Generating Documentation from Code

Rishi Subramanian & Weiran Guo

June 10, 2019

1 Task Description and Motivation

Documenting and explaining code is an important part of software engineering, and is sometimes considered to be more important than writing the code itself. By one measure, developers spend nearly 60% of their time on these activities [4]. Additionally, many programmers prefer writing code to English and neglect documentation for one reason or another.

This report is itself documentation of a software project. If we could develop an automated method to generate documentation, we would have saved several hours ☺.

2 Prior Work

Previously, the task of generating text from code has previously been done with the goal of creating ways to better retrieve information from sites such as Stack Overflow [5][4]. We want to generate documentation, which we assume to be different from summarization. We have observed that good documentation is often as long or longer than the code itself, and believe that summarization is a poor fit for this task.

Barone and Sennrich [1] attempted to treat this problem as a regular bilingual translation problem, but their main contribution was the creation of a code-text corpus and the translation results were only provided as a benchmark.

Our contribution differs from [1] since we have collected a much larger corpus, which will hopefully allow us to train better models. Additionally, the previous attempts use LSTM Seq2Seq models, which have largely been supplanted by transformers. We apply the current state of state in machine translation to this problem, which may also lead to better results.

3 Overview

Figure 1 provides an overview of our process, from data collection to the final model evaluation.

