Meta Learning for Code Summarization

Anonymous ACL submission

Abstract

Source code summarization is the task of generating a high-level natural language description for a segment of programming language code. Current neural models for the task differ in their architecture and the aspects of code they consider. In this paper, we show that three state-of-the-art models for code summarization work well on largely disjoint subsets of a large code base. This complementarity motivates model combination: We propose three meta-models that select the best candidate summary for a given code segment. The two neural models improve significantly over the performance of the best individual model, obtaining an improvement of 2.1 BLEU points on a dataset of code segments where at least one of the individual models obtains a non-zero BLEU.

1 Introduction

Source code summarization is the task of generating short natural language statements describing a segment of code (Haiduc et al., 2010; Sridhara et al., 2010). Such summaries serve an integral role in software development by aiding code comprehension (Takang et al., 1996; Xia et al., 2018). The recent availability of large code bases and advances in machine learning have given this task significant attention at the interface between NLP and software engineering. Most neural network-based approaches build on machine translation (MT) strategies, framing code summarization as a text-to-text generation task (Richardson et al., 2017).

A first interesting parallel to MT research in NLP is that code summarization models also differ substantially in their assumptions about the nature of the task. Some adopt a sequence-to-sequence mapping approach (Iyer et al., 2016; Eberhart et al., 2020), while others take into account code structure, e.g., abstract syntax trees (ASTs) (Hu et al., 2018a; Wan et al., 2018; LeClair et al., 2019), or infer latent structure with graph neural networks (LeClair et al., 2020) or transformers (Ahmad et al., 2020). Another active direction, again similar to many NLP tasks, is the inclusion of contextual and background information, through API calls (Hu et al., 2018b), information from other methods or projects (Haque et al., 2020; Bansal et al., 2021), or exploiting the symmetry between code summarization and generation (Wei et al., 2019).

In this paper, we follow up on the observation by LeClair et al. (2019) that current models perform well for some examples. An analysis on three state-of-the-art methods (*NeuralCodeSum*, *ast-attendgru*, *attendgru*, cf. Sec. 2.1) on the Funcom dataset (Sec. 3) shows that the models are indeed largely *complementary* (cf. Figure 1): Each of the individual models creates the best summary for a substantial number of code segments, with the best model *NeuralCodeSum*, winning in about 6.4k of 22k cases where any model predicts a summary with non-zero BLEU. Table 1 illustrates this complementarity on two short methods: even though all models learn cues from code identifiers (here, method and variable names), in most cases they are only partially successful, and no single model is always best.

Based on these observations, we propose to combine the strengths of the individual code summarization models with meta learning (Naik and Mammone, 1992), training a new model that selects the best summary, given the original code segment and candidate summaries. We find a statistically significant improvement over the best individual models.

2 Methods

Given a sequence $T = (t_1, ..., t_l)$ of code tokens, code summarization is the task to produce a sequence $S = (w_1, ..., w_k)$ of words describing the code. The predictions are evaluated against reference summaries, using BLEU score (Papineni et al., 2002) as also customary in MT.

Code	Source	Summary	BLEU
	Reference	gets the value of the helpful votes property	
`public BigInteger getHelpfulVotes(){ return helpfulVotes; }`	*NeuralCodeSum*	gets the value of the helpful votes property	1.00
	attendgru	gets the value of the reason votes property	0.59
	ast-attendgru	gets the value of the reason type property	0.54
	Reference	determines whether to display the last button in the bottom pane	
`public void displayLastButton(boolean b) { bottomPane.lastButton.setVisible(b); }`	*NeuralCodeSum*	display the last button	0.17
	attendgru	displays the last button	0.00
	ast-attendgru	display the last button in the panel	0.46

Table 1: Summaries predicted by three state-of-the-art code summarization models and BLEU score compared to a human-written reference.

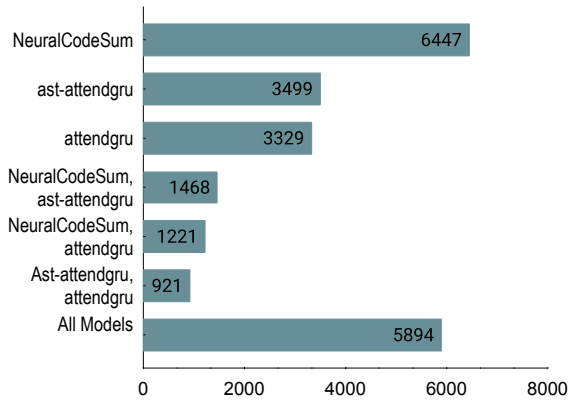

Figure 1: Complementarity of code summarization models: # of FunCom methods for which each model achieves highest BLEU score (, indicates draw).

