Recommendations for Datasets for Source Code Summarization

Alex LeClair, Collin McMillan
Department of Computer Science and Engineering
University of Notre Dame
{aleclair, cmc}@nd.edu

Abstract

Source Code Summarization is the task of writing short, natural language descriptions of source code. The main use for these descriptions is in software documentation e.g. the one-sentence Java method descriptions in JavaDocs. Code summarization is rapidly becoming a popular research problem, but progress is restrained due to a lack of suitable datasets. In addition, a lack of community standards for creating datasets leads to confusing and unreproducible research results – we observe swings in performance of more than 33% due only to changes in dataset design. In this paper, we make recommendations for these standards from experimental results. We release a dataset based on prior work of over 2.1m pairs of Java methods and one sentence method descriptions from over 28k Java projects. We describe the dataset and point out key differences from natural language data, to guide and support future researchers.

1 Introduction

Source Code Summarization is the task of writing short, natural language descriptions of source code (Eddy et al., 2013). The most common use for these descriptions is in software documentation, such as the summaries of Java methods in JavaDocs (Kramer, 1999). Automatic generation of code summaries is a rapidly-expanding research area at the crossroads of Computational Linguistics and Software Engineering, as a growing tally of new workshops and NSF-sponsored meetings have recognized (Cohen and Devanbu, 2018; Quirk, 2015). The reason, in a nutshell, is that the vast majority of code summarization techniques are adaptations of techniques originally designed to solve NLP problems.

A major barrier to ongoing research is a lack of standardized datasets. In many NLP tasks such as Machine Translation there are large, curated datasets (e.g. Europarl (Koehn, 2018)) used by several research groups. The benefit of these standardized datasets is twofold: First, scientists are able to evaluate new techniques using the same test conditions as older techniques. And second, the datasets tend to conform to community customs of best practice, which avoids errors during evaluation. These benefits are generally not yet available to code summarization researchers; while large, public code repositories do exist, most research projects must parse and process these repositories on their own, leading to significant differences on one project to another. The result is that research progress is slowed as reproducibilty of earlier results is difficult.

Inevitably, differences in dataset creation also occur that can mislead researchers and over or understate the performance of some techniques. For example, a recent source code summarization paper reports achieving 25 BLEU when generating English descriptions of Java methods with an existing technique (Gu et al., 2018), which is 5 points higher than the original paper reports (Iyer et al., 2016). The paper also reports 35+ BLEU for a vanilla seq2seq NMT model, which is 16 points higher than what we are able to replicate. While it is not our intent to single out any one paper, we do wish to call attention to a problem in the research area generally: a lack of standard datasets leads to results that are difficult to interpret and replicate.

In this paper, we propose a set of guidelines for building datasets for source code summarization techniques. We support our guidelines with related literature or experimentation where strong literary consensus is not available. We also compute several metrics related to word usage to guide future researchers who use the dataset. We have made a dataset of over 2.1m Java methods and summaries from over 28k Java projects available via an online appendix (URL in Section 6).

Proceedings of NAACL-HLT 2019, pages 3931–3937
Minneapolis, Minnesota, June 2 - June 7, 2019.

2 Related Work

Related work to this paper consists of approaches for source code summarization. As with many research areas, data-driven AI-based approaches have superseded heuristic/template-based techniques, though overall the field is quite new. Work by Haiduc *et al.* (Haiduc et al., 2010a,b) in 2010 coined the term "source code summarization", and several heuristic/template-based techniques followed including work by Sridhara *et al.* (Sridhara et al., 2010, 2011), McBurney *et al.* (McBurney and McMillan, 2016), and Rodeghero *et al.* (Rodeghero et al., 2015).

More recent techniques are data-driven, though the overall size of the field is small. Literature includes work by Hu *et al.* (Hu et al., 2018a,b) and Iyer *et al.* (Iyer et al., 2016). Projects targeting problems similar to code summarization have been published widely, including on commit message generation (Jiang et al., 2017; Loyola et al., 2017), method name generation (Allamanis et al., 2016), pseudocode generation (Oda et al., 2015), and code search (Gu et al., 2018). Nazar *et al.* (Nazar et al., 2016) provide a survey.