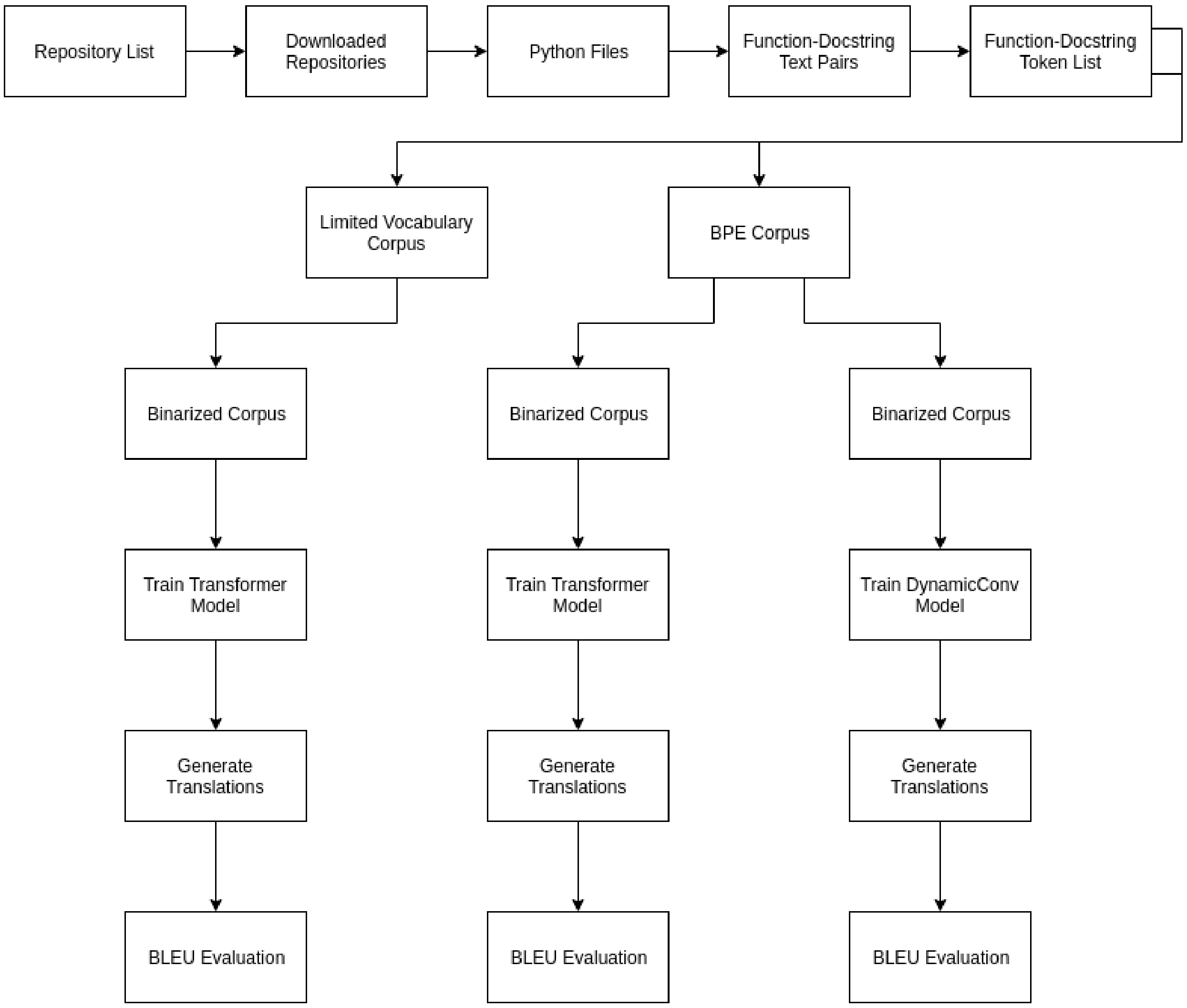


Figure 1: Flowchart overview of our project

4 Dataset

4.1 Collection

We initially decided to use Python as our target language, since we were very familiar with it and the standard library provides convenient tools for parsing code. In addition, Python documentation strings are a core feature of the language and are represented as expressions rather than comments like in Javadoc or Doxygen.

We obtained a list of Github repositories from the Public Git Archive [9], which contains all Github repositories with at least 50 stars as of January 1, 2018. The list contains nearly 190 thousand repositories, of which about 40 thousand had at least one Python file. To reduce network and storage requirements, we only considered repositories with at least ten Python files, leaving us with 16 thousand repositories. We wrote a script which cloned all of these repositories and discarded the revision history and metadata. This dataset contained about 1.5 million files with the `.py` extension.

We had hoped to use the data collected by Markovtsev and Long [9], but the distributed version comes in the form of nonstandard archives which were very slow to process for our requirements. Instead, we obtained their metadata as a CSV file.

4.2 Cleaning

Since we were only interested in function-documentation pairs for translation, we used the `ast` module in the Python standard library to parse each file, then walked through the generated abstract syntax trees to select subtrees containing function definitions and corresponding docstrings. We then used the external `astor` library to convert the subtrees back to valid python code. This had the effect of cleaning comments and extraneous whitespace, as well as ensuring that all of the code was formatted consistently.

If a file could not be parsed, we ignored it. One notable example of this is the `print` statement in Python 2 which was replaced with a function in Python 3, which excluded many Python 2 files from our dataset.

After we extracted the function-docstring pairs, we removed those with documentation not in English. We used the `langdetect` package. However, since even documentation in foreign languages contains some English words to describe the code, we observed a few false negatives produced by the detector.

After this cleaning and preprocessing, we were left with 2.8 million function-documentation pairs. This is small compared to natural language datasets. For example, the WMT2014 English-German corpus has 4.5 million sentence pairs while the English-French corpus has 36 million sentence pairs. However, we do not split our text into sentences since splitting functions in a way that preserves correspondences with the text would be difficult.

Our corpus is also much larger than the one published by Barone and Sennrich [1], which contains just over 150 thousand pairs.

We applied a 90-5-5% train-valid-test split and shuffled the data to ensure that files from each project were evenly distributed among the splits.

4.3 Preprocessing

To ensure proper and consistent tokenization of both code and text, we performed our own tokenization rather than rely on Fairseq's preprocessing.

We tokenized the English text using `spacy`, which we chose due to its performance and accuracy. Even though documentation often contains snippets and direct references to code, we did not notice any errors relating to this.

We tokenized the Python text using the `tokenize` module in the standard library. We discarded the token attributes and only kept the corresponding text. We replaced spaces in indentation and string literals with the HTML escape code `%20` so that these would not be split afterward. When writing to the final text file, we used the `unicode_escape` encoding to ensure that tabs and newlines would be represented by `\t` and `\n`, respectively.

The average Python function had 110 tokens, while the average docstring had 50 tokens. This challenged our assumption that the two would have similar lengths. We used a maximum sequence length of 6,000 tokens and ignored the few examples that exceeded this limit

We prepared the text data for training using the `fairseq-preprocess` script, which builds dictionaries for each language and binarizes the corpora. The full vocabulary for Python contained 4.2 million tokens and the full vocabulary for English contained 1.3 million tokens. About 350 thousand tokens were shared between the two, mainly punctuation but also identifiers used to describe the code.

4.3.1 Limiting Vocabulary

In order to make the data more manageable, we chose to limit the vocabulary size for each language to 50 thousand words. To do this, we kept the top 50 thousand words by frequency and marked the rest as unknowns. As a result, about 8% of the Python tokens and 3% of the English tokens were replaced with unknowns. This means that in the Python corpus, 99% of the vocabulary accounted for just 8% of the tokens.

