Project-Level Encoding for Neural Source Code Summarization of Subroutines

Aakash Bansal, Sakib Haque, and Collin McMillan
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{abansal1, shaque, cmc}@nd.edu

***Abstract*—Source code summarization of a subroutine is the task of writing a short, natural language description of that subroutine. The description usually serves in documentation aimed at programmers, where even brief phrase (e.g. "compresses data to a zip file") can help readers rapidly comprehend what a subroutine does without resorting to reading the code itself. Techniques based on neural networks (and encoder-decoder model designs in particular) have established themselves as the state-of-the-art. Yet a problem widely recognized with these models is that they assume the information needed to create a summary is present within the code being summarized itself – an assumption which is at odds with program comprehension literature. Thus a current research frontier lies in the question of encoding source code context into neural models of summarization. In this paper, we present a project-level encoder to improve models of code summarization. By project-level, we mean that we create a vectorized representation of selected code files in a software project, and use that representation to augment the encoder of state-of-the-art neural code summarization techniques. We demonstrate how our encoder improves several existing models, and provide guidelines for maximizing improvement while controlling time and resource costs in model size.**

***Index Terms*—source code summarization, automatic documentation generation, neural networks**

I. Introduction

Source code summarization is the task of writing short, natural language descriptions of that code [1], [2]. Typical targets of summarization are the subroutines of a software project. The purpose of the descriptions is to provide human readers with a big picture view of what each subroutine does. Even a single phrase e.g. "compresses data to a zip file" can help a person understand code without having to read every detail of that code [3]. Summaries of subroutines form the foundation of much documentation aimed at programmers such as JavaDocs [4], and the literature is replete with studies demonstrating how programmers often rely on these summaries, only turning to reading the code itself as a last resort [5]. And while a majority of documentation is still written manually, recent research has made inroads towards automatic code summarization [6].

The backbone of almost all state-of-the-art approaches to automatic source code summarization is the neural encoder-decoder model architecture [7]–[9]. This architecture has its roots in machine translation [10], in which the encoder creates a vectorized representation of a sentence in one language (e.g. English), while the decoder creates a representation of that same sentence in a different language (e.g. French). When trained with enough data (on the order of millions of examples [11]), these models can learn to associate patterns in the encoder representation to patterns in the decoder representation. After training, the encoder can be given an input example, and the model can generate a likely decoder representation and therefore a likely output example – and translate French sentences to English. For source code, the encoder's job is to represent the source code, while the decoder represents the source code summary – give the encoder source code, and the decoder generates a summary.

The obvious problem with these approaches is that they can only generate a summary based on whatever source code is passed to the encoder. Thus these approaches make a tacit assumption that all of the information necessary to generate that summary is present in that source code. This assumption is at odds with decades of program comprehension literature [12]–[14]. This literature is quite clear that high-level descriptions such as summaries very often contain concepts that can only be understood in the context of the other code in the same software project. To paraphrase a classic example, a subroutine called `book()` can only be fully understood if it is also known that it exists in a class called `Seats` in a project called `AircraftTravel` [15].

In this paper, we present a project-level encoder to augment existing encoder-decoder neural models of source code summarization. Our approach is "project level" in that it creates a vectorized representation of a subset of code files in a software project. Our approach augments existing models in that the output of our approach may be combined with encoder portion of most existing code summarization models: most models contain an encoder for the source code itself that produces some vectorized representation of that code, and our encoder extends that representation. The advantage to our encoder is that it provides context to the model about the software project in which a subroutine exists, so that the model does not rely only on the information in that subroutine.

We evaluate our project encoder in three ways. First, we implement our project encoder as an addition to four existing neural source code summarization techniques. We demonstrate

This work is supported in part by NSF CCF-1452959 and CCF-1717607 grants.

that our encoder boosts the performance of these techniques by between 4 and 8% in terms of BLEU scores in a large Java dataset, and between 9% and 17.5% in ensemble models in that dataset. Second, we compare our whole-project encoder with a competitive approach that attempts to summarize the context surrounding code, and found between 1.5% and 7% improvement in terms of BLEU score in the Java dataset. Third, we study the time and resource costs of our project encoder, to determine the costs associated with the increased performance of our approach.

We provide all data and implementations via our online appendix (see Section VII).

II. Background & Related Work

This section describes related work and supporting technologies, namely the neural encoder-decoder architecture.

A. Source Code Summarization

Figure 1 shows key papers related to source code summarization in the last four years. The list is not exhaustive and only includes peer-reviewed work. Papers are broadly categorized as based on the encoder-decoder architecture (column E), and whether their novelty (and primary means of improvement over baselines) is based on structural information about the source code itself (column S), or contextual information about surrounding source code (column C). Two observations are apparent. First, recent approaches are based on some variant of an encoder-decoder architecture. Prior to 2017, code summarization research focused on templates or information retrieval, but these have recently given way to neural encoder-decoder designs [2], [6], [32], [33].

A second observation is that the strong trend has been to squeeze ever more information out of the source code being summarized itself, in the form of structural information. The trend began around the time Hu *et al.* [9] used the abstract syntax tree to mark up the source code tokens in the encoder's input sequence, and was followed up by Allamanis *et al.* [20], LeClair *et al.* [7], among others noted in the table, with different AST-based code representations. Advancement continued as AST path-based encoders [24], [34] were followed by AST graph neural network-based encoders [35]. While some research has been dedicated to novel representations of the text in code (e.g. via Transformer models [29]), the tendency has been towards more and more complex representations of the code structure. Very recently, multi-edge and hybrid GNN structures have been devised [8], [31].