Of note is that no standard datasets for code summarization have yet been published. Each of the above papers takes an ad hoc approach, in which the authors download large repositories of code and apply their own preprocessing. There are few standard practices, leading to major differences in the reported results in different papers, as discussed in the previous section. For example, the works by LeClair *et al.* (LeClair and McMillan, 2019) and Hu *et al.* (Hu et al., 2018a) both modify the CODENN model from Iyer *et al.* (Iyer et al., 2016) to work on Java methods and comments. LeClair *et al.* and Hu *et al.* report very disparate results: A BLEU-4 score of 6.3 for CODENN on one dataset, and 25.3 on another, even though both datasets were generated from Java source code repositories.

These disparate results happen for a variety of reasons, such as a difference in data set sizes and tokenization schemes. LeClair *et al.* use a data set of 2.1 million Java method-comment pairs while Hu *et al.* use a total of 69,708. Hu *et al.* also replace out of vocabulary (OOV) tokens in the comments with <UNK> in the training, validation, and testing sets, while LeClair *et al.* remove OOV tokens from the training set only.

3 Dataset Preparations

The dataset we use in this paper is based on the dataset provided by LeClair *et al.* (LeClair and McMillan, 2019) in a pre-release. We used this dataset because it is both the largest and most recent in source code summarization. That dataset has its origins in the Sourcerer project by Lopes *et al.* (Lopes et al., 2010), which includes over 51 million Java methods. LeClair *et al.* provided the dataset after minimal initial processing that filtered for Java methods with JavaDoc comments in English, and removed methods over 100 words long and comments >13 and <3 words. The result is a dataset of 2.1m Java methods and associated comments. LeClair *et al.* do additional processing, but do not quantify the effects of their decisions – this is a problem because other researchers would not know which of the decisions to follow. We explore the following research questions to help provide guidelines and justifications for our design decisions in creating the dataset.

3.1 Research Questions

Our research objective and contribution in this paper is to quantify the effect of key dataset processing configurations, with the aim to make recommendations on which configurations should be used. We ask the following Research Questions:

RQ$_1$ What is the effect of splitting by method versus splitting by project?

RQ$_2$ What is the effect of removing automatically generated Java methods?

The scope of the dataset in this paper is source code summarization of Java methods – the dataset contains pairs of Java methods and JavaDoc descriptions of those methods. However, we believe these RQs will provide guidance for similar datasets e.g. C/C++ functions and descriptions, or other units of granularity e.g. code snippets instead of methods/functions.

The rationale behind RQ$_1$ is that many papers split the dataset into training, validation, and test sets at the unit of granularity under study. For example, dividing all Java methods in the dataset into 80% in training, 10% in validation, and 10% in testing. However, this results in a situation where it is possible for code from one project to be in both the testing set and the training set. It is possible that similar vocabulary and code patterns

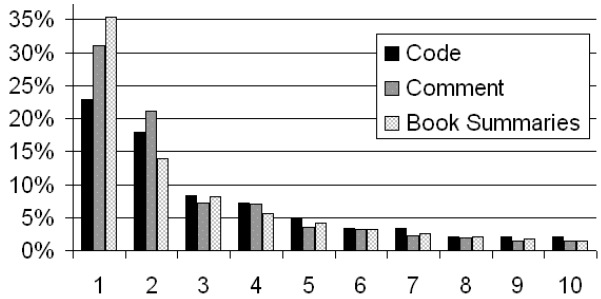

Figure 1: Word count histogram for code, comment, and the book summaries. About 22% of words occur one time across all Java methods, versus 35% in the book summaries.

