Code Summarization

Chi Xiao

1 Brief Introduction

Source code summarization is the task of writing natural language descriptions of source code. Work in this document is to search and read relevant articles and perform code reproduction.

2 Documents

2.1 Background & Related Work

The term "source code summarization" was coined around 2009 by Haiduc et al.[2] for the task of generating short descriptions of source code. The word "summarization" referred to the underlying technologies borrowed from the Natural Language Processing research community used to summarize natural language documents. At the time, these were dominated by keyword extraction techniques, such as ranking the top-n words in a document using tf/idf or a similar metric.

However, this line of research was largely put on ice around 2017, with the introduction of neural models of source code and encoder-decoder architectures (e.g., seq2seq, graph2seq).[3] Fig.1 depicts this history. Column I in the figure groups techniques based on IR, manual feature design, and other heuristics. Column N refers to papers in which the underlying model is based on a neural architecture. Column G means the code is represented via graph or graph-like features such as the AST. Column T means the model is Transformer-based. Column C means the intellectual merit of the paper is in using the code context.

Between 2017 and 2019, many papers achieved big gains from the big data input. Their efforts were focused on how to pre-process the data for use in existing neural models. Since then, two complementary strategies have emerged to best improve performance of code summarization: 1) better models of the code itself, such as by Zügner et al. [4] and Liu et al. [5], and 2) models that include context information, such as by Haque et al. [6].

More recently, retrieval-based techniques have also been employed to use contextual information. In 2020, Wei et al. [7] introduce Re2Com, a technique to find similar functions in a database and use corresponding summaries as a secondary input to neural network. In 2021, Li et al. [8] introduced a technique

	I	N	G	T	C
McBurney (2016) [13]	x				x
Zhang *et al.* (2016) [14]	x				x
Iyer *et al.* (2016) [15]		x			
Rodeghero *et al.* (2017) [16]	x				x
Fowkes *et al.* (2017) [17]	x				
Badihi *et al.* (2017) [18]	x				
Loyola *et al.* (2017) [19]		x			
Lu *et al.* (2017) [20]		x			
Jiang *et al.* (2017) [21]		x			
Hu *et al.* (2018) [22]		x			
Hu *et al.* (2018) [23]		x	x		
Allamanis *et al.* (2018) [24]		x	x		
Wan *et al.* (2018) [25]		x			
Liang *et al.* (2018) [26]		x			
Alon *et al.* (2019) [27], [28]		x	x		
Gao *et al.* (2019) [29]		x			
LeClair *et al.* (2019) [30]		x	x		
Nie *et al.* (2019) [31]		x			
Haldar *et al.* (2020) [32]		x			
Ahmad *et al.* (2020) [33]		x		x	
Haque *et al.* (2020) [34]		x			x
Zügner *et al.* (2021) [35]		x	x		
Liu *et al.* (2021) [36]		x	x		
Bansal *et al.* (2021) [12]		x			x

Figure 1: History

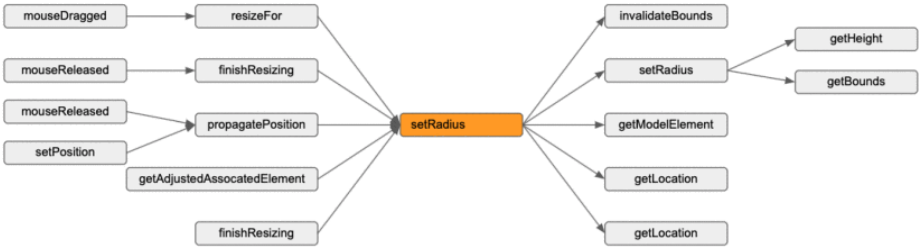

Figure 2: call context example

to retrieve summaries of similar functions and used them as a template. They proposed a module to edit these summaries with new information from the target function. These approaches are important, given the wide re-use of source code in online repository.

2.2 Function Call Graph Context Encoding for Neural Source Code Summarization[1]

This research is based on the author's previous research, [9] and it is also the research that I focus on reading and plan to conduct code reproduction.