	E	S	C
Loyola *et al.* (2017) [16]	x		
Lu *et al.* (2017) [17]	x		
Jiang *et al.* (2017) [18]	x		
Hu *et al.* (2018) [19]	x		
Hu *et al.* (2018) [9]	x	x	
Allamanis *et al.* (2018) [20]	x	x	
Wan *et al.* (2018) [21]	x	x	
Liang *et al.* (2018) [22]	x	x	
Alon *et al.* (2019) [23], [24]	x	x	
Gao *et al.* (2019) [25]	x		
LeClair *et al.* (2019) [7]	x	x	
Mesbah *et al.* (2019) [26]	x	x	
Nie *et al.* (2019) [27]	x	x	
Haldar *et al.* (2020) [28]	x	x	
Ahmad *et al.* (2020) [29]	x		
Haque *et al.* (2020) [30]	x		x
Zügner *et al.* (2021) [8]	x	x	
Liu *et al.* (2021) [31]	x	x	
<this paper>	x		x

Fig. 1. Key peer-reviewed related work from the last four years. Column E means the approach is an encoder-decoder architecture. S means that the improvement of the model relies primarily on structural information about the source code being summarized, such as a subroutine's AST. C means the improvement is primarily due to contextual information.

Much more rare is work that attempts to improve performance by integrating contextual information. Code context may be broadly defined as the source code in the methods, files, and packages surrounding a particular snippet of code [36]. Program comprehension literature is quite clear that the context surrounding source code is critical to understanding that code, with work ranging from psychological/physiological studies [12]–[14] to empirical/technical solutions [37]–[40] verifying this conclusion. The use of code context for summarization was mainstream among older template- and IR-based techniques [2], [32], [41], though it is currently overlooked among neural network-based solutions. Haque *et al.* [30] are a notable exception. They use text from each function in the same file as part of the encoder portion of their model, and show improvement over different baselines.

This paper focuses on contextual information. Specifically, we focus on project context, which is the context provided by every source code file in the same project as the subroutine we are summarizing. This context is more broad than the file-level context proposed by Haque *et al.* [30], but like that work, our approach is complementary to most encoder-decoder approaches rather than competitive. Our approach augments the solutions based on the structure of the code itself, it does not replace them.

B. Neural Encoder-Decoder Architecture

The neural encoder-decoder architecture revolves around two independent vectorized representations of parallel data. The parallel data may be a sentence in one language and its translation in another language, an image and a caption of that image, or a subroutine and a natural language summary of that subroutine. Since each "side" of the parallel data may be quite different, the means of generating the vectorized representation will also be different. Usually the purpose of an encoder-decoder model is to create one "side" of the data out of the other (e.g. create a summary out of a subroutine). The input side is referred to as the encoder, and the output side is referred to as the decoder. Thus to train a model to e.g. translate from French to English, the encoder receives French sentences and the decoder receives the parallel English sentences. The encoder-decoder architecture has its roots in work by Sutskever *et al.* [42] published 2014. Since then

the architecture has blossomed and found an extremely wide variety of uses, as several survey papers testify [43]–[45].

The vast majority of encoder-decoder architectures work because of a similarity calculation that links the encoder and decoder representation, called an attention mechanism. Essentially what the attention mechanism does is compute the similarity between the encoder and decoder representations, which helps the model learn to associate features in those representations. For example, a single word in a French sentence would be associated with its counterpart in the English sentence. Attention was proposed by Bahdanau *et al.* [10] and has become an integral part of most encoder-decoder models.

This paper follows in the tradition of most encoder-decoder models, though with a small twist to the attention mechanism. As the next section will show, we maintain an independent attention mechanism for our whole-project encoder, so the model can learn to attend to both the encoder for that subroutine and our whole-project encoder. In effect, the model will learn from both a local context of the subroutine itself and a global context of the whole software project. When defined in these terms, our work is related to "cascade attention" from image processing [46]–[49]. For example, work by Wang *et al.* [50] detects human emotion with a closeup of a person's face and also a zoomed out image of the entire room. The "cascade" is that the model attends to both the closeup and the zoomed out image. The idea is that it may detect crying in a face, but then understand it as either sadness or happiness depending on the context. Likewise, our approach is to learn from the subroutine's source code (via any number of existing encoders), then form a better understanding of it with our whole-project encoder.

III. Our Approach

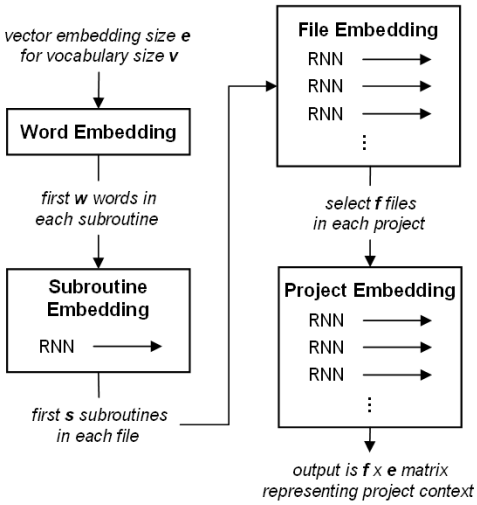
Our approach, in a nutshell, is to create an encoder of a selection of the files in the same project as a subroutine, then combine this encoder with an arbitrary encoder of the subroutine itself. This section starts with our definition of project context. We then provide an overview of the encoder and guidelines for combining with existing models.

A. Project Context

We define "project context" as all source code files of the same language in the whole software project in which a subroutine resides. For example, for a Java method, the project context would be all other Java files in the same project. The advantage of this broad definition of project context is that it allows the model to learn from the high level concepts that are described in many areas of the project (we illustrate this advantage with examples in Section VI). A potential disadvantage is that the project context will often be very large. A risk is that the model size could become so large that it is not feasible due to time or resource constraints – at the time of writing, not every user may be expected to have a GPU with 16gb VRAM, for instance. While we study this risk in RQ_4, controlling these potential costs is a key factor in our model design. Therefore, even though project context is defined as all source code files, not all information from all files will be included in the model.

