Neural code summarization: Experiments in Python and Bash

Benjamin N. Peloquin
Department of Psychology
Stanford University
Stanford, CA 94305
`bpeloqui@stanford.edu`

Abstract

Code summarization, the task of generating natural language descriptions from code, has clear applications in domains such as code search, automated documentation, and programming learning environments. Despite these potential applications, academic research has placed more emphasis on the related, but reverse task of program synthesis from natural language. Given the success of neural machine translation applied to natural languages and the increase of large-scale, publicly available datasets of source code and natural language documentation, recent work has examined code summarization using end-to-end neural architectures. In this work we build on these recent advances, exploring neural code summarization in two novel domains: Bash scripting and Python programs. We cast the problem in terms of neural machine translation, comparing several sequence to sequence architectures. We find that character-level modeling appears particular important in this domain and describe the challenges that make code summarization distinct from vanilla machine translation with natural languages.

1 Introduction

While code generation from natural language is a long-standing goal of artificial intelligence research (Sammet, 1966), less work has examined the reverse task – generating natural language from code. Code summarization has clear productivity and pedagogical implications both in improving code search applications (Giordani & Moschitti, 2012), automating function documentation, and providing tools for beginner programmers to understand new or complex code. Despite these applications, research on code summarization has been fairly limited, primarily relying on hand-engineered templates restricted to specific domains (Sridhara et al, 2010). Recent approaches have re-framed the problem in terms of neural machine translation (NMT) – adopting a fully data-driven approach using sequence to sequence (seq2seq) architectures mapping code to natural language (Iyer et al. 2016).

In part, this re-framing as NMT follows the recent success of neural approaches in language to language translation (Sutsekever et al., 2014). Equally important, however, is the rise of open source software projects and the availability of public data repositories linking code and natural language. These datasets present an opportunity to pursue neural approaches on data sizes that approximate those used in standard language-to-language NMT, avoiding the need to curate hand-annotated datasets. Dyer et al. (2013) estimate that over a billion lines of source code are publicly available online – likely a dramatic under-estimate of the amount of code available in 2019. Importantly, such code is commonly annotated with natural language documentation in the form of comments, docstrings or other descriptive content. Publicly available data combining programming and natural language potentially represents an opportunity for work in both the program synthesis and code summarization communities. However, it also raises a new set of challenges primarily stemming from issues of data quality. As we will highlight in this work, task difficult is directly related to data complexity and open-source datasets present high degrees of variability in quality and content leading to difficult learning environments.

In this work, we examine the task of code summarization in two novel domains – Bash scripting and Python programming. To our knowledge this is the first project which has examined code summarization in these domains. We explore task performance with a set of neural models that vary in architectural and data-modeling considerations. In the following sections we describe the limited set of previous projects which have examined this task. We then describe the set of models we will explore. To preview our results, we find good performance on the more constrained task of generating natural language descriptions of Bash commands with significant improvement emerging from character-level models. By contrast we find poor performance on the much harder task of Python docstring generation, however we do again see improvements stemming from learning character-level representations. Following our discussion of results, we present an analysis of the issue of data-complexity focusing primarily on the Python dataset.

2 Related work

Despite the increases in the availability of parallel corpora of natural language and code and the success of neural methods on language to language translation, little work has examined the task of code summarization using end-to-end neural methods. For example, previous work has examined the task of predicting class-level comments by learning n-gram and topic models from open source Java projects (Mavshovitz-Attias & Cohen, 2013). Others have attempted to create models for suggesting method and class names by learning high-dimensional embedding spaces (Allamnis et al. 2015a). Perhaps closest to the actual task of code summarization, Wong & Mooney (2007) introduced a system that learns to generate natural language sentences from lambda calculus expressions via a semantic parser.

The first completely data-driven approach to code summarization is presented by Iyer et al. (2018) who explored LSTM-based models to translate C# and SQL into natural language (English) descriptions. The authors collected a dataset of 66,015 C# and 32,337 SQL title/query pairs from the website StackOverflow – a popular website for posting programming related questions and answers. They trained token-level LSTM with additive attention. Interestingly, they did not use a sequence to sequence architecture. Rather they generated natural language summaries by attending to a code snippet producing an intermediate representation based on the current LSTM hidden state. A combination of this intermediate representation and the hidden state are used to generate the next natural language token which is fed to the next LSTM cell. This is repeated until a fixed number of words or <END> token are generated. They did not find improvements using pre-trained embeddings. They evaluated their system using METEOR, sentence-level Bleu-4, and also employed human evaluation to assess the "naturalness" and "informativeness" of the generated summaries. They found significant improvements over a set of non-neural baselines across all their metrics achieving a highest Bleu score of 0.205 on the C# dataset.