4.3.2 Byte-Pair Encoding

We also applied byte-pair encoding to the original text (with the full vocabulary) using the `subword-nmt` package. We learned the encoding over both languages jointly. Since we did not know what to expect in terms of the final vocabulary size, we used the default of 10 thousand. This worked very well as the percentage of tokens replaced with unknowns was reduced to 0% on the training set and 0.3% on the validation and testing sets.

Once again, this is smaller than the vocabularies used by experiments in natural language translation, which use vocabularies of 30-40 thousand BPE-encoded words. This difference is likely because Python code already resembles English.

5 Models

We had originally planned to test an LSTM-based Seq2Seq model [12], a transformer model [13], and [14], currently very close to the state of the art for machine translation. Given two versions of our dataset (limited vocabulary and BPE), we created six experiments to run. Ultimately, we wound up running three due to constraints on time and cloud credits.

We used Pytorch and the Fairseq extension [11], which is comparable to Tensorflow with the Tensor2Tensor package. The choice was personal, since we had more experience with Pytorch. Additionally, Fairseq supports a wider range of model architectures than Tensor2Tensor. These packages also have builtin support for features such as distributed and mixed precision training, which would be very useful if we were to scale up for future work.

We ran all models on a Tesla V100 GPU on Google Cloud, then on a GTX 1080 after we ran out of credits. We chose the V100 to take advantage of its AI specific features such as tensor cores and half precision floating point support. We ran into precision issues when using half precision, so we reverted to 32-bit.

For expedience, we trained our models for ten epochs initially. We then continued to train the transformer model with BPE for ten more epochs. We used a batch size of 32 sentences, which may have been inefficient on the V100 with 16 GB of video memory.

The training script produced logs every 100 batches, which we used to visualize training progress. We evaluated the model on the validation set and saved checkpoint every 10,000 batches and at the end of each epoch. To reduce file size, we provide only the last and best (by validation loss) checkpoints.

We used the Adam optimizer with the recommended default parameters [7] ($\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$) since these typically work well without further tuning. We did not use any learning rate scheduler or annealing. This is in contrast to both Vaswani et al. [13] and Wu et al. [14], who used Adam with custom learning rate schedulers and slightly different hyperparameters. These differences could be partly responsible for our poor results, as explained in the next section.

Using BPE had a huge effect on the training process. Not only did it improve the results, it reduced the size of the models by 75% and the VRAM usage during training by 30%.

5.1 Seq2Seq

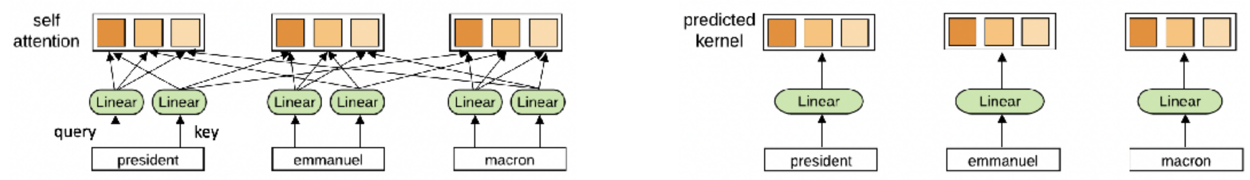
We tried to use an LSTM Seq2Seq model [12] as a baseline, even though we expected transformers to be better. However, the sequential nature of the model meant that it took a very long time to train (4 hours per epoch compared to over 1 hour for the transformer). We got bored and stopped it.

Our configuration used bidirectional LSTMs with 2 layers each in the encoder and decoder. We used a 256-dimensional embedding and 256 hidden nodes.

5.2 Transformer

We used the transformer model as described by Vaswani et al. [13], albeit with a smaller size. We used 2 layers in the encoder and decoder, embedding dimension of 256, and hidden layers with 512 nodes. This model took about 75 minutes to train each epoch.

We also tried training the transformer with the specification described in the original paper, but abandoned this model after it became clear that we wouldn't be able to train it for a reasonable amount of time.



(a) Self-attention (b) Dynamic convolution

Figure 2: Comparison of attention and dynamic convolutions. Image taken from Wu et al. [14]

5.3 DynamicConv

In addition to the transformer model, we also used DynamicConv [14], which is close to the state of the art for machine translation [8]. We chose this over the current best method, backtranslation, since we believed it would be a better fit for a one-way translation task. Backtranslation is discussed in more detail in section 7.3.2.