B. Model Overview

Our model centers around four vector spaces: the word, subroutine, file, and project embeddings. The input to these is regulated by five hyperparameters, noted in the figures below:



Word Embedding The word embedding is identical to that presented in many papers on neural NLP topics, including code summarization. Essentially, each word is represented as an e-length vector. To control model size, a maximum of the first v most-common words is included in the embedding space, others are marked with a default out-of-vocab token, also in the embedding space. A "word" in source code is defined by the preprocessing procedure. We use the preprocessor recommended for code summarization by LeClair *et al.* [11].

Subroutine Embedding The subroutine embedding results in a vectorized representation of each subroutine. The input to the subroutine embedding is the word embedding vector for the first w words in the subroutine. Then we pass these words as a sequence through a recurrent neural network. The final state of that RNN is the vectorized representation of the subroutine. We chose to use the first w words (as opposed to w random words, words ranked via tf/idf, etc.) because these words will include the signature of the method, which has been shown to be the most important component for summarization [51].

File Embedding The file embedding results in a vectorized representation of each file. The input is the embedding for the first s subroutines in a file – an s x e matrix because each subroutine is represented with an e-length vector. We then use each row in the matrix as a position in a sequence, which we send to an RNN. The final state of the RNN is the file embedding vector. An RNN is a reasonable choice to combine subroutine vectors because the subroutines occur in the file in an order defined by the author of that file. The meaning of this order may be disputed, however, so future work may consider aggregating these vectors by some other means such as averaging.

Project Embedding The project embedding output is the final output of the project encoder, prior to applying attention.

The input is the file embedding for f files in the project. We chose these files from the project with an operation `SELECT`. In our implementation, the `SELECT` operation randomly chooses f files from the project for each subroutine – each subroutine has a new f random selections. The output of the project embedding is an f x e matrix in which each row is a file embedding and each column is an index in the vector representation of those files. Note that we do not aggregate this matrix into a single vector. The reason is that we use an attention mechanism (not shown in the figure above) to attend each position in the decoder to each position in this project embedding. The design of our attention mechanism is identical to Luong *et al.* [52], though in principle another may be used. The result is that the model will learn to attend to the most important files in the project embedding. We present an example of how attention to the project embedding helps the model in Section VI.

C. Implementation Guidelines

Our implementation guidelines fall into two categories: hyperparameter/setup recommendations, and suggestions for integration with other encoder-decoder models.

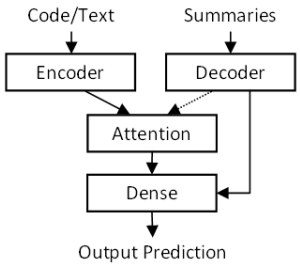
1) Hyperparameters: While a grid search for every parameter is not feasible due to high computation costs, we chose the following based on both related literature and pilot tests:

e	100	vector length
v	10000	vocab size
w	25	words per subroutine
s	10	subroutines per file
f	10	files per project
RNN	GRU	type of RNN

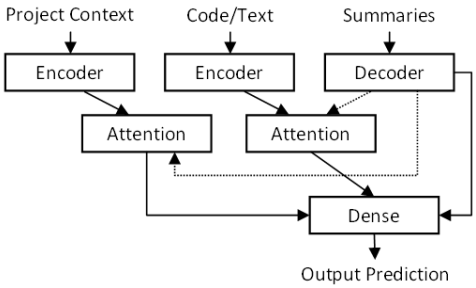
The values of 100 for e and 10,000 for v are based on successful results and recommendations by LeClair *et al.* [11] for neural source code summarization. The value for w is based on findings that the signature of a subroutine typically contains the most valuable textual information about that subroutine (since it neatly condenses the return type, name, and parameter types in a few words) – we chose 25 because that value covers the entire signature in a majority of subroutines and because several RNN designs have been shown to lose the ability to preserve dependencies when the sequence becomes too long. We chose a GRU as the RNN as a balance between ability to preserve dependencies and time cost of computation.

The values for s and f are more subjective. On the one hand, maximizing these numbers means the model can consider much more of the project context. But on the other hand, computation and memory cost will rapidly become prohibitive. Consider that one file is 100kb of memory in the model (s x w x e x 4 bytes = 10 x 25 x 100 x 4 bytes). Each project context matrix is then 1mb (f = 10 files). The costs add up because the datasets may involve millions of subroutines.

2) Integration: Recall that our intent for our whole-project encoder is to be integrated with the encoder portion of an existing encoder-decoder model. The simplest means of integration is to treat the whole-project encoder as independent of all other parts of the model, and connect its output to the existing encoder's output after attention is applied. Consider a "vanilla" seq2seq-like model like those used in the first neural code summarization papers [9], [33], that has a single RNN as an encoder of the words in the subroutines and a single RNN as the decoder for the words in the summary. This model would typically have an attention mechanism between the encoder and decoder which would adjust the emphasis of the information in the encoder based on the decoder. Then the attended encoder output would be combined to the decoder output and connected to a fully-connected output, as below:



Integrating this model with our whole-project encoder would involve rewiring the output of the decoder to an attention mechanism for the whole-project encoder (in addition to the pre-existing encoder), as mentioned under the Project Embedding heading in the previous section. Then the output of the existing encoder, the whole-project encoder, and the decoder would be combined and connected to the fully-connected output layer:



Three key details stand out as questions for this implementation (for maximum clarity, readers may elect to follow along in our implementation in file `models/attendgru_pc.py` in our online appendix (Section VII)). First, we combine the output of the encoders and the decoder by concatenating the matrices, which triples the vector length. We then use a fully-connected layer to squash these long vectors back to the specified vector size e (lines 80-85 in the implementation). The effect is that the model learns how to combine the vectors during training.