3 Code summarization as neural machine translation

We follow Iyer et al. (2016) in adopting a fully end-to-end neural approach to code summarization. We depart from their exact formulation in that we frame the task in terms of neural machine translation (NMT) employing seq2seq architectures. Our problem formulation is straightforward, we examine two datasets of parallel corpora of code and natural language. First we examine a hand-curated dataset of Bash commands and natural language descriptions previously collected by Lin et al. (2016). Second we examine a much larger dataset of python function and docstring pairs recently released by github. Among the benefits from learning end-to-end neural models for this task is that we can more easily scale to larger and more diverse datasets unlike previous approaches that involved hand-engineered grammars, manual template engineering, or other forms of hand-crafted featurization (Ngonga Ngomo et al., 2013; Koutrika et al., 2010).

	NL-bash	Bash-script	NL-docstring	Python function
n	9,305	9,305	310,092	310,092
max	129	63	125	10
min	3	1	3	4
mean	15.1	9.8	11.1	6.6
sd	7.5	6.2	8.2	1.9

Table 1: Bash / Python dataset length statistics. Note that we manually limited Python functions to between 4-10 tokens.

3.1 LSTM models

We used the seq2seq formulation specified by Sustkever et al. (2014) training a two-layer LSTM encoder and two-layer bi-LSTM decoder with multiplicative attention for 30 epochs. Our system generates a summary one word at a time, using attention over the source code and the context of previously generated words by the LSTM model. We did not find significant improvements using pre-trained embeddings and learned word embeddings from scratch using a dimension size of 256. Among various hyper-parameter and architectural considerations we found that results were most sensitive to learning- and dropout-rates, tuning these parametesr to the validation set. We found best results with a learning rate of $\alpha = 1e^{-5}$, $dropout = 0.5$ in the Bash dataset and $\alpha = 1e^{-4}$, $dropout = 0.3$ in the Python dataset. We used the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-9}$. In general, we report results from the best performing model chosen via validation set loss – a form of regularization via early stopping.

3.1.1 Character-based

We explored a character-level LSTM model using a vocabulary of 96 commonly used ASCII characters, learning character-level embeddings from scratch. Training the character-level model was orders of magnitude slower than training token-level models. In fact, at the time of writing we are still training the Python character-level model, but report results for the best model as of Tuesday, March 19th. Despite the slower training, results on the Bash dataset indicate that character-level representations may be especially important for learning tasks that involve code. In particular, function and variable naming leads to high degrees of variability between programs and complicates standard notions of semantics. The problem of function and variable name semantics appears to be a central concern particularly in datasets collected from real-world data – functions may have identical semantic behavior, but vastly different surface forms (the actual characters used).

3.1.2 Token-based

We explore a token-level LSTM model using regex-based tokenization supported the NLTK library. This model treats words as the basic unit rather than characters and more closely follows standard NMT approaches. Importantly, as we will see in the case of the Python data, increasing dataset size leads to linear increases in vocabulary size – a unique problem in working with code data.

3.2 Transformer model

We used the Transformer formulation specified by Vaswani et al. (2017) modifying existing implementations from OpenNMT[1]. We trained the model for 30 epochs with minimal hyperparameter tuning. Our encoder model was composed of stack of $n = 6$ layers with multi-head attention using $h = 8$ heads on the Bash dataset and $n = 3$ layers and $h = 4$ heads on the Python dataset. These decreases were primarily related to GPU memory constraints. Query, key and value projections were assigned dimensions of 64 each and we used a drop-out rate of 0.3. In addition to the attention sub-layers, each layer in both the encoders and decoders contained position-wise fully connected layers with a hidden layer of dimension 2048 in Bash and 1024 in Python. The model was optimized using the Adam optimizer with parameters $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-9}$, varying the learning rate over training according to the formula described in Vaswani et al. (2017) using 4000 warm-up steps.

