CodeCaption
A dataset for captioning data science code

Ioana Baldini
IBM Research
ioana@us.ibm.com

Kavitha Srinivas
IBM Research
kavitha.srinivas@ibm.com

Jiri Navratil
IBM Research
jiri@us.ibm.com

Abstract

Automatically understanding the semantics of source code opens up opportunities for creating sophisticated programming assistants that are much needed in today's world dominated by software. To encourage more work in this area, we introduce a dataset, along with several tasks, for "captioning" data science source code. The dataset comprises Python function code from major data science libraries. The corresponding "captions" are extracted from well-formed docstrings succinctly describing the functions. We propose three different datasets and related tasks as follows: (1) Given a pair *(Code, Caption)*, decide whether the *Caption* indeed describes the *Code*; (2) Given a triplet *(Code, Caption A, Caption B)*, choose the correct *Caption* matching the *Code*; and (3) Given source code of a function, generate the appropriate caption. We utilize adversarial techniques to render the datasets appropriately difficult, leaving sufficient headroom for improvement over baseline systems using state-of-the-art NLP models and summarization techniques. The datasets will be made available to the community as a benchmark to aid further research in the area of automated code semantics.

1 Introduction

The ability of automatically understanding the semantics of software code leads to software engineering tools capable of semantic code search, automated documentation, bug identification and even code generation. This observation spurred research in machine learning on software code. Although similarities exist between programming languages and natural language, there are substantial differences between the two, in particular, in the way programming tasks are encoded, thus rendering a straightforward application of machine learning to code challenging. Simply applying natural language processing techniques do not work out of the box, and significant innovation is needed to address these differences [3].

The goal of our work is to encourage development of such innovative techniques by creating and providing the community with well-defined tasks and benchmarks relevant to code semantics.

1.1 Existing tasks and corpora

Machine learning on code is a relatively new domain with few datasets. For a comprehensive review of existing machine learning approaches on code we refer the reader to a survey by [3]. Much of the work done so far has drawn on code from open repositories, selected on the basis of the number of stars on GitHub, for instance, or maturity/popularity of the project. Other datasets are built from StackOverflow discussions and code [13]. The tasks are varied in formulation and purpose. Some examples of such tasks include:

- code summarization [9, 5, 12] in which the task is to generate function names, short documentation or comments [16].

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), ML For Systems Workshop, Vancouver, Canada.

- class name, variable name or variable usage prediction [2, 17, 4] in which the task predicts variable names or misuse of variables, which can be used in debugging.
- code completion tasks [11, 18, 6] meant to assist with code construction.
- program synthesis [8, 21, 22, 10, 19, 20] which generates fragments of code or entire programs from some form of specification.

Despite tremendous work in this domain, not all datasets are made publicly available and no standard datasets and tasks have emerged yet. One dataset based on Python was recently introduced [15], but proved to be difficult to use due to duplications in the train/test set [1, 9].

For the datasets published previously, there are two main issues: (1) the quality of code and its documentation in most open source projects tends to vary considerably, creating too much variability and potentially affecting the quality of the datasets derived from code at large; (2) most of the tasks bridging natural language and code have focused on generative tasks, such as producing code summaries or method names, which are difficult to evaluate due to the subjectivity of the task [1]. The subjective nature of the generative tasks makes evaluation and assessment of progress difficult.

In this work, we take a systematic approach to creating a dataset with corresponding machine learning tasks that are easier to evaluate with the goal of advancing the automatic understanding of code semantics.

1.2 Dataset and task construction

Our starting point is the observation that code within libraries tends to be accompanied by higher quality documentation in terms of consistency and structure, as compared to code in the wild. Based on this observation, we adopt library code for the dataset generation. In particular, we focus on Python libraries in the data science domain because these libraries tend to have code that is shorter, more linear, with less branching, more self-contained and well-defined, compared to large general purpose codebases. We believe code from such a domain has a better chance of having the kind of structure needed to provide a good signal for machine learning (ML). We include code in popular data science libraries in Python such as scikit-learn, statsmodels, numpy, scipy, pandas.

While the data science code is well structured, its documentation tends to be long and detailed, often including descriptions of the mathematical underpinnings of the particular technique at hand. Any ML task that is geared towards generation of such descriptions is unlikely to be practical. In contrast, we took advantage of the docstring structure to extract a 'caption' for each function. In Python, this tends to be a short 1-2 sentence summary of the function's core characteristics appearing at the beginning of the Python docstring. Extracting all functions along with their captions results in a data pool we refer to as *CodeCaption*.