Second, we share the word embedding space between the code/text encoder and the whole-project encoder (around line 64 in the implementation). This is possible because both models use the same vocabulary, and saves both memory space and computation time. Separate word embeddings would be necessary for different vocabularies, or if it is desirable to use a pretrained embedding, etc.

Third, the project context input is separate from the other encoder/decoder input, and must be extracted from the dataset prior to training. This requirement is not likely to be a problem for models that already use the source code of the subroutines

to generate summaries, but there may exist application domains where this data is not available.

IV. Evaluation

In this section, we describe our evaluation, including research questions, methodology, datasets, and baselines.

A. Research Questions

Our research objective is to determine the degree of difference in performance that our whole-project encoder imbues on other recent neural code summarization techniques. We explore this difference from several angles by asking the following Research Questions (RQs):

- RQ_1 What is the difference in performance of recent baselines when augmented with our encoder, according to standard quality metrics over a large dataset?
- RQ_2 What is the difference in performance when measuring only the action word prediction quality?
- RQ_3 What is the difference in performance compared to a baseline code context encoder?
- RQ_4 What is the cost of the performance increase in terms of model time to train?

The rationale behind RQ_1 is that the vast majority of neural code summarization research uses a set of established metrics (namely, BLEU) to evaluate the quality of the predicted summaries to a gold set. We follow this practice to compare the baselines to our augmented versions of those baselines. Evaluations in many papers stop here. Yet, complaints are rising that the accepted evaluation practice may not capture quality to the extent desired [53], so we ask several more RQs to give a more complete picture.

We ask RQ_2 in light of new recommendations by Haque *et al.* [54] on evaluating code summarization techniques. They observe that an overwhelming majority of code summaries start with an action word, in keeping with style guide recommendations (e.g. "connects to game server" or "adds row to sql table"). They find that this action word is a critical piece of the prediction quality, and recommend that the quality of the prediction of these words should be assessed independently from the assessment of the entire summary output.

The purpose of RQ_3 is to evaluate our approach against a baseline for contextual information. Recall that most code summarization techniques focus on the code within a subroutine (or other code snippet) itself. Our approach augments these techniques. However, at least one other code context encoder has been proposed, so we evaluate against it.

The rationale behind RQ_4 is that adding our whole-project encoder will impose some time cost over the baselines, and recent work from industry reports that training time due to added model complexity creates engineering difficulties in practice [55]. We study this cost to guide cost-benefit analysis to using our approach.

B. Methodology

Our methodology for answering RQ_1 closely follows the procedures of almost all neural source code summarization papers to date. We obtain two datasets of subroutines and summaries of those subroutines, and divide them into training/validation/test subsets (our dataset preparation procedures described in the next section). Then we train each of our baselines (also described below) with these datasets to a maximum of 10 epochs. We use the teacher forcing training procedure, as do most recent code summarization papers [34], [35]. We chose the trained model from the epoch that achieved the highest validation accuracy, so each baseline has the opportunity to find an optimum within a reasonable training time ceiling (each epoch for most models takes 2-3 hours, so ten epochs is approximately 24 hours). Then we use each baseline's trained model to predict a code summary for the subroutines in the test set. Finally, we report the BLEU [56] and ROUGE-LCS [57] scores for each baseline. We repeat the entire procedure for our versions of the baselines that we augment with the whole-project encoder. Note that this procedure for RQ_1 is not novel – our intent is to adhere to community standards.

We elected to focus on an in-depth metrics-driven evaluation rather than a human study. While human studies are often considered a gold standard for evaluation, Chatzikoumi *et al.* [58] point out that reality is more nuanced. Human studies are very valuable, but have two key problems. First, they are not reproducible because people are subject to biases, fatigue, mistakes, and other factors, so people may give very different results. Second, humans can only be expected to evaluate a few dozen or hundred samples. In this paper, we have 24 model configurations to test, and a test set with tens of thousands of samples. Therefore, we decided to focus on an in-depth analysis of metrics-driven evaluation.

To answer RQ_2, we follow the recommendations of Haque *et al.* [54]. The training process is almost identical to RQ_1. The difference is that a filter based on the Stanford NLP package [59] is used to extract the action word from the gold set summary (in practice, this is usually the first word), and the model is trained to predict just that word. Then during testing, the model is asked to predict the action word for the subroutines in the test set. Precision and Recall [60] are used to assess the quality of the predictions for each action word: precision is the percent of predictions of that action word which were correct, recall is the percent of instances of that action word in the gold set that are predicted. The macro average of these precision and recall values across all action words is reported. For clarity, we also produce confusion matrices to the extent possible within page limits.

For brevity, we select only a subset of the best-performing approaches from RQ_1 and RQ_2 as representative examples for the remaining RQs. However, we provide further results and details in our online appendix.

Our methodology for RQ_3 is to follow the same procedures as in RQ_1 and RQ_2, except to compare models augmented with our whole-project encoder to models augmented with a file context encoder proposed by Haque *et al.* [30].

For RQ_4, we measure the size and training time for each of the models during training for RQ_1. We report these resource

costs alongside performance improvements for those models.

C. Datasets

We use a Java dataset of 2.1m Java methods from 28k projects created by LeClair *et al.* [11] under strict quality guidelines. These guidelines were tested for their effect on code summarization results, namely that the training and test sets are split by project, so that data from the test set does not leak into the test set by virtue of being in the same project. We do not use datasets from other papers because they tend to be drawn from the same set of projects on online, open-source repositories (namely, Github), they tend to be smaller, and they are not vetted to the degree as this Java dataset.