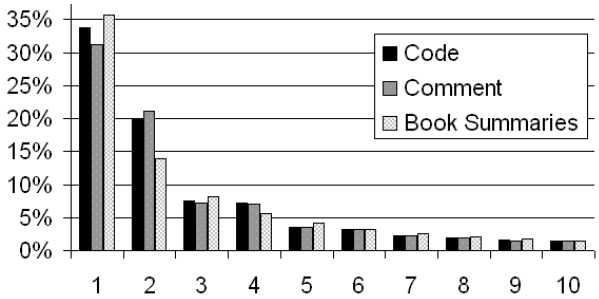

Figure 2: Histogram of word occurrences per document. Approximately 34% of words occur in only one Java method, 20% occur in two methods, etc.

are used in methods from the same project, and even worse, it is possible that overloaded methods appear in both the training and test sets. However, this possibility is theoretical and a negative effect has never been shown. In contrast, we split by project: randomly divide the Java projects into training/validation/test groups, then place all methods from e.g. test projects into the test set.

The rationale behind RQ_2 is that automatically generated code is common in many Java projects (Shimonaka et al., 2016), and that it is possible that very similar code is generated for projects in the training set and the testing/validation sets. Shimonaka *et al.* (Shimonaka et al., 2016) point out that the typical approach for identifying auto-generated code is a simple case-insensitive text search for the phrase "generated by" in the comments of the Java files. LeClair *et al.* (LeClair and McMillan, 2019) report that this search turns out to be quite aggressive, catching nearly all auto-generated code in the repository. However, as with RQ_1, the effect of this filter is theoretical and has not been measured in practice.

3.2 Methodology

Our methodology for answering RQ_1 is to compare the results of a standard NMT algorithm with the dataset split by project, to the results of the same algorithm on the same dataset, except with the dataset split by function. But because random splits could be "lucky", we created four random datasets split by project, and four split by function, seen in Table 2. We then use an off-the-shelf, standard NMT technique called `attendgru` provided pre-release by LeClair *et al.* (LeClair and McMillan, 2019) and used as a baseline approach in their recent paper. The technique is just an attentional encoder/decoder based on single-layer GRUs, and represents a strong NMT baseline used by many papers. We train `attendgru` with each of the four training sets, find the best-performing model using the validation set associated with that training set (out of 10 maximum epochs), and then obtain test performance for that model. We report the average of the results over the four random splits. Note that we used the same configuration for `attendgru` as LeClair *et al.* report, except that we reduced the output vocabulary to 10k to reduce model size.

Our process for RQ_2 is similar. We created four random split-by-project sets in which automatically generated code was *not* removed. Then we compared them to the four random split-by-project sets we created for RQ_1 (in which auto-generated code was removed).

3.3 Dataset Characteristics

We make three observations about the dataset that, in our view, are likely to affect how researchers design source code summarization algorithms. First, as depicted in Figure 1, words appear to be used more often in code as compared to natural language – there are fewer words used only one or two times, and in general more used 3+ times. At the same time (Figure 2), the pattern for word occurrences per document appears similar, implying that even though words in code are repeated, they are repeated often in the same method and not across methods. Even though this may suggest that the occurrence of unique words in source code is isolated enough to have little affect on BLEU score, we show in Section 4 that this word overlap causes BLEU score inflation when you split by function. This is important because the typical MT use case assumes that a "dictionary" can be created (e.g., via attention) to map words in a source to words in a target language. An algorithm applied to code summarization needs to tolerate multiple occurrences of the same words. To compare the source code, comments, and natural language datasets we tokenized our data by removing all special characters, lower casing, and for source code – splitting camel case into separate tokens.

SplittingStrategy	Set1	Set2	Set3	Set4
Split by project	17.81	16.73	17.11	17.99
Split by function	20.97	23.74	23.67	23.68
Auto-generated code included	19.11	19.09	18.04	15.66

Table 1: Average BLEU Scores from 15 epochs for each of the four sets.

A related observation is that Java methods tend to be much longer than comments (Figure 3 areas (c) and (d)). Typically, code summarization tools take inspiration from NMT algorithms designed for cases of similar encoder/decoder sequence length. Many algorithms such as recurrent networks are sensitive to sequence length, and may not be optimal off-the-shelf.