[1]`https://github.com/OpenNMT/OpenNMT-py/blob/master/onmt/encoders/transformer.py`

3.3 Naive baseline

As a baseline we examine Bleu scores under a naive bi-gram language model trained on natural language data (either Bash command descriptions or Python docstrings). During description generation we forward sampled from the language model to produce a description where the probability of generating a given description of length n is given by:

$$p(description) = p(token_1|SOS) \times p(token_2|token_1) \times \ldots \times p(EOS|token_n) \qquad (1)$$

Note that this baseline does not condition on the source code at all, but reflects the underlying distribution of tokens in the training set. We expect poor performance, however this baseline provides relevant context for our analysis of BLEU.

4 Experiment 1: Bash to English

Bash is a Unix Shell and command language used as the default login shell for most Linux distributions as well as Apple's macOS (`https://en.wikipedia.org/wiki/Bash_(Unix_shell)`). Shell commands typically consist of three components – a utility (e.g. find, grep), optional flags (e.g. -name, -i), and arguments (e.g. "*.java", "TODO"). There are over 250 Bash utilities with new ones being added by third party developers.

4.1 Dataset

Lin et al. (2018) released a dataset containing 9,305 Bash commands and expert-written descriptions with coverage over 102 unique Bash utilities using 206 different flags. The corpus was scraped from websites (QA forums, tutorials, tech-blogs, course materials) and annotation was performed via crowd-sourcing on a platform that included freelancers comfortable with shell-scripting. We define our own train/val/test splits from the 9K examples, with splits of 0.80/0.10/0.10.

4.1.1 Examples

1. Example use of the "grep" utility:

   ```
   grep -l "TODO" *.java
   ```

   find .java files in the current directory tree that
   contain the pattern 'TODO' and print their names

2. Example use of the "find" utility

   ```
   find . -type f | sort -nk 5,5 | tail -5
   ```

   display the 5 largest files in the current directory and
   its sub-directories

3. Example use of the "tar" utility

   ```
   tar -cvf images.tar $(find / -type f -name *.jpg)
   ```

   search for all jpg images on the system and archive
   them to tar ball "images.tar"

Table 1 displays descriptive statistics for Bash commands and natural language descriptions in Lin et al. (2018). Both programs and descriptions tend to be fairly short (average length approximately 10 and 15 tokens, respectively). However, this is a many-to-many mapping problem – multiple bash programs can be described by multiple natural language descriptions and vice versa. This variance is true even of utility semantics which tend to be the most stable in the dataset. In other words, a description such as "find all the files in ..." can often be accomplished with multiple utilities (e.g. "find" or "ls"). Perhaps most difficult, however is free variable naming. In addition to utility commands and option flags, each command references a unique piece of data, specific to the context of a given example.

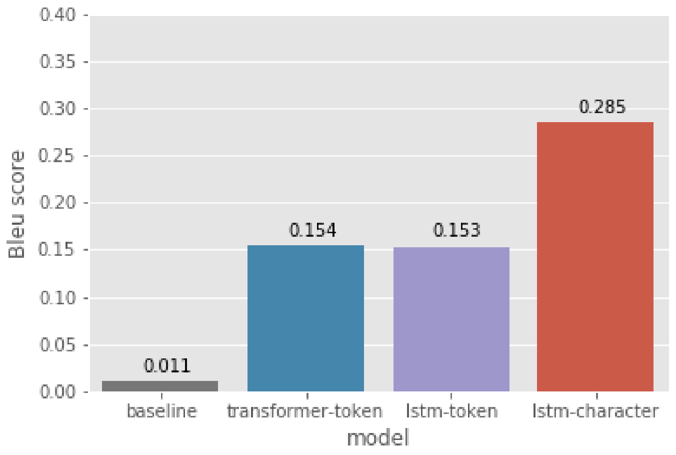
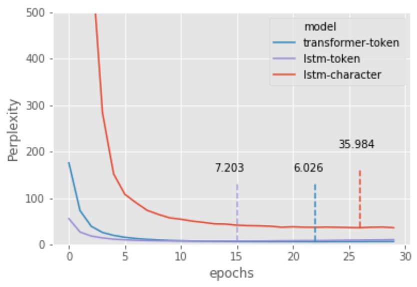
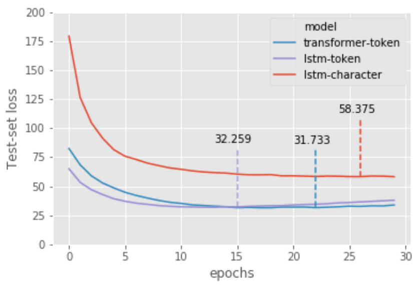


Figure 1: Test-set Bleu scores for each model on the Bash dataset.