We made one key change to the dataset in this paper when compared to previous papers: we improved the filter for code clones among the subroutines. The original configuration in the datasets filtered code clones only by exact duplicates. Since then, a study by Microsoft Research determined that this filter was insufficient for some ML tasks related to code, and recommended a new filtration procedure [61]. We applied that filter to the dataset in this study. The results we report for baselines in our experiments may have different values (usually lower values) than reported in the original papers for those baselines even for the same datasets. The reason for this difference is our stricter removal of code clones. It was necessary to rerun all experiments rather than rely on results reported in earlier papers, though the only difference in data or configuration was the code clone filtration procedure.

D. Baselines

We use four baseline neural code summarization techniques. We then augment each with our whole-project encoder. At a technical level, we build our implementations in a framework provided by Haque *et al.* [30] in their reproducibility package for their paper on file context encoding.

`attendgru` is a typical seq2seq-like design like those used in early neural code summarization papers (and mentioned in Section III-C2). The only input to the encoder is the text from the source code of the subroutine itself.

`ast-attendgru` was proposed by LeClair *et al.* [7] and built on `attendgru` as well as work by Hu *et al.* [9]. It uses a flattened AST to represent subroutines.

`graph2seq` is a representative example in a recent class of graph neural network (GNN)-based techniques. These techniques use information extracted from the AST and other relationships in code [8], [31], [35].

`code2seq` is a representative of AST path-based representations of code. This baseline is a faithful reimplementation of a model proposed by Alon *et al.* [34], though with several hyperparameter changed to match those in other baselines.

We denote the versions of these baselines augmented with our whole-project encoder with the suffix **`-pc`** for "project context." For example, `attendgru-pc` and `code2seq-pc`.

For RQ_3, we use the file context encoder from Haque *et al.* [30] as a baseline. Models augmented with the file context encoder are denoted with the suffix **`-fc`**.

E. Software / Hardware Details

Our hardware platform consisted of an HP Z-640 workstation with a Xeon E-1650v4 CPU, 128GB system memory, and two Nvidia Quadro P5000 GPUs with 16GB VRAM each. Key software included CUDA 10.0 and Tensorflow 2.4.

F. Threats to Validity

The key threats to validity to this study include the datasets and the implementation details. We chose a vetted dataset with millions of examples, but it is possible that results may not generalize to all datasets or other languages. Likewise, results may vary given the plethora of implementation decisions, such as the means of combining the whole-project encoder output with other encoder output. Caution is advised in drawing conclusions from these results beyond the scope of large open-source dataset in Java, or when implementation details differ significantly from those presented.

V. Evaluation Results

In this section, we discuss our evaluation results, including answers to our research questions and supporting analysis.

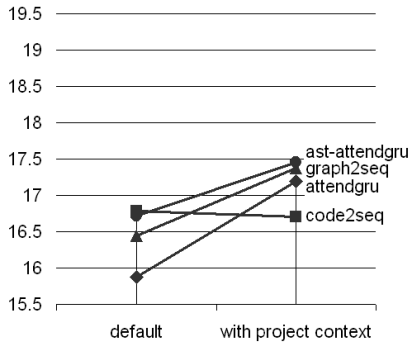
A. RQ_1: Effect of Augmenting Baselines

We found improved levels of BLEU and ROUGE scores across several baselines and configurations, when comparing default versions of the baselines to versions augmented with our project encoder. Figure 2 summarizes these results. We report results under two key conditions: solo and ensemble. A solo model is a single trained model – it includes the model weights of the epoch which achieved the highest validation accuracy, under the training procedure described in Section IV-B. An ensemble model combines two trained models. For example, the models for `attendgru` and `attendgru-pc` would be combined to form an ensemble model denoted "nc+fc" (no context plus file context, see first column of Ensemble Models table in Figure 2). The combination procedure is to calculate the element-wise mean of the output predictions from each model, as recommended by Garmash *et al.* [62]. We use this procedure in light of experimental findings for ensemble neural code summarization models by LeClair *et al.* [7].

Two observations stand out for the solo models. First, for the three baselines `attendgru`, `ast-attendgru`, and `graph2seq`, aggregate BLEU score improves between 4 and 8% when our project encoder is added to the model. The greatest improvement occurred for `attendgru`, which rose from 15.87 to 17.19 BLEU. Note that this version is the one described in our Integration example in Section III-C2. It shows that even a relatively simple baseline can achieve competitive BLEU scores by adding project context (`attendgru` is just a vanilla seq2seq-like model with a single unidirectional GRU in the encoder and decoder).

Higher performance is observed for `ast-attendgru` and `graph2seq`, which is expected based on previous studies [30]. The higher performance is because both model designs use information from the AST of the subroutine. The `graph2seq` model uses a GNN, while `ast-attendgru` flattens the tree

		BLEU					ROUGE-LCS		
		A	1	2	3	4	P	R	F
Solo Models	attendgru	**15.87**	36.22	18.89	11.55	8.03	55.22	46.98	48.94
	attendgru-fc	**16.67**	36.53	19.59	12.31	8.76	56.54	47.44	49.78
	attendgru-pc	**17.19**	37.34	20.20	12.71	9.10	56.12	47.81	49.88
	ast-attendgru	**16.72**	37.18	19.84	12.30	8.62	56.16	47.83	49.84
	ast-attendgru-fc	**17.18**	37.64	20.21	12.69	9.03	55.86	48.11	49.94
	ast-attendgru-pc	**17.45**	37.34	20.47	13.02	9.32	57.08	48.05	50.37
	graph2seq	**16.44**	36.14	19.54	12.18	8.49	57.43	47.32	50.01
	graph2seq-fc	**16.26**	35.95	19.19	11.95	8.48	56.56	46.95	49.50
	graph2seq-pc	**17.37**	37.61	20.37	12.89	9.21	55.70	47.87	49.74
	code2seq	**16.78**	37.88	20.14	12.30	8.45	55.45	48.32	49.80
	code2seq-fc	**16.45**	36.34	19.43	12.09	8.58	56.66	47.29	49.79
	code2seq-pc	**16.7**	35.92	19.53	12.44	8.91	57.75	47.08	50.01