Based on the pool described above, we propose three tasks designed to fill the gaps outlined earlier. The first two tasks address the difficulties with subjectivity inherent in generating unconstrained text/code by providing a fixed set of alternatives to choose from. The first task predicts whether a caption belongs to a piece of code. It is a binary classification task detecting pairs *(Code, Caption)* that belong together versus those that do not. The "positive pairs" were extracted from library code, while the "negative pairs" were generated in an adversarial fashion to make the task sufficiently challenging. The second task is an easier variant of the first, where the system needs to decide which of two candidate captions belongs to the code. Both these tasks are binary classification tasks for which accuracy can simply help evaluate and compare different techniques. The third task involves text generation, similar to [9, 5, 12].

In addition to the datasets and the tasks definition, we provide baseline results using state-of-the-art models to establish that the proposed tasks are sufficiently challenging. For this, we used BERT [7] to create baseline models yielding a 73% classification accuracy on Task 1, and 80% classification accuracy on Task 2. For the generative Task 3, we include baselines using neural machine translation (NMT) (F1 of 0.3) and models based on recent work of Fernandes et al. on structured neural summarization [9] (F1 of 0.3), which was targeted specifically at code documentation generation.

[1]Although generative tasks such as code summarization and documentation generation are desirable from a pragmatic standpoint, they are subjective by design. Even if human performance were taken into account, it is hard to imagine the case of two developers choosing to name a method exactly the same way, or generate documentation that is exactly the same.

2 Methodology

To construct the dataset, we included code from the following popular Python data science library distributions: scikit-learn, statsmodels, numpy, scipy, pandas. All testing and tutorial code as well as functions with empty docstrings were excluded. Our final collection had 9851 functions with their corresponding docstrings. We partitioned this collection such that 80% was used for training and 20% for test. For models that required hyperparameter tuning, we used 10% from the training set as a development set [2]. It is important to note that, by design, there are no repetitions among samples. While some duplication in high-level functionality is possible across libraries, no duplication at the code level occurs. Duplication has rendered previous datasets hard to use [1].

As mentioned previously, the docstrings tend to be well structured, typically with a line or two in the beginning summarizing the functionality of the code. We extract this brief description from each docstring and refer to it as the *code caption*. The next sections describe the datasets and the associated tasks. For a sample of the dataset, we invite the reader to check the Appendix.

2.1 Task 1: Detecting matching code captions

The first task is to detect whether a caption belongs to a piece of code. It is a binary classification task detecting matching (i.e., positive) pairs *(Code, Caption)*. The positive pairs were extracted from library code, while the non-matching, negative pairs were generated in an adversarial fashion to make the task sufficiently challenging. To create the negative pairs, we followed an iterative procedure. For each sample, a random caption was drawn from the complement of the train (or test) set, yielding a dataset with one positive and one negative pair per code sample [3].

This (initial) dataset was used to train a classifier based on BERT [7]. We used the small, pre-trained configuration of BERT adopting default settings of the paraphrase identification task, because code captions are analogous to a paraphrase of code. A sequence length of 128 [4] was used.

This first BERT-based model, trained on random negative pairings, resulted in an accuracy of 93%. To increase the difficulty of the dataset, in the next iteration, we selected adversarial negative samples, i.e., samples that are competitive with the positive ones: for each function code, we randomly chose 100 captions; the generated pairs were fed to the first BERT-based classifier followed by a random selection of one (negative) caption that the classifier labeled as positive. This iteration was repeated for both train and test set resulting in a new train/test datasets.

The newly generated datasets were used again to train a classifier based on BERT as explained above. This time, the classifier reached an accuracy of 83%. We repeated the adversarial procedure one more time. In this iteration, we increased the initial sampling buffer to 500 captions to maintain sufficient numbers of incorrectly labeled (i.e., competitive) candidates. After the second adversarial iteration, the BERT-based classifier obtained 73% accuracy, which we deemed as providing sufficient headroom to declare this as the final dataset.

When training the BERT-based classifiers, we considered function code as a sequence of tokens. We applied simple tokenization to both the code and the captions. In our experimentation, a lot of the signal used by BERT was present in the function signature. We hypothesize that improving on the performance of this classification task will require exploiting the structured nature of code to further approximate its semantics.