(a) Perplexity 1



(b) Test-set loss

Figure 2: Left panel displays perplexity values over training epochs with the value from the best-model annotated. Right panel displays average loss values on the test set with the value from the best-model annotated.

4.2 Results

Figure 1 compares test set Bleu scores for the Bash dataset. While imperfect, Bleu (Papineni et al. 2002) is commonly used to assess machine translation tasks and is a interpolation of n-gram overlap of predicted and gold-set strings. On the Bash dataset we see significant gains moving from token- to character-level modeling with our highest Bleu score of 0.285 with the character-level LSTM model. Both token-level LSTM and Tansformer models outperform the naive baseline, but do not approach levels of the character-model. Qualitative evaluation of model outputs indicate that the success of the character-level model is due in part to handling OOV items, which token-level models cannot accommodate. While we cannot compare our model directly to results found by Iyer et al. (2016), we do find Bleu scores at and above what they found on both the C# and SQL datasets. Figure 2 left panel plots model perplexity values over training epochs with the best model perplexity values annotated. Interestingly, the character-level LSTM has a higher language perplexity than either of the token-level models. Figure 2, right panel displays test-set loss values. Again despite the higher Bleu-score the character-level LSTM model has higher test-set loss than either of the token-level models. In general, we find that token-level models appear to learn better utility semantics, but struggle to handle variable naming, which leads to the discrepancy in Bleu-score and test-loss statistics.

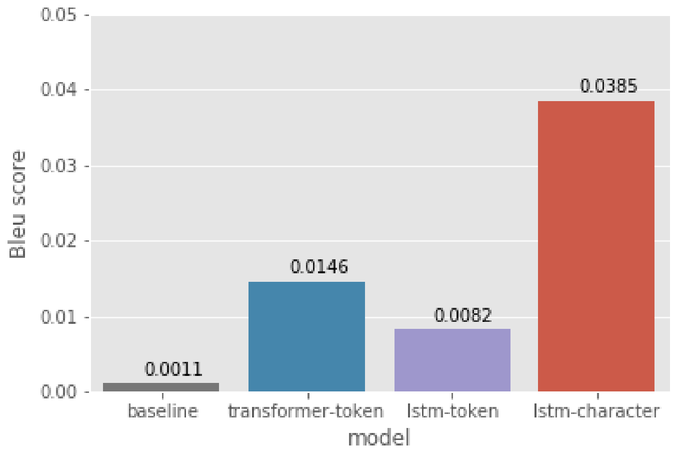


Figure 3: Test-set Bleu scores for each model on the Python dataset.

5 Experiment 2: Python to English

Python is a high-level, general purpose programming language which is distinctive for its readability and use of white-space. Unlike Bash, we do not begin a-priori with a set of semantically valid commands aside from a set of reserved words for function declaration and control flow (e.g. "def", "return", "for", etc). A typical python function, however, will be consistent in that it contains some function name, optional set of arguments, and function body. This semantic variability makes the task of code summarization with Python distinctly harder than for Bash.

5.1 Dataset

We examine a subset of a recently released dataset of python functions and docstrings publicly available on github[2]. In total, this dataset contains over 1.2 million docstring/function pairs. Inspection of the dataset indicated a high degree of variance in the quality of documentation. There are many different conventions for docstring usage[3], which receive variable amounts of adherence. For the current project we restrict our analysis to functions that vary in length from 4-10 tokens. Functions shorter than 4 tokens typically had empty function bodies (e.g. were a class function that was not yet implemented). Functions longer than 10 tokens had a high degree of variance in their documentation. Functions within the 4-10 token window tended to have some content and meaningful natural language descriptions in the docstrings. After sub-setting the remaining dataset set of $n = 310,092$ docstrings/function pairs we partitioned into train/val/test sets with proportions 0.8/0.1/0.1.

5.1.1 Preprocessing

We pre-processeed the data restricting the set of valid characters to a limited set of ASCII letters and basic arithmetic operators removing additional control flow structures such as parentheticals, commas, colons, and semi-colons.