			BLEU					ROUGE-LCS		
		mix	A	1	2	3	4	P	R	F
Ensemble Models	attendgru	nc+fc	**17.96**	37.83	20.92	13.49	9.74	58.26	48.79	51.29
	attendgru	nc+pc	**18.18**	38.26	21.14	13.64	9.91	57.80	48.98	51.18
	attendgru	fc+pc	**18.67**	38.63	21.62	14.10	10.32	58.49	49.29	51.71
	ast-attendgru	nc+fc	**18.47**	38.68	21.53	13.88	10.07	58.15	49.33	51.57
	ast-attendgru	nc+pc	**18.84**	38.80	21.88	14.26	10.41	58.71	49.50	51.91
	ast-attendgru	fc+pc	**19.06**	39.09	22.07	14.43	10.60	58.50	49.63	51.93
	graph2seq	nc+fc	**17.77**	37.32	20.79	13.37	9.62	59.16	48.58	51.50
	graph2seq	nc+pc	**18.59**	38.85	21.56	14.06	10.28	58.54	49.07	51.58
	graph2seq	fc+pc	**18.80**	38.62	21.72	14.23	10.48	58.56	49.20	51.70
	code2seq	nc+fc	**18.16**	38.35	21.24	13.60	9.81	58.14	49.14	51.47
	code2seq	nc+pc	**18.56**	38.50	21.55	13.99	10.21	58.49	49.23	51.63
	code2seq	fc+pc	**18.29**	37.67	21.12	13.82	10.18	59.15	48.72	51.64

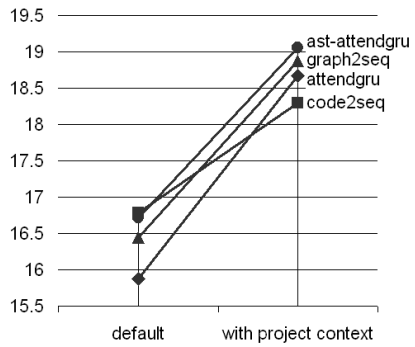


Fig. 2. Results summary for RQ_1 and RQ_3. The table shows the BLEU and ROUGE-LCS scores for baselines and our augmented versions of those baselines. Chart depicts relative improvement of our augmented version according to aggregated BLEU score. Column "mix" indicates which models were ensembled: nc for default/no-context, fc for file context, and pc for project context.

and uses it as input to an RNN. Note that while both models see an improvement with project context, it is lower in relative terms than for `attendgru`. We attribute this lower relative improvement to the increased amount of information that the model must learn in the same size vector space close to the output layer of the model. Recall from Section III-C2 paragraph 3 that two vectors of size e are generated: one for the original encoder and one for the project context encoder. Concatenating them results in a vector of length 2xe. To control model complexity, it is necessary to squash these vectors back to size e with a dense network. It is likely that information is lost. This effect probably also explains the *drop* in performance for `code2seq`. That baseline is extremely complex, and the vector size of e may be too confining.

The ensemble results are, in general, much higher. The chart at the lower-right of Figure 2 shows the default solo configuration to the "fc+pc" ensemble test condition. All models demonstrate considerable improvement, between 9 and 17.5% The reason for this improvement proffered by Garmash *et al.* [62] is that different models contribute more to some predictions than others. Mathematically, it means that the value of the argmax in the output vector of some models will be higher than others, because that model recognized a pattern closely-associated with that prediction. Haque *et al.* [30] pointed out that file context-based models contribute more to some subroutines than others. Project context helps overall, but there are still subroutines for which other models are more useful. The best performance is achieved by combining them.

Consider Figure 3. The improvement of different models is not necessarily distributed equally over all subroutines. The chart on the left shows that of 73k subroutines in the test set, `attendgru` earned the highest score for about 27.5k, compared to about 30k for `attendgru-pc`. This means that the reason `attendgru-pc` improved is because it created better predictions for only a portion of the results. In that light, consider the chart on the right. That chart compares solo `attendgru` to the "nc+pc" ensemble. It shows that there is a substantial subset of subroutines for which `attendgru` earns a higher BLEU score, even compared to the "nc+pc" ensemble of `attendgru` and `attendgru-pc`. What changes is that there is a much higher number of ties. The ensemble sometimes creates predictions more like `attendgru`, and likewise more like `attendgru-pc` for other subroutines. What is happening is that the output vector from `attendgru` has higher values for the predictions where it finds patterns closely associated with those predictions – when it does not find those patterns, the values are lower and `attendgru-pc` is often higher. The result is a better overall BLEU score.

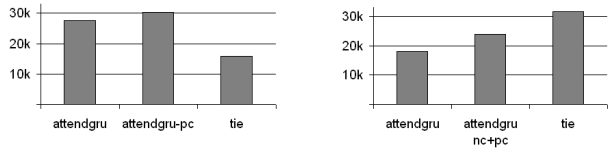


Fig. 3. (left) Number of subroutines in the test set for which `attendgru` and `attendgru-pc` each had a higher BLEU score, and the number of ties. (right) Comparison of solo `attendgru` to ensemble `attendgru nc+pc`.

