Code2Func: Function Name Generation from Source Code

Fabrice Harel-Canada
UCLA
fabricehc@g.ucla.edu

Jeya Vikranth Jeyakumar
UCLA
vikranth94@g.ucla.edu

Cibi Pragadeesh Kaliappa Arjun
UCLA
cibi.pragadeesh@gmail.com

ABSTRACT

The ability to automatically summarize or caption target sequences has an extensive history within natural language processing (NLP) research. However, such applications to computer programming languages are less well studied. We propose to approach the code summarization problem by reframing it into a translation task. Doing so allows us to employ a mixture of techniques developed in the neural machine translation (NMT) subfield to produce concise and reasonably accurate natural language outputs for a given source code snippet as input. We constrain the lengths of the summaries sufficiently to regard them as function names and thus present our model as Code2Func. We demonstrate the effectiveness of our approach for the task of function name prediction within Java and Python. Our code, data, and trained models are available at https://github.com/fabriceyhc/code2func.

INTRODUCTION

Code captioning is the process of generating a natural language title or brief summary for a given snippet of source code. Software developers regularly perform this task while familiarizing themselves with new code and when generating function names or documentation for their own code. In a broader sense, any task that involves understanding code necessarily entails an antecedent mental captioning of its purpose and we view the automation of this first step as an important time-saving contribution to the development process.

Previous work in this area can be categorized into two main paradigms: models that account for the differences between natural languages and programming languages [1, 2, 3, 4, 5, 6] and those that do not. Most sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), frame the code captioning task as a machine translation problem and treat source code like any other natural language - as a sequence of tokens. Such models were not specifically designed for handling programming languages, but have nonetheless been able to achieve state-of-the-art results. In contrast, approaches that leverage the syntactic structure of programming languages in the preprocessing and encoding portions of their pipelines to speed up training and improve the quality of the captions.

We present an alternative approach, Code2Func that synthesizes elements of both paradigms to generate code captions that are both concise and reasonably accurate. We demonstrate the effectiveness of our

approach on a dataset of the statically typed Java programming language [6] as well as a dataset of the dynamically typed Python programming language [7].

APPROACH

Our first step in accounting for the particularities of captioning code snippets is to represent them as a set of compositional paths sampled from its abstract syntax tree (AST). The root and leaves of an AST usually refer to user-defined values that represent identifiers and names from the code while the internal nodes represent some operator of the language like `for` loops, `if` statements, and `var` declarations. The first and last node of an AST path are terminals on all such nodes to increase the degree of correspondence with natural language. For example,whose values are tokenized as part of preprocessing. Furthermore, we recognize that many user-defined nodes employ stylized naming conventions wherein a small number of natural language words are concatenated to name a code variable or method. Therefore, we perform an additional "sub-tokenization" step the single token `createNewFile` would become three separate tokens `create, new,` and `file`.

The next step is to embed the tokenized AST paths into a fixed-length vector. The primary goal with this step is to create an informative input suitable for use with a traditional NMT model. For our study, we elected to use a transformer model [8] because of its strong performance on standard machine translation tasks. Additionally, the transformer's additional encoding layers provide an opportunity to learn a deeper representation of the AST paths. During decoding, the transformer attends over a different weighted average of the path embeddings to produce each output token, much like NMT models attend over token representations in the input sentence.

EVALUATION

In this exploration of the subject, we focus on one particular type of captioning task known as function name prediction, which we view as a special, more narrow kind of source code summarization, and is generally limited to the generation of short (1-5) output tokens (English words) for a given source code input. We treat this output as a prediction of a function name from its body. Quantitatively, we measure the number of exact string matches between the original function name and predicted one and report the accuracy accordingly. However, this metric is ultimately shallow in that a developer could reasonably choose many different function names and merely checking for an exact match does not preclude the possibility that our model discovered another reasonable name for the function. Therefore, to supplement the exact match accuracy metric, we used a combination of qualitative analysis on a select few examples and attempt another quantitative measure of evaluating the error between the actual name and predicted name using the **Damerau–Levenshtein distance** of the plain text strings. Damerau–Levenshtein distance is a string metric for measuring the edit distance between two sequences. It is the minimum number of operations (consisting of insertions, deletions or substitutions, or transpositions) required to change one sequence into the other.

We experiment with this task across two different datasets. The first dataset was produced by Barone et al. [7] and contains extracted Python 2.7 functions names and their code bodies. To facilitate comparisons with the original benchmarks, we used the same training/ validation/ test partitions as the original authors,

which consists of 109,108 training examples, 2,000 validation examples, and 2,000 test examples. The second dataset was produced by Alon et al. [6] and consists of 9,500 top-starred Java projects from GitHub, where 9,000 projects we used for training, 250 for validation, and 300 for testing - in all ~16 million examples. It is important to note that all inference is performed on code from projects not included in the training or validation sets to minimize the possibility of "leakage" between sets.

RESULTS

Our model scored an exact match accuracy of 12.3% in the Java dataset and 1.2% in the Python dataset. Further, we obtain the Damerau–Levenshtein distance vs number of test samples graph for both Java and Python datasets. In absolute counts, approximately 3,800 test samples have been predicted accurately (all string tokens match) in the Java dataset and 9 samples have been predicted correctly in the Python dataset.