2.1 Code Summarization Models

As sketched above, a number of code summarization models have been proposed in the literature. We consider three models. All use an encoder-decoder structure, and yield state-of-the-art results.

Text-based The *attendgru* model uses an LSTM as encoder to summarize the token sequence into a context vector (LeClair et al., 2019). The decoder then uses this vector to generate the summary.

Code structure-based The *ast-attendgru* model is an extension of *attendgru* (LeClair et al., 2019). In addition to the tokens, it also considers a flattened abstract syntax tree (AST). It encodes both inputs separately and feeds their concatenation into a decoder.

Transformer-based. The *NeuralCodeSum* model (Ahmad et al., 2020) uses a transformer with relative positional encoding and copy attention as encoder, and then predicts a summary with a decoder.

2.2 Meta-Learning Model

Given the complementarity of these models (cf. Figure 1), it would be very desirable to combine their strengths. There are multiple ways to do so. Straightforward combination of model output, as usual in ensembling (Rokach, 2010), is difficult for highly structured output such as summaries. LeClair et al. (2021) combine multiple source encoders with a joint decoder, which is effective but requires disassembling models. In this paper, we instead adopt a meta-learning approach (Naik and Mammone, 1992) in which we learn a *summary selector*. We formulate this task as *multi-label binary classification tasks*, where the meta-model predicts the suitability of each candidate summary, given the summary and the original code segment. We propose three such classifiers.

2.2.1 Feature-Based Meta-Model

Our first classifier, $meta_{\text{feat}}$, is a logistic regression model (Cox, 1958) whose features are designed to capture properties of code segments which may determine the difficulty of generating code summaries, building on ideas from performance prediction (Papay et al., 2020) and confidence estimation for summarization (Louis and Nenkova, 2009). We consider the following feature types:

Token and word frequencies Based on the frequency of each code token and each word across the codebase, we consider the harmonic means $\overline{freq}(T)$ and $\overline{freq}(S)$ of the code and summary, respectively. The hypothesis is that higher frequencies should make for simpler summarization.

Length We consider the number $|T|$ of tokens in a code segment and the number $|S|$ of words in a summary, with longer code segments and summaries indicating higher complexity and thus difficulty.

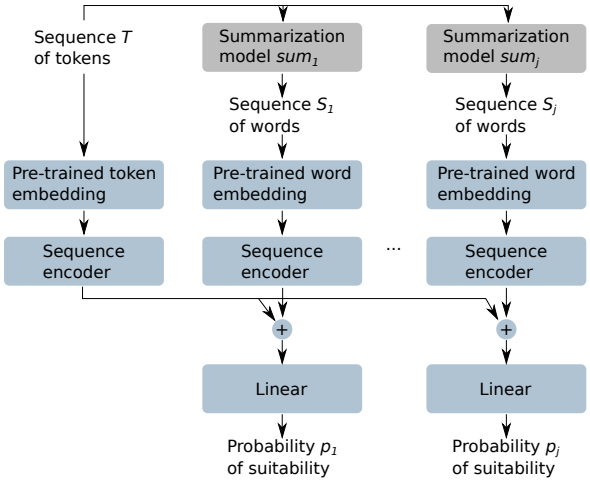

Figure 2: Architecture of our neural meta-models.

Distinctiveness We measure how distinctive a candidate summary I is compared to all summaries produced by the same model as the Kullback–Leibler divergence $Dis_{kl}(P_i||P)$, where P is the unigram distribution of all summaries, and P_i is the unigram distribution of candidate summary i. We expect low distinctiveness to lead to difficult summarization.

2.2.2 Neural Meta-Models

As an alternative to specifying the relevant features by hand, we define two neural meta-models that select a summary based on self-learned distributed features. More specifically, as shown in Figure 2, we first represent the input sequences (code tokens and summaries) in terms of FastText token and word embeddings, respectively, The choice of FastText is motivated by prior work showing that FastText outperforms other pre-trained token embedding models at accurately representing identifiers in source code (Wainakh et al., 2021). We pretrain these embeddings on the training dataset used in the evaluation (see below). After embedding, the model consists of two encoders, one for the code token sequence (generating a vector v_T) and one for each summary (generating a vector v_S).

The final step is to concatenate, for each summary, v_T and v_S. The concatenation is passed through two linear layers, and finally through a sigmoid function so that each summary is associated with a probability. The two sequence encoders and the linear layers are trained jointly.

Our two neural models differ only in the type of sequence encoder they use. The first model, called $meta_{\text{LSTM}}$, encodes sequences through a bi-directional LSTM. The other model, called $meta_{\text{TRN}}$, is based on a transformer.