2.2 Task 2: Select correct code caption out of two candidates

The second task determines the correct caption given a triplet with function code and two captions, one of which is the correct one. The dataset used for this task is derived from the previous dataset, using the negative captions created in the previous dataset. As in the previous task, we used BERT to generate a baseline model. For this classifier, same BERT paraphrase task configuration was

[2] Also referred to as validation set

[3] The pairs are always formed drawing on the corresponding partition, never crossing the train and test set boundary

[4] In our experimentation, we find that increasing the sequence length does not increase the performance significantly (within 1-2%), while the training time increases significantly. We posit that this is due to the fact that a lot of the signal is in the function signature which is captured in a shorter length sequence.

Dataset Size			
Task	**Train**	**Dev**	**Test**
Detect Matching Caption	15776	-	3946
Two-Choice Pick	7888	-	1973
Generate Caption	7099	789	1973

Vocabulary	
Code vocabulary size	26654
Caption vocabulary size	5450
Out-of-vocabulary rate	3.3%

Table 1: Dataset statistics.

Task	BERT accuracy on Test
Detect Matching Caption	73.1%
Two-Choice Pick	80.2%

Table 2: Performance of BERT-based classifiers.

employed, feeding the model two pairs of (Code, Caption) and training the classifier to select which one is the correct pairing.

The BERT model's accuracy was 80%. Performing iterations, as described in Section 2.1 above, gained no further increase in difficulty.

2.3 Task 3: Generating caption text, given code

Generating code captions is a much more difficult task and, due to the fact that different wordings can have similar semantics, this task is harder to evaluate. However, we included a generative task of producing captions for completeness and due to the popularity of such tasks [9, 5, 12]. Thus, given a function source code, an algorithm is expected to produce 1-2 sentence description in natural language, which then can be compared to the positive caption. For this task, we used the initial train/dev/test partitions. The baselines based on OpenNMT [14] and structured neural summarization [9] reached .3 F1 score, leaving considerable room for improvement.

3 Dataset and baseline results

3.1 Dataset statistics

Table 1 shows basic statistics for the three tasks. The first two tasks (Detect Match and Two-Choice Pick) come without a designated Development (Dev) partition as we did not perform hyperparameter tuning. Note that for the first task, there is an equal number of positive and negative captions. For the caption generation, we randomly split the training set further into training and dev set.

3.2 Baseline results

As described in Section 2, for the first two tasks, we built models based on BERT whose resulting accuracies are summarized in Table 2. We initialized BERT with weights pretrained on the language modeling task, then fine-tuned the model as a binary classifier using the training partition. As expected, the Two-Choice Pick task is somewhat easier resulting in an accuracy of 80.2%, compared to the Detection task yielding an accuracy of 73.1%.

For the generative task, we built two different models: The first based on the traditional neural machine translation (NMT). For the implementation we used the OpenNMT [14] library. No tuning of the model was performed relying on parameters that proved effective in traditional NMT settings [14]. The second baseline model adopted recent work on structural neural summarization [9](SNS). For this model, we performed grid-based hyperparameter search. Our task was similar in nature to the *MethodDoc* task from Fernandes et al, however the sizes of the datasets and the language used are different. Our dataset was smaller in size and built from Python code. Both models achieved similar results for ROUGE-1 F score, while the SNS model had a lower ROUGE-2 F score, leaving considerable room for improvement.

Model	ROUGE-1 F score	ROUGE-2 F score
NMT	0.31	0.19
SNS	0.30	0.13

Table 3: Performance of caption generation models.

4 Conclusion

We introduced a dataset called *CodeCaption* which comprises two classification tasks and one generative task, connecting function source code to their documentation. We focused on data science libraries to ensure good code/documentation quality. We believe that *CodeCaption* will be a useful resource to study machine learning techniques for various applications, such as semantic code understanding, documentation generation, learning with limited data, and language modeling for code. Furthermore, due to its limited size but focused domain, the *CodeCaption* dataset can be useful in studying aspects of transfer learning and domain adaptation in programming code.

References

[1] M. Allamanis. The adverse effects of code duplication in machine learning models of code. *CoRR*, 2018.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 2015.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 2018.

[4] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[5] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, 2016.

[6] P. Bielik, V. Raychev, and M. Vechev. Phog: Probabilistic model for code. In *International Conference on Machine Learning*, 2016.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

[8] K. Ellis and S. Gulwani. Learning to learn programs from examples: Going beyond program structure. In *International Joint Conference on Artificial Intelligence*, 2017.