5.1.2 Examples

1. Short function, short description

   ```
   def pupil_size return 19
   ```

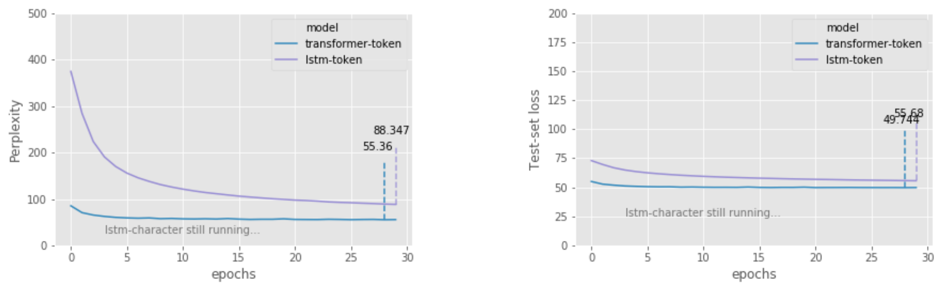
   returns dummy pupil size

2. Short function, long description

   ```
   def should_continue return true
   ```

[2] https://github.com/hamelsmu/code_search

[3] https://www.python.org/dev/peps/pep-0257/



(a) Perplexity 1 (b) Test-set loss

Figure 4: Left panel displays perplexity values over training epochs with the value from the best-model annotated. Right panel displays average loss values on the test set with the value from the best-model annotated.

hook to allow implementation to stop the algorithm early . defaults to true , meaning explore the entire solutions space

3. Long function, long description

```
def enabled return _editor value_to_boolean _editor get_element_value
_extension_config find enabled
```

gets or sets a boolean value that describes if the extension is enabled

4. Long function, short description

```
def connected message dummy mode eyetracker
not connected return true
```

gets or sets a boolean value that describes if the extension is enabled

6 Results

Examining the example docstring/function pairs highlights the difficulty of this task. There is a high degree of variability in the documentation – while some descriptions provide high-quality annotation for function behavior, others provide far more detailed information than appears necessary, while others appear woefully incomplete. Figure 3 plots test-set Bleu values among our models. Note that performance is an order of magnitude worse on this dataset compare to the Bash data. However, we once again see an advantage for character-level representations with a highest Bleu score of 0.0385 for the character LSTM model. Both the token-level transformer and LSTM models outperform the naive baseline. At the time of writing the character-level models were still running so we report best results without having finished all training epochs.

7 Discussion

In this work, we explore the task of summarising code with end-to-end neural models. This work builds on recent studies which have adopted NMT methods for the code summarization task, applying the methods in two novel domains - generating natural language descriptions for Bash commands and Python functions. We achieved relatively strong performance on the Bash dataset, achieving a highest Bleu score nearing 0.3 for the character-LSTM model. Future work should consider combinations of token and sub-token level models as in Luong & Manning (2016) or combinations of Transformer and LSTM based encoder / decoders.

Results on the Python dataset were far below those obtained on the Bash dataset with a highest Bleu score of 0.0385 for the character-level LSTM model. The Python dataset presents a far more challenging learning task as the quality and correspondence of docstrings and functions is highly variable.

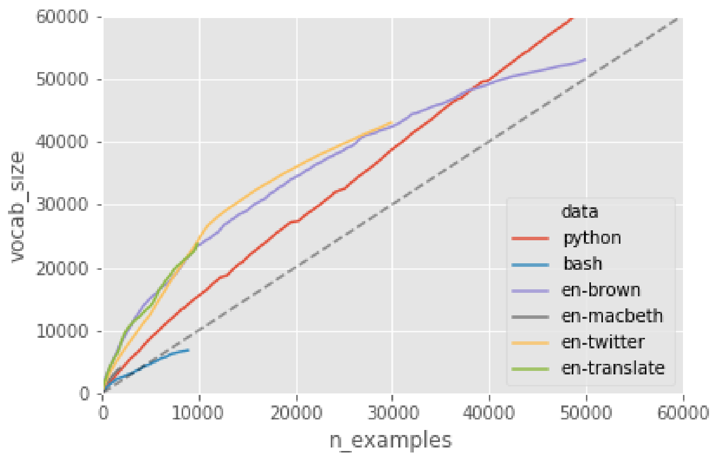


Figure 5: Linear growth in vocabulary growth for Python data exceeds growth in natural language datasets. Horizontal axis plots the number of examples. Verticle access plots the vocabular size.