2.2.3 Training and Querying Meta Models

As the goal of the meta-model is to maximize the overall BLEU score of the predicted summaries w.r.t. reference summaries, we train the meta-models in a supervised manner based on labels derived from BLEU scores. We label a summary as suitable if and only if it achieves the best BLEU score among all available candidate summaries. If multiple candidate summaries achieve the same, non-zero BLEU score, then all these candidates are labeled as suitable. Let B be the set of BLEU scores obtained by candidate summaries $S_1, ..., S_j$ for a code sequence T, then the training label for $T, (S_1, ..., S_j)$ is $p_1, ..., p_j$ where

$$p_i = \begin{cases} 1, & \text{if } BLEU(S_i, S_{ref}) = max(B) \\ 0, & \text{otherwise} \end{cases}$$

At inference time, we choose the candidate summary S_i with the highest predicted probability p_i.

3 Experimental Setup

Data. We use the *FunCom* dataset (LeClair and McMillan, 2019). It contains 2.1 million pairs of Java code segments and summaries, with an average of 51 tokens per segment and 15 words per summary. We use the authors' tokenization. As shown in Table 2, we divide the dataset into three partitions: for summary generation, for meta-learning, and for testing. The test partition corresponds to the one used in previous work (LeClair et al., 2019, 2020; Haque et al., 2020; LeClair et al., 2021), whereas the partition to train summarization models is smaller than in prior work, as we keep some data for the meta-model. Because for a substantial percentage of code segments, all summarization models fail to produce a summary with non-zero BLEU, we also consider a *filtered* dataset containing only segments where at least one summarization model achieves BLEU > 0. The filtered dataset hence are the cases where the meta-model has a chance to improve over the individual models.

Models and Evaluation. We first train the three code summarization models and then our meta-models, as defined in Section 2. We evaluate the summaries by the standard choice of corpus-level aggregated BLEU scores (Papineni et al., 2002). We consider three scenarios, which differ on whether the meta-model is trained on the

Partition	Split	All	Filtered
Summarization	train	1.4 million	NA
	valid	60k	NA
Meta	train	440k	101k
	valid	70k	5.3k
Test	test	101k	22k

Table 2: Statistics of the experiment datasets

	Model	Train/test of meta model		
		All/ all	All/ filtered	Filtered/ filtered
Summar.	*attendgru*	16.25	48.29	48.29
	ast-attendgru	16.62	49.35	49.35
	NeuralCodeSum	18.57	55.66	55.66
Meta	$meta_{\text{feat}}$	17.93	52.47	55.06
	$meta_{\text{LSTM}}$	18.94*	57.22*	**57.08***
	$meta_{\text{TRN}}$	**19.18***	**57.74***	56.94*

Table 3: BLEU scores on test set for individual summarization models and meta models. * indicates a statistically significant improvement over $NeuralCodeSum$ at α=0.05.

entire meta partition or only the filtered portion, and analogously whether it is evaluated on the full or the filtered portion of the test partition.

We make our code and data available. More information, along with hyperparameters, can be found in Appendix A.

4 Results

Table 3 shows our main results. We first consider the setup with training and test on the full dataset. Among the individual summarization models, the transformer-based *NeuralCodeSum* model works best, with a BLEU of 18.6. Both neural meta models improve over the individual models; the difference to *NeuralCodeSum* is statistically significant at α=0.05. The transformer-based meta model achieves the best result at 19.2 BLEU (+0.6 BLEU). In contrast, the feature-based meta model even underperforms the best individual code summary model. This highlights the difficulty of predicting the quality of summaries for code segments, while the quality of summaries for natural language texts has been predicted successfully (Louis and Nenkova, 2009).

If we evaluate the same models only on the filtered datasets – i.e., where the meta model has a chance of improving over the individual models (middle column) – we observe the same ranking of the models, but the margin between the best individual summarization model (*NeuralCodeSum*, 55.7 BLEU) and the neural meta learning models has increased: We obtain a BLEU of 57.2 for the $meta_{LSTM}$ (+1.5 BLEU) and a BLEU of 57.7 for the $meta_{TRN}$ (+2 BLEU); differences again are significant at $\alpha = 0.05$. We take these numbers as an indication that the neural meta-learning approach is generally successful for code segments for which "sensible" summary candidates (with BLEU > 0) have been produced by the individual models.

Finally, the right-hand column assesses the consequences of training the meta-models only on such "sensible" summary candidates exist. Compared to the middle column, the meta-model results are slightly lower.In other words, the apparently uninformative summary candidates still contribute to the success of the meta model. Taken together with the observation that the results for the BiLSTM changes much less (-0.1 BLEU) than for the transformer (-0.8 BLEU), we propose the following interpretation: Pairs of code segments and nonsensical summaries may still help the neural model in learning to encode typical code token and word sequences, which is more important for the transformer, with its higher capacity, than for the BiLSTM.