Java top-10	return	set	get	add	create	initialize	test	remove	check	is	*other*
return	6160	32	2365	10	72	14	6	6	262	1	2093
set	11	10388	13	40	9	15	2	2	1	0	558
get	4491	22	3210	7	13	14	2	2	30	0	907
add	4	46	1	2950	14	1	10	3	1	0	293
create	62	21	25	43	980	16	16	0	2	0	538
initialize	23	14	3	2	22	1632	0	0	0	0	201
test	35	2	2	1	2	1	1232	0	61	0	170
remove	6	1	0	1	0	1	3	1186	0	0	303
check	244	9	3	2	3	0	66	3	526	3	607
is	95	39	10	15	13	7	4	7	49	29	366
other	2715	2147	1023	517	605	256	727	190	378	3	22266

Java	top-40			top-10			top-10n			get/set		
	p	r	f	p	r	f	p	r	f	p	r	f
attendgru	.53	.44	.45	.69	.61	.60	.70	.54	.54	.99	.99	.99
attendgru-fc	.52	.46	.47	.68	.63	.62	.72	.51	.53	.99	.99	.99
attendgru-pc	.53	.47	.47	.65	.62	.61	.71	.53	.54	.99	.99	.99
ast-attendgru	.54	.46	.47	.69	.61	.61	.72	.52	.53	.99	.99	.99
ast-attendgru-fc	.55	.47	.47	.61	.61	.60	.72	.51	.53	.99	.99	.99
ast-attendgru-pc	.55	.44	.46	.67	.63	.62	.68	.52	.53	.99	.99	.99
graph2seq	.56	.45	.46	.68	.61	.61	.69	.54	.55	.99	.99	.99
graph2seq-fc	.55	.45	.46	.70	.60	.61	.68	.52	.54	.99	.99	.99
graph2seq-pc	.56	.45	.46	.70	.59	.60	.71	.51	.54	.99	.99	.99
code2seq	.54	.47	.46	.70	.59	.59	.71	.49	.52	.99	.99	.99
code2seq-fc	.53	.47	.47	.71	.59	.60	.71	.53	.54	.99	.99	.99
code2seq-pc	.54	.46	.47	.68	.62	.62	.69	.53	.54	.99	.99	.99

Fig. 4. *(top)* Confusion matrix showing results for top-10 action words for ast-attendgru-pc, the best performer in terms of f-measure. *(bottom)* Overall results under standard conditions in the Java dataset.

B. RQ$_2$: Action Word Prediction

We found broadly similar performance in terms of precision and recall for action word prediction. Recall that Haque *et al.* [54] recently recommended focusing on the prediction of the action word in source code summaries, given that word's importance in the summary. Following these recommendations, we report the top-40, top-10, top-10n (which is the top 2-12, skipping get/set), and get/set. The model should be able to distinguish get from set with very high accuracy, the top-10 and top-10n results being a more difficult problem, and the top-40 with even more difficulty. The idea is that if the model cannot even predict the correct action word, then it may have little hope of predicting the rest of the summary.

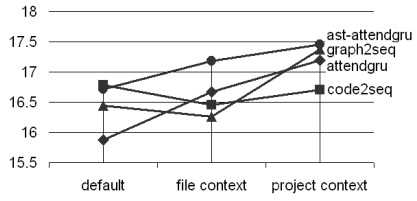
We do not observe a large difference attributable to file or project context. Our interpretation of this result is that the subroutine itself tends to provide most of the information needed to predict the action word – many times the correct action word is in the name of the function e.g. "book" for `book()` in the class `Seat` in project `AircraftTravel`. The higher BLEU and ROUGE scores for project context must therefore be due to improvements in the prediction of other parts of the code summary. For example, if the summary is "book seat on airplane", the subroutine name will provide the action word "book", but code context will help find "seat" and "airplane." We explore an example like this in Section VI.

C. RQ$_3$: Comparison to File Context

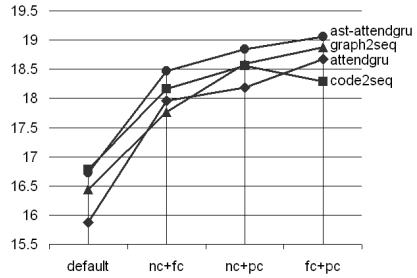
We observe improvement over the baselines that are enhanced with file context. Figure 5 depicts the change in aggregate BLEU score across key model configurations. Figure 5a shows the default baseline model, followed by the file context and project context versions of those models. Figure 5b shows the default baseline model, followed by the nc+fc and nc+pc ensembles. Recall that the "file context" versions are those provided by Haque *et al.* [30] and form the nearest competition for models that include code context (see Section II-A).

Overall, the project context versions of the baselines achieve higher aggregate BLEU scores than the file context versions. However, the gains are not uniform. For example, note in Figure 5a that `graph2seq-fc` is the lowest performing file context model, while `graph2seq-pc` is nearly tied for the top position of solo models. This finding seems to be at odds with scores reported by Haque *et al.* [30] for `graph2seq-fc`. We attribute the difference to the enhanced removal of code clones we performed for experiments in this paper (see Section IV-C). The file context contains many subroutines that are considered clones by the recommend clone removal technique [61], because these subroutines may be overloaded or only slightly modified. Future researchers using file context may consider leaving clones in the file context, and only remove them from the list of subroutines in the test set to ensure fairness.

The ensemble models also show that project context helps achieve higher BLEU scores than file context. Figure 5b shows marked improvement from nc+fc ensembles (no context combined with file context) to nc+pc ensembles, then again from nc+pc to fc+pc ensembles. The exception is `code2seq`, which we believe is due to the same vector size restriction described in Section V-A. The gain of fc+pc over other ensembles implies that file context and project context contributes to model predictions in orthogonal ways, in the same vein as the ensemble results for RQ$_1$, above.



(a) Solo



(b) Ensemble

Fig. 5. Comparison of aggregate BLEU scores for `-fc` and `-pc` models. This figure is a depiction of values in Figure 2.