A third observation is that the words in methods and comments tend to overlap, but in fact a vast majority of words are different (70% of words in code summary comments do not occur in the code method, see Figure 3 area (b)). This situation makes the code summarization problem quite difficult because the words in the comments represent high level concepts, while the words in the source code represent low level implementation details – a situation known as the "concept assignment problem" (Biggerstaff et al., 1993). A code summarization algorithm cannot only learn a word dictionary as it might in a typical NMT setting, or select summarizing words from the method for a summary as a natural language summarization tool might. A code summarization algorithm must learn to identify concepts from code details, and assign high level terms to those concepts.

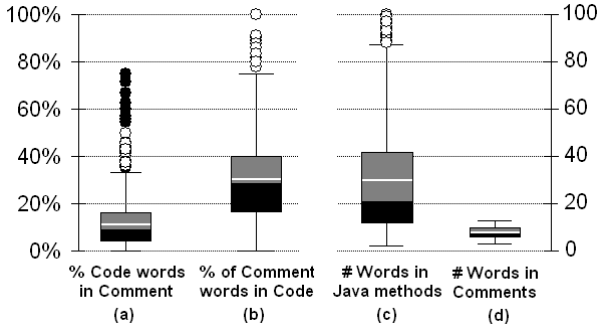

Figure 3: Overlap of words between methods and comments (areas a and b). Over 30% of words in comments, on average also occur in the method it describes. About 11% of words in code, on average, also occur in the comment describing it. Also, word length of methods and comments (areas c and d). Methods average around 30 words, versus 10 for comments.

4 Experimental Results & Conclusion

In this section, we answer our Research Questions and provide supporting evidence and rational.

4.1 RQ_1: Splitting Strategy

We observe a large "false" boost in BLEU score when split by function instead of split by project (see Figure 4). We consider this boost false because it involves placing functions from projects in the test set into the training set – an unrealistic scenario. An average of four runs when split by project was 17.41 BLEU, a result relatively consistent across the splits (maximum was 18.28 BLEU, minimum 16.10). In contrast, when split by function, the average BLEU score was 23.02, and increase of nearly one third as seen in Table 1. Our conclusion is that splitting by function is to be avoided during dataset creation for source code summarization. Beyond this narrow answer to the RQ, in general, any leakage of information from test set projects into the training or validation sets ought to be strongly avoided, even if the unit of granularity is smaller than a whole project. We reiterate from Section 1 that this is not a theoretical problem: many papers published using data-driven techniques for code summarization and other research problems split their data at the level of granularity under study.

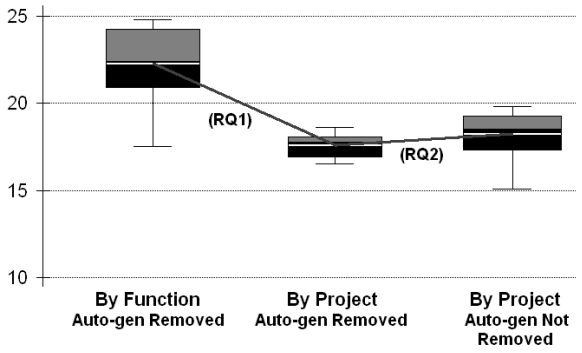

Figure 4: Boxplots of BLEU scores from attendgru for four runs under configurations for RQ_1 and RQ_2.

	BP Set1	BP Set2	BP Set3	BP Set4	BF All Sets
Training Set	1,935,860	1,950,026	1,942,291	1,933,677	1,943,723
Validation Set	105,693	100,920	104,837	105,997	107,984
Testing Set	107,568	98,175	101,993	109,447	107,984

Table 2: Number of method-comment pairs in the train, validation, test sets used in each random split set when split by project (BP) and by function (BF).

4.2 RQ_2: Removing Autogen. Code

We also found a boost in BLEU score when not removing automatically generated code, though the difference was less than observed for RQ_1. The baseline performance increased to 18 BLEU when not removing auto-generated code, and it varied much more depending on the split (some projects have much more auto-generated code than others). Our recommendation is that, in general, reasonable precautions should be implemented to remove auto-generated code from the dataset because we do find evidence that auto-generated code can affect the results of experiments.