5 Conclusions

The present paper exploits the complementary nature of different code summarization models through a meta-learning approach. We find that neural models can predict the best summary from a set of candidates created by three state-of-the-art models, yielding an increase in BLEU of up to 2.1 points. We believe our results to be promising, and future improvements of individual summarization models will give our meta-models better predictions to choose from. At the same time, our results also highlight directions for future work, including meta-model introspection (why does the transformer succeed where the manual features fail?) and a re-evaluation of BLEU as summary evaluation metric (Fabbri et al., 2021).

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of ACL*.

Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *Proceedings of ICPC*, pages 253–264. IEEE.

David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232.

Zachary Eberhart, Alexander LeClair, and Collin McMillan. 2020. Automatically extracting subroutine summary descriptions from unstructured comments. In *2020 IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

Alexander R. Fabbri, Wojciech Kryściński, Bryan McCann, Caiming Xiong, Richard Socher, and Dragomir Radev. 2021. SummEval: Re-evaluating Summarization Evaluation. *Transactions of the Association for Computational Linguistics*, 9:391–409.

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

Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *2020 Mining Software Repositories (MSR)*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of ICPC*, page 200–210, New York, NY, USA.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge. In *Proceedings of 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 ACL*, pages 2073–2083, Berlin, Germany.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings ICLR*.

A. LeClair and C. McMillan. 2019. Recommendation for datasets for source code summarization. In *Proceedings of NAACL*.

Alexander LeClair, Aakash Bansal, and Collin McMillan. 2021. Ensemble models for neural source code summarization of subroutines. In *Proceedings of ICSME*.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of ICPC*.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of ICSE*, page 795–806. IEEE Press.

Annie Louis and Ani Nenkova. 2009. Performance confidence estimation for automatic summarization. In *Proceedings of EACL*, pages 541–548, Athens, Greece.

D.K. Naik and R.J. Mammone. 1992. Meta-neural networks that learn by learning. In *Proceedings of IJCNN*, volume 1, pages 437–442 vol.1.

Sean Papay, Roman Klinger, and Sebastian Padó. 2020. Dissecting span identification tasks with performance prediction. In *Proceedings of EMNLP*, page 4881–4895, Online.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of ACL*, page 311–318, USA. Association for Computational Linguistics.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Kyle Richardson, Sina Zarrieß, and Jonas Kuhn. 2017. The code2text challenge: Text generation in source libraries. *Proceedings of the 10th International Conference on Natural Language Generation*.

Lior Rokach. 2010. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1):1–39.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of ASE*.

Armstrong Takang, Penny Grubb, and Robert Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *J. Prog. Lang.*, 4:143–167.

Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. Idbench: Evaluating semantic representations of identifier names in source code. In *Proceedings of ICSE*, pages 562–573.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of ASE*.

Dataset	Model	Epoch	Training Time
Entire	$meta_{\text{TRN}}$	5	$\approx 30min/epoch$
	$meta_{\text{LSTM}}$	7	$\approx 15min/epoch$
	$meta_{\text{feat}}$	5000	$\approx 5min/model$
Filtered	$meta_{\text{TRN}}$	4	$\approx 10min/epoch$
	$meta_{\text{LSTM}}$	5	$\approx 3min/epoch$
	$meta_{\text{feat}}$	200	$\approx 2min/model$

Table 4: Details on number of epochs and training time.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Proceeedings of NeurIPS*.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976.

A Reproduciblity

Neural Meta Models We set the dimensionality for code embedding and summary embeddings as 100. The model are trained using the mini-batch size of 25 with Adam (Kingma and Ba, 2015) optimizer and learning rate as $1e-4$. Considering the average length of code and NL sequences we set the input size as 50 and 20 respectively. We computed training loss using Binary Cross Entropy loss method. The $meta_{\text{LSTM}}$ comprises of two layer Bi-LSTM encoder layers. Similarly the $meta_{\text{TRN}}$ has two neural attention head and two encoder layers. Details on individual layer dimensions are detailed in the accompanying code repository.

Feature Meta Models The $meta_{\text{feat}}$ model utilizes Logistic regression model from Scikit-lean (Pedregosa et al., 2011). For training on entire dataset we use 'saga' and on filtered subset 'liblinear' as solvers. Due to data imbalance on entire dataset we use class weights (False: 1 ,True: 5).

Code In the accompanying implementation set we provide, source code for meta models.Additionally, the repository contains separate `run.sh` scripts for executing an end-to-end cycle for training meta-learning models. Table 4 details information on approximate training times and number of epochs for individual models and dataset. For training summarization models we use publicly available implementations of the models.

Dataset For training summarization models we use the filtered dataset from Funcom repository. In the accompanying dataset we provide pre-trained word embeddings and curated files for training meta models. The dataset can be found at the following link.