This model resembles transformers with attention, but substitutes the fully connected self-attention layer with a convolutional layer. This is illustrated in figure 2. Furthermore, the kernels are different for each time step. The authors claim this is more computationally efficient than attention and works better on long sequences. This is important for our usage since our sequences are much longer than natural language sentences.

Once again, we used a significantly reduced model for our experiments. Due to the advantage provided by BPE over simply limiting the vocabulary, we did not train on the limited vocabulary corpus. The model contains 2 encoder and decoder layers, with 256 dimension encodings and 512 nodes in each hidden layer. We used convolution kernel sizes of 3 and 7 in two layers.

This produced a model that was similar to the transformer in size and training time. This architecture is more complex than the transformer and would have benefited from more exploration of the possible parameters.

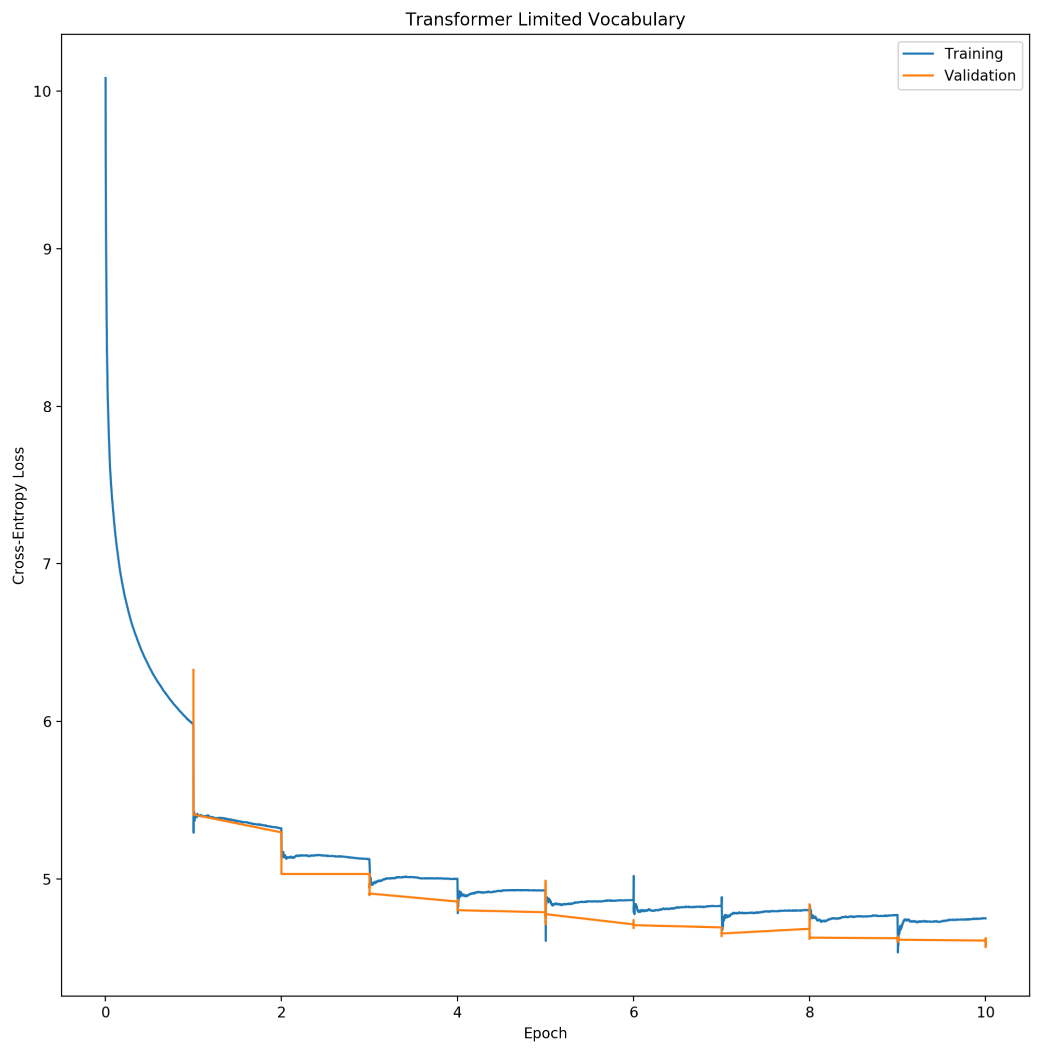
6 Results

Figure 3 shows the training progress of our models. There was an issue with the training transformer BPE model in the cloud that caused training to start from scratch, hence the large jump after epoch 15.