The innovation of this paper is mainly reflected in the unique Call Context model and the perfect neural network structure for combining the Function Call Graph.

2.2.1 Modeling Call Context

In this paper, they define the call context of a subroutine as the nodes that fall within two edges from the subroutine in the program call graph. This scope includes the callers of a subroutine and that caller's callers. Plus, it includes the functions that a subroutine calls, plus the functions those functions call. Their definition of call context is in line with related work, which has repeatedly shown that human programmers almost always find the information they need within two edges in the function call graph [10], [11], [12], [13], [14].

Consider the example call context in Fig. 2. The function setRadius() is the target, and is part of its own call context. To build the rest of the call context, they take the first b functions that setRadius() calls. Then, they take the first b calls that those functions make. These functions are the "right side" of the call context in Fig. 2. Then to make the "left side", they add a maximum of b functions in which setRadius() is within the first b calls. Then they add the b functions that call each of those functions. The maximum number of functions in the call context is then $2 * (b^2 + b) + 1$.

This scope, while "only" encompassing two hops in the call graph, turns out to cover an average of 8% of a typical program in their subset. For example, in their dataset of 190k Java methods, projects have a median of 170 methods, and the mean call context of a method includes about 14 methods.

2.2.2 Neural Model

The heart of their prediction model is a graph neural network (GNN) that creates a vectorized representation of the functions in the call context. They use a recurrent neural network (RNN) to create a vector representation for the initial state of each function in the call context. Then They use a GNN to propagate information among these functions based on their function calls. They combine this call context information with information from a standard encoder-decoder model to predict a summary for the function.

An overview of the neural model underpinning our approach is in Fig. 3. In general, their model is based on an encoder-decoder architecture like most approaches to neural code summarization. What is novel is that they add components to the encoder to help the model learn from call graph context. The gray components in Fig. 3 (area 1) indicate a standard encoder-decoder model in which the encoder's input is the source code of the function and the decoder learns to represent the summaries. This encoder-decoder model is the foundation of almost all neural source code summarization techniques, and they continue to use it in their approach.

2.2.3 Other Works

They evaluated different configurations of their approach against several baselines in a quantitative experiment, followed by a comparison to the reference summaries in a qualitative experiment. In the quantitative experiment, they showed that their approach improves over the baselines in a large dataset. They also showed that their approach improves automated summaries over other models for a niche set.

In the qualitative experiment, they show that participants found their summaries reasonably accurate, readable, and concise. However, a majority of them found both generated and reference summaries incomplete.

2.2.4 Further Work Needed

Broader impacts of this paper include suggestions for future work implied by their experimental results. First, they advise that future work explore different subsets of their dataset where one approach excels over another. They recommend ensembles as a way to combine improvements for future work. Second, they found that metric scores do not correlate with each other in terms of improvements. They recommend future work to use multiple metrics, especially metrics such as METEOR in favor of the standard BLEU metric. Third, the qualitative experiment suggests studying the accuracy of the underlying reference examples.

3 Code Reproduction

Not yet inplemented.