5 Discussion

This paper provides benefits to researchers in the field of automatic source code summarization in two areas. First, we provide insight into the effects of splitting a Java method and comment dataset by project or by function, and how these different splitting methods effect the task of source code summarization. Second, we provide a dataset of 2.1m pairs of Java methods and one sentence method descriptions in a cleaned and tokenized format (discussed in 6) as well as a training, validation, testing split.

Note however that there may be cases where researchers wish to adapt our recommendations for a specific context. For example, when generating comments in an IDE. The problem of code summarization in an IDE is slightly different than what we have presented, and would benefit from including code-comment pairs from the same project. IDEs have the advantage of access to a programmer's source code and edit history in real time – they do not rely on a repository collected post-hoc. Moreno *et al.* (Moreno et al., 2013) take advantage of this information to generate Java class summaries in an eclipse plugin – their tool uses both the class and project level information from completed projects to generate these summaries, while not using any information from outside sources.

However, even in this case, care must be taken to avoid unrealistic scenarios, such as ensuring that the training set consists only of code older than the code in the test set. For example, consider a programmer at revision 75 of his or her project who requests automatically generated comments from the IDE, then goes on to write a total of 100 revisions for the project. An experiment simulating this situation should only use revisions 1-74 as training data – revisions 76+ are "in the future" from the perspective of the real world situation.

6 Downloadable Dataset

In our online appendix we have made three downloadable sets available. The first is our SQL database, generated using the tool from McMillan *et al.* (McMillan et al., 2011), that contains the file name, method comment, and start/end lines for each method, we call this dataset our "Raw Dataset". We also provide a link to the Sourcerer dataset (Linstead et al., 2009) which is used as a base for the dataset in LeClair *et al.* (LeClair and McMillan, 2019). In addition to the Raw Dataset, we also provide a "Filtered Dataset" that consists of a set of 2.1m method comment pairs. In the Filtered Dataset we removed auto-generated source code files, as well all method's that do not have an associated comment. No preprocessing was applied to the source code and comment strings in the Filtered Dataset. The third downloadable set we supply is the "Tokenized Dataset". In the Tokenized Dataset, we processed the source code and comments from the Filtered Dataset identically to the tokenization scheme described in Section 5 of (LeClair and McMillan, 2019). This set also provides a training, validation, and test set as well as a script to easily reshuffle these sets.

The URL for download is:

http://leclair.tech/data/funcom

Acknowledgments

This work is supported in part by the NSF CCF-1452959, CCF-1717607, and CNS-1510329 grants. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors

References

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100.

Ted J Biggerstaff, Bharat G Mitbander, and Dallas Webster. 1993. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press.

William Cohen and Prem Devanbu. 2018. Workshop on nlp for software engineering.

Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 13–22. IEEE.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010a. Supporting program comprehension with source code summarization. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010b. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge. In *IJCAI*, pages 2269–2275.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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)*, volume 1, pages 2073–2083.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE Press.

Philipp Koehn. 2018. European parliament proceedings parallel corpus 1996-2011. `http://www.statmt.org/europarl/`. Accessed October 25, 2018.

Douglas Kramer. 1999. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM.

Alexander LeClair and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. *International Conference on Software Engineering (ICSE) (*under review*)*.

Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336. 10.1007/s10618-008-0118-x.

C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. 2010. UCI source code data sets.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856*.

Paul W McBurney and Collin McMillan. 2016. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120. ACM.

L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker. 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 230–232.

Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE.

Chris Quirk. 2015. Nsf interdisciplinary workshop on statistical nlp and software engineering. `http://www.languageandcode.org/nlse2015/`. Accessed October 25, 2016.

Paige Rodeghero, Cheng Liu, Paul W McBurney, and Collin McMillan. 2015. An eye-tracking study of java programmers and application to source code

summarization. *IEEE Transactions on Software Engineering*, 41(11):1038–1054.

Kento Shimonaka, Soichi Sumi, Yoshiki Higo, and Shinji Kusumoto. 2016. Identifying auto-generated code by using machine learning techniques. In *Empirical Software Engineering in Practice (IWESEP), 2016 7th International Workshop on*, pages 18–23. IEEE.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM.

Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 101–110. ACM.