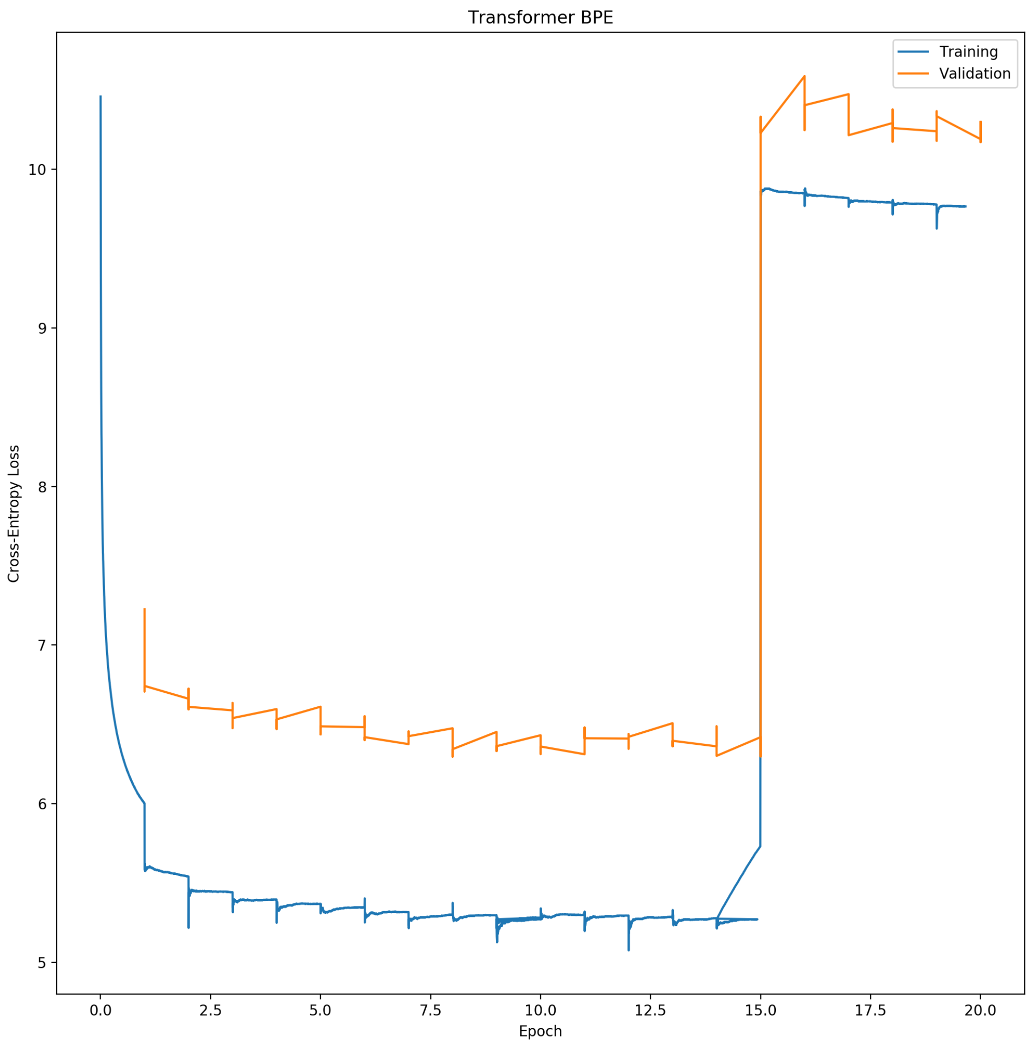
Other than this glitch, both training and validation losses are still decreasing, and more training is needed to improve the results.

