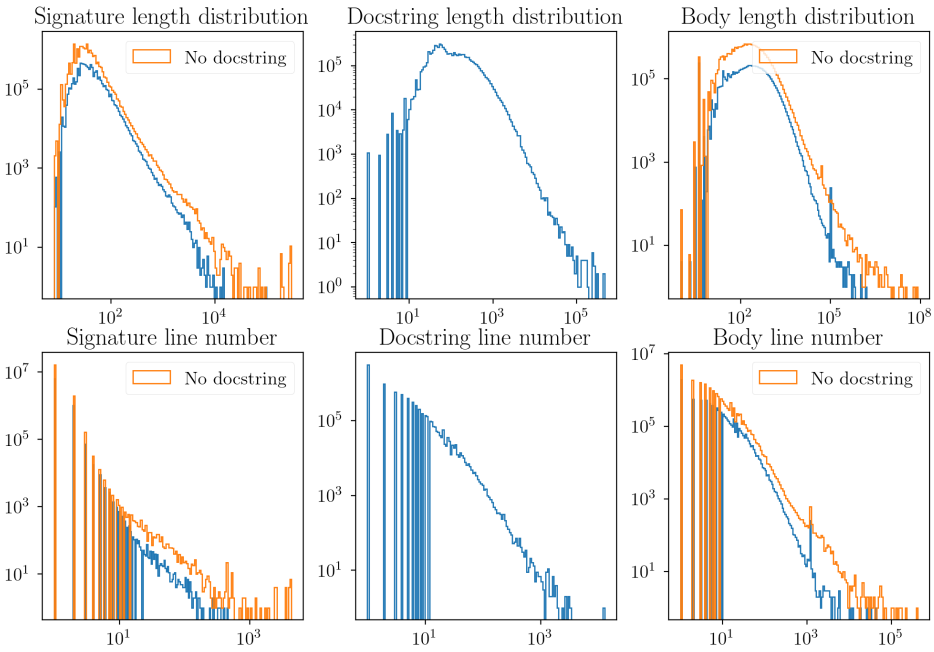

Figure 5: Histogram of the number of characters (top row) in the PYTHON signatures (left), docstrings (middle), and method body (right). The blue lines are for methods with docstrings, the yellow lines are for methods without docstrings. The vast majority of docstrings have more than 10 characters. The bottom row shows histograms of the number of lines for the same features described in the top row.

pre-trained initialization converged to a better validation loss 25× faster than the randomly initialized model.

A.3 GPT2 training details

Our GPT2 experiments also used the FAIRSEQ library, with the OpenAI English checkpoint supplied by the HuggingFace library. Figure 8 shows the complete training script, where for the English pre-trained initialization a pre-trained checkpoint was provided. Each models was trained on 4 Tesla V100 GPUs with 16GB of memory each, for 7 days.

A.4 Multi-mode training details

In order to better teach PYMT5 to understand the relationships between all the different features of code (signatures, docstrings, and bodies) we taught it to translate between

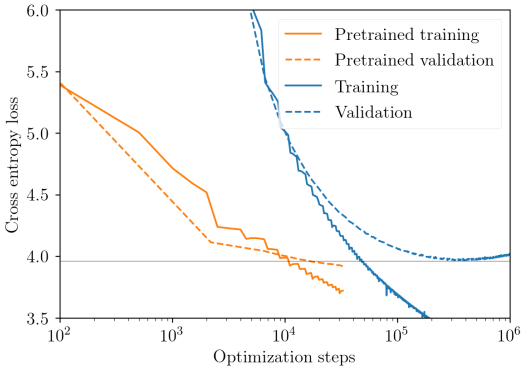

Figure 6: Learning curves for training a sequence-to-sequence transformer, translating from python method definitions to their docstrings. Blue curves represent the training and validation loss, and show that convergence (validation loss stops decreasing) occurs after 3.97×10^5 steps or 183 epochs. The optimization of the pre-trained model with identical hyperparameters reaches and beats the best validation loss at 1.5×10^4 steps or 7 epochs.

Figure 7: The `fairseq-train` script used to pre-train PyMT5, setting all the relevant hyper-parameters.

Figure 8: The `fairseq-train` script we used to train our GPT model baselines

all pairs of combinations of these features which do not contain the same feature in both the source and target. In this way, the model can learn to produce method bodies using both signatures and docstrings, or one or the other. Table 5 spells out exactly which combinations were provided to the model as a source and target. For each source example the comment string '`# target <feature> (<style>)`' was added, instructing the model which feature combination (e.g. signature and body). Only if a docstring was in the target, a style imperative was added, where the styles are defined and discussed in the main text.

Figure 9 shows the training curves for PyMT5, where the solid black line is the training loss, and all the other curves are the validation loss for each of the tasks indicated in tab. 5. The dashed lines indicate tasks where docstrings are present in the target, showing that these are generally less predictable than code-only targets (as the validation loss is larger). PyMT5was trained on 16 Tesla V100 16GB GPUs for 62 epochs, or 5 weeks training time.

Targets \ Sources	Signature	Docstring	Body	Sig + doc	Sig + body	Doc + body
Signature		✓	✓			✓
Docstring	✓		✓		✓	
Body	✓	✓		✓		
Sig + doc			✓			
Sig + body		✓				
Doc + body	✓					

Table 5: A table of all possible translation possibilities between the 3 features of a function: the signature (sig), docstring (doc), and body. We train our model to translate between sources and targets indicated with a ✓, which were chosen as all pairs of feature combinations which do not contain the same feature in both the source and target. The system is then instructed to target code bodies when performing function completion.

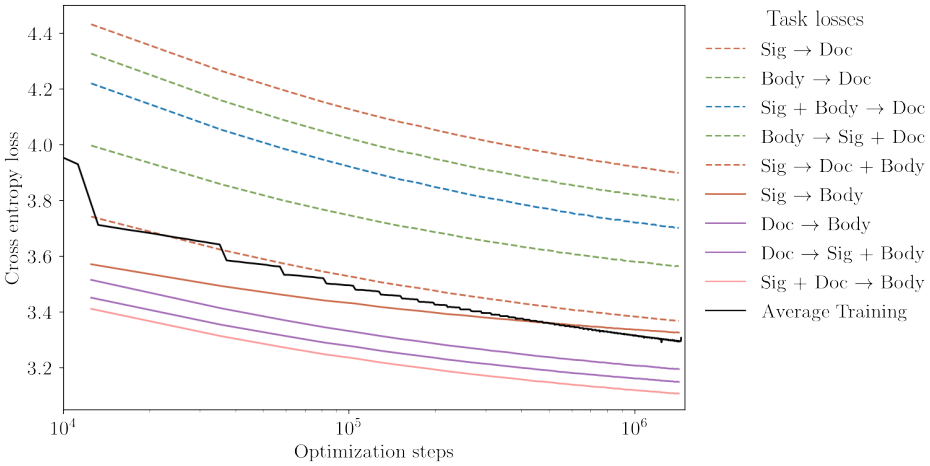

Figure 9: Learning curve for the multi-mode training, where the black line is the training loss, and the other lines are the validation loss for each mode of translation. Dashed lines indicate the docstrings are in the target, solid lines have only code in the target.