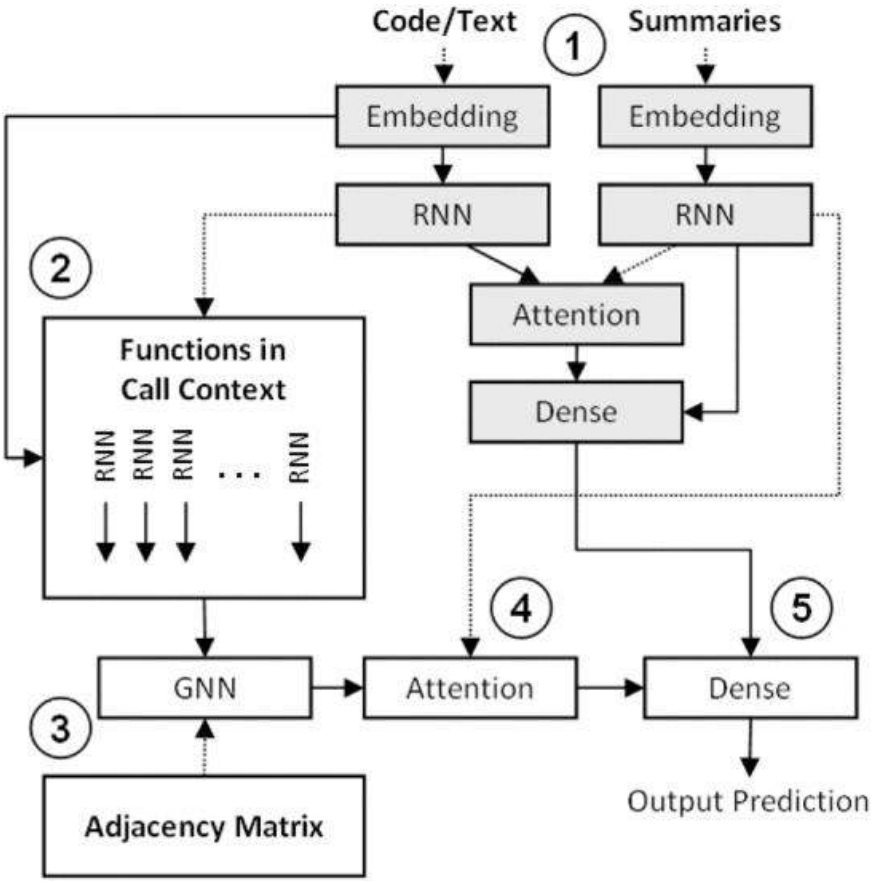

Figure 3: model

References

[1] Aakash Bansal, Zachary Eberhart, Zachary Karas, Yu Huang, Collin McMillan, "Function Call Graph Context Encoding for Neural Source Code Summarization", IEEE Transactions on Software Engineering, vol.49, no.9, pp.4268-4281, 2023.

[2] S. Haiduc, J. Aponte, L. Moreno and A. Marcus, "On the use of automated text summarization techniques for summarizing source code", Proc. IEEE 17th Work. Conf. Reverse Eng., pp. 35-44, 2010.

[3] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, "A survey of machine learning for big code and naturalness", ACM Comput. Surveys, vol. 51, no. 4, pp. 1-37, 2018.

[4] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec and S. Günnemann, "Language-agnostic representation learning of source code from structure and context", Proc. Int. Conf. Learn. Representations, 2021.

[5] S. Liu, Y. Chen, X. Xie, J. K. Siow and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN", Proc. Int. Conf. Learn. Representations, 2021.

[6] S. Haque, A. LeClair, L. Wu and C. McMillan, "Improved automatic summarization of subroutines via attention to file context", Int. Conf. Mining Softw. Repositories, 2020.

[7] B. Wei, Y. Li, G. Li, X. Xia and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation", Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng., pp. 349-360, 2020.

[8] J. Li, Y. Li, G. Li, X. Hu, X. Xia and Z. Jin, "EditSum: A retrieve-and-edit framework for source code summarization", Proc. IEEE/ACM 36th Int. Conf. Automated Softw. Eng., pp. 155-166, 2021.

[9] A. Bansal, S. Haque and C. McMillan, "Project-level encoding for neural source code summarization of subroutines", Proc. IEEE/ACM 29th Int. Conf. Prog. Comprehension, pp. 253-264, 2021.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie and C. Fu, "Portfolio: Finding relevant functions and their usage", Proc. 33 rd Int. Conf. Softw. Eng., pp. 111-120, 2011.

[11] E. Hill, L. Pollock and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance", Proc. IEEE/ACM 22nd Int. Conf. Automated Softw. Eng., pp. 14-23, 2007.

[12] M. Eaddy, A. V. Aho, G. Antoniol and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval dynamic analysis and program analysis", Proc. IEEE 16th Int. Conf. Prog. Comprehension, pp. 53-62, 2008.

[13] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector and S. D. Fleming, "How programmers debug revisited: An information foraging theory perspective", IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 197-215, Feb. 2010.

[14] T. D. LaToza and B. A. Myers, "Developers ask reachability questions", Proc. IEEE/ACM 32nd Int. Conf. Softw. Eng., pp. 185-194, 2010.