Due to the sizes of our models and the limitations on training, we were not able to obtain good translations. The transformer model obtained BLEU scores of 1 and 3 with the limited vocabulary and BPE, respectively. The DynamicConv model obtained a score of 2. Needless to say, these are terrible. The baseline provided by Barone and Sennrich [1] achieves a BLEU score of nearly 14. While this is much higher than ours, it is also much lower than natural

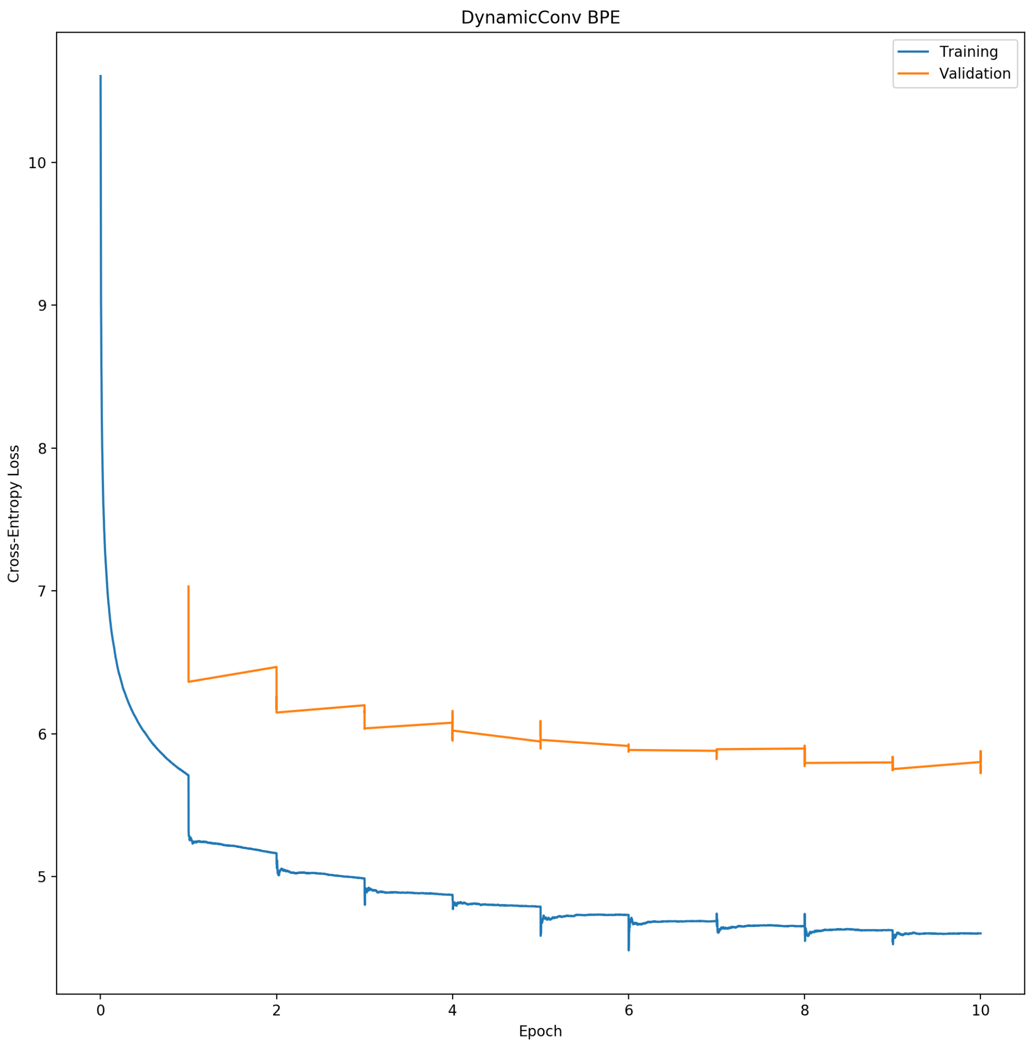
Figure 3: Training and validation losses over time



(a) Losses for transformer with limited vocabulary



(b) Losses for transformer with BPE vocabulary



(c) Losses for DynamicConv with BPE vocabulary language translation results, which are currently around 45.

Despite this, the produced text resembles valid English. For example, given a signature `popitem()`, the ground truth documentation is “Pop the item with key k from the dictionary and return it as a two-tuple (k, v). If k doesn’t exist, raise a KeyError.”. The generated text is “Return the number of bytes in bytes.”, which is grammatically correct, but bears no resemblance to the original.

7 Future Directions

Due to the constraints and challenges we faced over the quarter, we were unable to fully explore the problem at hand. To continue work in this area, we have identified several approaches to move forward.

7.1 Low Effort

The obvious continuation of our work is to invest more time and resources into training the models. As seen in the previous plots, we did not train the models to convergence due to a lack of computational resources. We had to limit our training in order to submit this project by the required dates, but we have plans to resume work after the quarter is over.