7.1 Linear vocabulary growth scaling in Python data

One dimension that makes modeling the Python data particular challenging is that while in most natural language datasets the vocabulary scales sub-linearly with the number of examples, in the Python dataset vocabulary growth scales linearly. Figure 5 plots the relationship between vocublary size and number of training examples. While datasets containing natural language including a corpus of 30k tweets, the Brown corpus, and a commonly used English dataset for translation all display approximately logarithmic growth in vocabulary size as a function of training examples, vocabulary size increases linearly for Python. This isn't surprising – there's a high degree of variability in function and variable naming between different programs. In fact, there are an infinite possible number of functions that could possibly accomplish the same task. This connects to a deeper issue of how to best capture function semantics, which are necessarily defined within the context of a particular program environment as opposed to externally (as we have with natural language and even to some extent with Bash utilities and option flags).

8 Conclusion

We presented a set of experiments framing code summarization as neural machine translation in two novel domains - Bash scripting and Python programming. Our results on the Bash dataset indicate that there is clear utility in approaching code summarization using NMT techniques, especially when modeling sub-token level units on constrained languages. By contrast, results on the Python dataset and our subsequent analysis indicate that simply treating programs and documentation as parallel corpora may be problematic with more real-world datasets. Future work should examine the degree to which we can gain traction on such datsets by further pre-processing – such as function and variable anonymization, or pre-training representations using GloVe or word2vec algorithms on domain-specific code corpora.

References

[1] Allamanis, M., Barr, E., Bird, C. & Sutton, C. (2015a). Suggesting accurate method and class names. In Proceedings of the 2-15 10th Joint Meeting on Foundations of Software Engineering.

[2] Dong, L. & Lapata, M. (2016). Language to Logical Form with Neural Attention. Association for Computational Linguistics.

[3] Dyer, R., Nguyen H.A., Rajan, H. & Nguen T. (2013). Boa: A language and infrastructure for analyizing ultra-large-scale software repositories. In Proceedings of the 2013 International Conference on Software Engineering.

[4] Giordani, A. & Moschitti, A. (2012). Translating questions to SQL queries with generative parsers discriminatively reranked. In Proceedings of COLING 2012.

[5] Gu, J., Lu, Z., Li, H. & Li, V. (2016). Incorporating Copying Mechanism in Sequence-to-Sequence Learning. Association for Computational Linguistics.

[6] Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., Patwary, M.D., Yang, Y. & Zhou, Y (2017). Deep Learning Scaling Is Predictable, Empirically. arXiv:1712.00409v1 [cs.LG | stat.ML]

[7] Kant, N. (2018). Recent Advances in Neural Program Synthesis. arXiv:1802.02353v1 [cs.AI | cs.PL]

[8] Iyer, S., Konstas, I., Cheung, A. & Zettlemoyer, L. (2016). Summarizing Source Code using Neural Attention Model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016.

[9] Lin, Xi., Wang, Chenglong., Zettlemoyer, Luke., & Ernst, Michael D. (2018). NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. International conference on language resources and evaluation.

[10] Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kocisky, T., Wang, F., & Senior, A. (2016). Latent predictor networks for code generation. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany.

[11] Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N. & Barzilay, R. (2016). Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. Empirical Methods in Natural Language Processing.

[12] Luong, M. & Manning, C. (2016). Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models. In Proceedings of the 41st annual meeting on computational linguistics.

[13] Oda, Y., Neubig, H., Hata, H., Sakti, S., Toda, T., & Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation. In 30th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society.

[14] Papineni, K., Roukos, S., Ward, T. & Zhu, W. (2002). Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on computational linguistics.

[15] Sutsekever, I., Venyals, O., & Le, Q.V. (2014). Sequence to sequence learning with neural networks. In Proceedings of the 27th International Conerence on Neural Information Processing Systems, NIPS'14, pgs 3104-3112, Cambridge, MA, MIT Press.

[16] Wong, Y. & Mooney, R. (2007). Generation by inverting a semantic parser that uses statistical machine translation. In Proceedings of the 2007 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.

[17] Zavershynzkyi, M., Skidanov, A. & Polosukhin, I. (2018). NAPS: Natural Program Synthesis Datasets. Neural Abstract Machines Program Induction (NAMPI) at ICML.