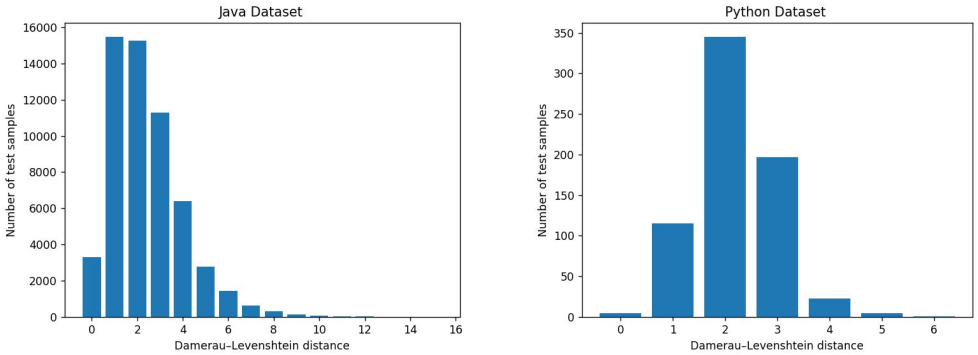

Figure 3. Graphs showing the Damerau–Levenshtein distance for Java (left) and Python (right).

Tokenized Java AST	True	Predicted
`Override Nm0 MarkerExpr Mth Prim1 longoverride Nm0 MarkerExpr Mth Nm2 METHOD_NAMElong Prim1 Mth Nm2 METHOD_NAMElong Prim1 Mth Bk Ret Nm0 timestampMETHOD_NAME Nm2 Mth Bk Ret Nm0`	`get timestamp`	`get timestamp`

Table 1. Example of a tokenized Java AST and the true / predicted function names.

Tokenized Python AST	True	Predicted
`key module body functiondef name bucket args arguments args name id things ctx param name id key ctx param vararg... ret ctx load decorator list`	`bucket`	`bucket name`

Table 2. Example of a tokenized Python AST and the true / predicted function names.

We observe that our Code2Func method performs a good job in predicting the function names for the Java dataset with the majority of the samples having Damerau– Levenshtein distance of two or fewer. Its performance is less impressive with the Python dataset and we believe this is for two reasons. First, the

training set is significantly smaller than the Java dataset and NMT models like the transformer require extremely large training corpora. This could be rectified by increasing the size of the Python dataset. The second reason is that Python is a dynamically typed language, which obscures the types in our static analysis environment and provided less information to the model.

When considering the exact name matching metric, our model's accuracy may seem very poor on the first pass. However, in a normal classification task, the baseline accuracy of random guessing is defined as $1/C^n$ where C is the number of possible classification labels and n is the number of output tokens. In our case, C is the set of all words in the English language, or at least the training set vocabulary tailored for some specific task, and is usually in the thousands if not tens of thousands. So our model achieves an accuracy of 12.3% in the Java dataset and 1.2% in the Python dataset actually performs much better than random guessing of function names. Ultimately, this metric is a shallow one to use because it doesn't account for the semantic similarity between words. It is quite possible to not get a match but to still have output a useful function name. So in addition to the quantitative metrics, we performed an admittedly small scale analysis of the predicted function names and both the input and the true method name. We find that in many cases, the model does predict a function name which is a close match to the true function name. As an example, 'bucket_name' was the predicted function name for the actual function called 'bucket' as seen in Table 2. We believe this should be considered a reasonable output.

DISCUSSION

One major goal of this effort that unfortunately did not materialize in time was the incorporation of BERT [9] to obtain contextual embeddings of the code before feeding it to the transformer model. To our knowledge, no other research has produced a contextualized word embedding for a programming language. Ultimately, we failed to produce a working BERT embedding due to the massive vocabulary of code tokens (>273k) and a lack of sufficient computing power. In future work, we will investigate different methods to tokenize code which can help in reducing the vocabulary size thereby making it easier to train contextual embeddings for different coding languages. We also plan to reduce the verbosity of the AST encoded functions and extract only the contents of the tree nodes, effectively stripping out the metadata tags added by the library, which likely contributed unnecessary noise to the inputs in our experiments.

CONCLUSION

In this paper, we formulate the code captioning problem as an NMT task and propose a method to automatically name functions in different programming languages. We construct the Code2Func model generates a function name given the code defining the function. We explain in detail the data preprocessing methods to convert the raw code to AST and feeding it to the Transformer model to make the predictions. Finally, we do a comprehensive evaluation of Code2Func using Java and Python datasets. Our results indicate that given large volumes of training data, we can generate function names given the code with considerable accuracy. Code2Func shows degradation in performance for Python dataset but we discussed solutions on how the performance can be improved further.

REFERENCES

1. Iyer, Srinivasan & Konstas, Ioannis & Cheung, Alvin & Zettlemoyer, Luke. (2016). Summarizing Source Code using a Neural Attention Model. 2073-2083. 10.18653/v1/P16-1195.
2. J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "TASSAL: Autofolding for Source Code Summarization," 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), Austin, TX, 2016, pp. 649-652.
3. Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, 826-831.
4. Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension (ICPC '18). ACM, New York, NY, USA, 200-210.
5. Hu, Xing & Li, Ge & Xia, Xin & Lo, David & Lu, Shuai & Jin, Zhi. (2018). Summarizing Source Code with Transferred API Knowledge. 2269-2275. 10.24963/ijcai.2018/314.
6. Alon, Uri & Levy, Omer & Yahav, Eran. (2018). code2seq: Generating Sequences from Structured Representations of Code.
7. Miceli Barone, Antonio Valerio & Sennrich, Rico. (2017). A parallel corpus of Python functions and documentation strings for automated code documentation and code generation.
8. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000–6010, 2017.
9. Devlin, Jacob & Chang, Ming-Wei & Lee, Kenton & Toutanova, Kristina. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.