We could also use larger models that match the sizes used in the original literature. We used cut down versions for expedience, but these do not come close to the performance of the originals. We did try to train the Transformer-base model used by Vaswani et al. [13], but could not train it to a useful level with the resources we had. For comparison, the authors of all of the methods we considered used multiple GPUs and GPU servers for several days, which was well outside of our budget.

More thoroughly exploring different hyperparameters would also have resulted in better results. We explored a few different configurations of models and optimizers, but settled on our three configuration

7.2 Medium Effort

We also started implementing several ideas to improve our work which we did not have time to complete:

7.2.1 Java Corpus

In addition to our Python corpus, we also considered using a similar Java dataset. This corpus is larger, with about 4 million files. We chose to use Python due to the ease of parsing. We found out after the fact about the Eclipse Java Development Tools (JDT) used by some of the other groups, which fill a similar purpose to Python’s `ast` and `tokenize` modules.

The rigid structure of Java programs means that we expect to see more consitency in the size of functions. Cursory inspection also shows that the Java programs are better documented than the Python programs, and the structure of the documentation is more consistent between projects due to the Javadoc standard. The quality of the Python documentation varied from single sentences to detailed descriptions of function inputs, outputs, and inner workings.

Since Python uses whitespace to represent blocks, we kept those characters in our vocabulary represented as escaped `\n` and `%20`. As a result, many of the most frequent words in our vocabulary are just strings of spaces. Using curly braces, while harder to read by humans, would be more compact and reduce the size of our vocabulary.

Given these factors, and Java's imprtance in enterprise software development, exploring this data would be a logical next step.

7.2.2 Using Syntax Trees

Previous work in the field has shown significant improvements when incorporating syntactic information from the abstract syntax tree. This can come in the form of tree-based recurrent networks such as those used by Chen, Liu, and Song [2], or using a serialized S-expression representation used by Hu et al. [4]. The latter could be tokenized and used with our existing models.

We also noticed that many of the most common tokens were punctuation such as parentheses, newlines, and comments. We kept these since they include valuable syntactic information, but an AST representation would convey the same information with fewer extraneous words.

7.2.3 Enforcing Summarization

When designing our experiments, we assumed from experience that good documentation would be similar in length to the original code. However, this assumption did not hold for our dataset, since the average docstring was only a third of the size of the average function by number of tokens.

In addtion, the Python-English translation model may not be a good fit since not every part of the code is represented in the documentation and vice versa.

7.2.4 Subword Tokenization

Since we left the identifiers and literals untouched in our tokenization, our vocabulary was very large since most of these were unique. When limiting the vocabulary, we found that 99% of our vocabulary made up just 8% of the tokens in our corpus. This also means that it would be very hard to gain any information from the text of these tokens without the ability to learn the contexts in which they appear.

To resolve this, we could split the identifiers by camel and snake cases, and tokenize the string literals regularly. Implementing the latter would actually require less work that our current scheme, where we explicitly prevented strings from being escaped based on spaces. Splitting identifiers would allow the translator to group similar names, such as various errors and exceptions.

The BPE scheme we used is not aware of camel case and snake case, and the two schemes could be used together for maximum effect.

7.3 High Effort

These are potential long-term future directions for this research that we did not fully consider:

7.3.1 Multilingual Models

One emerging trend in machine translation is the use of multilingual models. These have been used by entities such as Google to efficiently translate between multiple languages, even when there is little training data for a given pair [6]. These models exploit the latent similarities between human languages to approximate something resembling a universal meta-language.

Given that the syntactic and semantic similarities between computer languages is greater than those between human languages, a model like this could make sense if applied here. For example, C/C++, Java, Javascript, and C# are syntactically similar enough that a translation model would benefit from the larger training corpus.

Additionally, most programming languages of interest share a common set of operations such as assingments, conditional branching, and iteration. If we had a way to identify such similarities in syntax trees, we could expand this model to include languages as dissimilar as C and Python.

7.3.2 Bidirectional Translation

Perhaps the holy grail of AI-assisted software engineering would be a system that translates natural language into code. This is the reverse of the problem attempted here, but is probably much harder to solve.

The current state of the art for bilingual machine translation is backtranslation [3], which uses a unidirectional model to generate translation pairs from a large corpus of plain text. For example, given a small English-German bilingual corpus and a large English monolingual corpus, the authors train a translation model on the bilingual corpus. The trained model is then used to generate a large German corpus from the provided English corpus. The model is then trained on the synthetic corpus, which yield better results than simply using the bilingual corpus.