[9] P. Fernandes, M. Allamanis, and M. Brockschmidt. Structured neural summarization. In *International Conference on Learning Representations*, 2019.

[10] C. Finegan-Dollak, J. K. Kummerfeld, L. Zhang, K. Ramanathan, S. Sadasivam, R. Zhang, and D. Radev. Improving text-to-SQL evaluation methodology. In *Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018.

[11] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, 2012.

[12] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin. Summarizing source code with transferred api knowledge. In *International Joint Conference on Artificial Intelligence, IJCAI-18*, 2018.

[13] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Annual Meeting of the Association for Computational Linguistics*, 2016.

[14] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Association for Computational Linguistics 2017, System Demonstrations*, 2017.

[15] A. V. Miceli Barone and R. Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *International Joint Conference on Natural Language Processing*, 2017.

[16] D. Movshovitz-Attias and W. W Cohen. Natural language models for predicting programming comments. In *Annual Meeting of the Association for Computational Linguistics*, volume 2, 2013.

[17] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Symposium on Principles of Programming Languages*, 2015.

[18] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Conference on Programming Language Design and Implementation*, 2014.

[19] X. Xu, C. Liu, and D. Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. In *International Conference on Learning Representations*, 2017.

[20] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017.

[21] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Conference on Empirical Methods in Natural Language Processing*, 2018.

[22] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, 2017.

A Supplemental Material

We present here just a few samples from the dataset to illustrate the type of tasks we constructed. Our intention was to submit the dataset as supplementary material, however the submission website did not seem to allow for such an option.

A.1 Task 1: Determine true captions

Code:

```
def get_pkg_info(pkgname, dirs=None):
    from numpy.distutils.npy_pkg_config import read_config
    if dirs:
        dirs.append(get_npy_pkg_dir())
    else:
        dirs = [get_npy_pkg_dir()]
    return read_config(pkgname, dirs)\n"
```

Positive Caption

```
Return library info for the given package.
```

Note that the caption contains the word *library* which does not appear in the actual code.
Code:

```
def get_pkg_info(pkgname, dirs=None):
    from numpy.distutils.npy_pkg_config import read_config
    if dirs:
        dirs.append(get_npy_pkg_dir())
    else:
        dirs = [get_npy_pkg_dir()]
    return read_config(pkgname, dirs)\n"
```

Negative Caption

```
If static or shared libraries are available
then return their info dictionary.
```

Even the negative caption contain the word library. For a human, the fact that the code does not seem to contain any information on *static* or *shared* is a signal that this caption has a lower chance of being the correct one.

A.2 Task 2: Select correct code caption out of two candidates

Code:

```
def _warn_if_deprecated(key):
    d = _get_deprecated_option(key)
    if d:
        if d.msg:
            print(d.msg)
            warnings.warn(d.msg, FutureWarning)
        else:
            msg = "'{key}'␣is␣deprecated".format(key=key)
            if d.removal_ver:
                msg += ('␣and␣will␣be␣removed␣in␣{version}'
                        .format(version=d.removal_ver))
            if d.rkey:
                msg += ",␣please␣use␣'{rkey}'␣instead."
                        .format(rkey=d.rkey)
            else:
                msg += ',␣please␣refrain␣from␣using␣it.'

            warnings.warn(msg, FutureWarning)
        return True
    return False
```

Caption A (positive):

```
Checks if `key` is a deprecated option and if so, prints a warning.
```

Caption B (negative):

```
if key id deprecated and a replacement key defined, will return the
replacement key, otherwise returns `key` as-is
```

A.3 Task 3: Generating caption text, given code

Code:

```python
def update_tr_radius(Delta, actual_reduction, predicted_reduction,
                     step_norm, bound_hit):
    if predicted_reduction > 0:
        ratio = actual_reduction / predicted_reduction
    elif predicted_reduction == actual_reduction == 0:
        ratio = 1
    else:
        ratio = 0

    if ratio < 0.25:
        Delta = 0.25 * step_norm
    elif ratio > 0.75 and bound_hit:
        Delta *= 2.0

    return Delta, ratio
```

Caption:

```
Update the radius of a trust region based on the cost reduction.
```

Note that the concept *trust region* does not appear in the actual code and it would be hard, if not impossible, to be generated automatically. The function name contains a shortcut for it in the form of *tr*.