Our corpus was smaller than common machine translation corpora and would probably have benefited from this data augmentation. One challenge would be finding a sufficiently large English corpus which can be used to produce program outlines.

7.3.3 Different Translation Units

In our attempt, we tried to translate to generate documentation for functions, since they are well-defined syntactically and have corresponding documentation. However, adding support for different translation units would allow us to provide different levels of information to users. For example, translating individual lines of code could be used by programmers to quickly explain a tricky part of a program, while translating entire libraries could be used for high level search and indexing features.

There has also been some emerging work in translating entire documents rather than individual sentences [15][10]. These methods augment the sequence-to-sequence translation models by adding additional contextual information to the source and target embeddings.

This would be especially useful for functions, where only a small part of the function body may be important and noteworthy. Adding additional context would help the translator skip over the boilerplate.

8 Code Explanations

Section	Path
4.1	data_utils/get_repolist.py, data_utils/downloaded_repos.py
4.2	data_utils/build_dataset.py, data_utils/clean_dataset.py
4.3	data_utils/data_to_text.py, data/data_raw/, data/full_vocab/
4.3.1	data/limited_vocab/
4.3.2	data/bpe_corpus_raw/, data/bpe/
5	models/train_models.py, models/checkpoints/

Table 1: Correspondences between project code and this report

References

[1] Antonio Valerio Miceli Barone and Rico Sennrich. "A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation". In: (July 7, 2017). arXiv: `1707.02275 [cs]`. URL: `http://arxiv.org/abs/1707.02275` (visited on 06/10/2019).

[2] Xinyun Chen, Chang Liu, and Dawn Song. "Tree-to-Tree Neural Networks for Program Translation". In: (Feb. 10, 2018). arXiv: `1802.03691 [cs]`. URL: `http://arxiv.org/abs/1802.03691` (visited on 11/17/2018).

[3] Sergey Edunov et al. "Understanding Back-Translation at Scale". In: (Aug. 28, 2018). arXiv: `1808.09381 [cs]`. URL: `http://arxiv.org/abs/1808.09381` (visited on 05/05/2019).

[4] Xing Hu et al. "Deep Code Comment Generation". In: *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*. The 26th Conference. Gothenburg, Sweden: ACM Press, 2018, pp. 200–210. ISBN: 978-1-4503-5714-2. DOI: `10.1145/3196321.3196334`. URL: `http://dl.acm.org/citation.cfm?doid=3196321.3196334` (visited on 06/04/2019).

[5] Srinivasan Iyer et al. "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)*. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Berlin, Germany: Association for Computational Linguistics, 2016, pp. 2073–2083. DOI: `10.18653/v1/P16-1195`. URL: `http://aclweb.org/anthology/P16-1195` (visited on 06/04/2019).

[6] Melvin Johnson et al. "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation". In: *Transactions of the Association for Computational Linguistics* 5 (Dec. 2017), pp. 339–351. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00065. URL: https://www.mitpressjournals.org/doi/abs/10.1162/tacl_a_00065 (visited on 06/04/2019).

[7] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (Dec. 22, 2014). arXiv: 1412.6980 [cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 10/27/2018).

[8] *Machine Translation*. URL: http://nlpprogress.com/english/machine_translation.html (visited on 06/09/2019).

[9] Vadim Markovtsev and Waren Long. "Public Git Archive: A Big Code Dataset for All". In: *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18* (2018), pp. 34–37. DOI: 10.1145/3196398.3196464. arXiv: 1803.10144. URL: http://arxiv.org/abs/1803.10144 (visited on 06/05/2019).

[10] Sameen Maruf and Gholamreza Haffari. "Document Context Neural Machine Translation with Memory Networks". In: (), p. 10.

[11] Myle Ott et al. "Fairseq: A Fast, Extensible Toolkit for Sequence Modeling". In: *Proceedings of NAACL-HLT 2019: Demonstrations*. 2019.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: (), p. 9.

[13] Ashish Vaswani et al. "Attention Is All You Need". In: (June 12, 2017). arXiv: 1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 05/05/2019).

[14] Felix Wu et al. "Pay Less Attention with Lightweight and Dynamic Convolutions". In: (Jan. 29, 2019). arXiv: 1901.10430 [cs]. URL: http://arxiv.org/abs/1901.10430 (visited on 06/04/2019).

[15] Jiacheng Zhang et al. "Improving the Transformer Translation Model with Document-Level Context". In: (), p. 10.