D. RQ_4: Effects on Model Size

Adding project context to a baseline increases the complexity of the model, and this complexity comes at a cost in terms of time to train. While it may be tempting to write off training time as a "one time sunk cost", in fact this added time imposes engineering challenges that affect cost-benefit decisions [55]. We report training time per epoch as a proxy for this complexity cost. All data points were collected on the hardware platform described in Section IV-E.

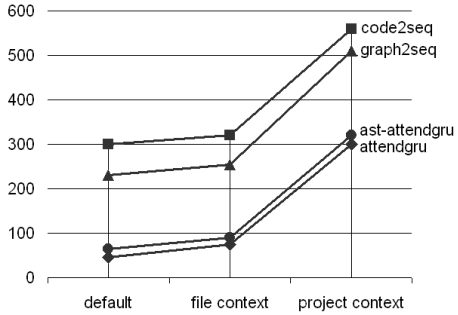


Fig. 6. Training time in minutes per epoch.

We observe about a 3x time cost for `attendgru` and `ast-attendgru`, and about a 2x cost for `graph2seq` and `code2seq`, when comapring default configurations to the versions with project context. While the number of minutes is subject to hardware and software settings, we report these numbers to assist practitioners in deciding how to deploy these technologies. For instance, time required for `ast-attendgru-pc` is roughly equal to `code2seq-fc` even though `ast-attendgru-pc` achieves a higher BLEU score. At that time limit `ast-attendgru-pc` may be the best choice even if `code2seq` is a better baseline than `ast-attendgru`.

An implication of this finding is that the costs of including project context can be very high. We find improvement in terms of overall BLEU score, especially for ensemble models, the training difficulty is 2x to 3x even for a modest setting of f=10. Future researchers may note the potential of even a portion of project context for improving prediction performance, and may consider guiding effort into reducing the costs so that more context may be considered.

VI. Discussion & Conclusion

This paper advances the state of the art in two ways:

1) We propose an encoder that creates a vectorized representation of project context for use in neural models of software source code.
2) We demonstrate the benefit of this encoder for the specific problem of source code summarization.

The first advancement is important because of its potential impact on many areas of software engineering research. Allamanis *et al.* [63] present a survey of neural models for various software engineering tasks, and separate these tasks into two categories: code generational and code representational. A code generational task is like code completion or automatic repair, in which the model is expected to create new source code. A code representation task is like code summarization or bug localization, in which the model must create some internal representation of the program, and use it to predict something about the software, such as a code summary or if a subroutine contains a particular kind of bug.

The project context encoder we propose has potential in many code representational tasks. Essentially what the encoder does is create a vectorized representation of the files surrounding a particular area of code in a project. This representation could be used in many ways. For example, a neural model for predicting bugs based on the code in a subroutine could append our project context encoder. It may help the model learn that a particular pattern in the code may be associated with bugs for some types of projects, but not others. This benefit is only a hypothetical discussion – the point is that this paper may have benefits beyond the specific problem of code summarization.

The implementation and experiment in this paper focus on code summarization, and demonstrate how project context improves predictions in terms of BLEU and ROUGE scores for several baselines (RQ_1). We show that our model seems to be providing orthogonal information by improving predictions for a subset of subroutines, and that by using an ensemble procedure, the benefits of project context can be combined with file context and default models. We also show that these improvements seem to be focused on areas outside the action word (RQ_2), and that project context tends to result in overall better scores that file context alone (RQ_3), even if an ensemble has the highest observed performance (fc+pc ensemble models, RQ_1). Our project context encoder leads to an advancement of the state of the art for code summarization.

The reason that project context helps is that many methods are very difficult to understand from only the source code of a single subroutine. Code summarization techniques that consider only the subroutine itself make the tacit assumption that that subroutine contains all the information necessary to summarize it. Consider Example 1 (method #29987000 in the dataset [11], we list ID numbers for reproducibility). This method is from a GUI program for managing config files. Its purpose is to stop and cleanup a plugin. However, this purpose is hard to ascertain without seeing the project context.

Compare the predicted summary for Example 1 by `attendgru` versus `attendgru-pc`. The model `attendgru` predicts "stops the bundle", which seems a reasonable guess considering that it has access only to the source code of that subroutine. The method is called `stop()`, which begs the question "stops what?" The word "plugin" is in the source code, but so is the word "translator", "messages", "bundle", etc. Many methods in the training set have a pattern in which the action word is followed by a word from the parameter list [54], and a simple seq2seq-like model such as `attendgru` can learn this pattern effectively [7], [9]. So `attendgru` guesses "stops the bundle."

The project context helps the model learn what the method really means. The method in Example 1 appears in the file `ConfexPlugin.java`, which is in the project with five other files. While there is far too much data in these files to reprint

Method #29987000: (from test set)

reference	this method is called when the plug in is stopped
attendgru	stops the bundle
attendgru-pc	this method is called when the plug in is stopped

```
public void stop(BundleContext context)
    throws Exception {
  super.stop(context);
  plugin = null;
  Translator.removeAllMessages();
  Translator.removeAllTranslatables();
}
```

files in net.confex.application:

ApplicationWorkbenchAdvisor.java
ApplicationWorkbenchWindowAdvisor.java
ConfexApplication.java
→ ConfexPlugin.java
Perspective.java
ToolbarLayout.java

Example 1. A method from the test set for which attendgru-pc wrote the correct summary, while attendgru did not. Method is in the project net.confex.application, which contains the six listed files.

Method #805539: (from training set)

reference this method is called when the plug in is stopped

```
public void stop(BundleContext context)
    throws Exception {
  super.stop(context);
  if (this.logManager != null) {
    this.logManager.shutdown();
    this.logManager = null;
  }
  if (searchProviderManager != null) {
    searchProviderManager.dispose();
    searchProviderManager = null;
  }
  plugin = null;
}
```

files in net.bioclipse:

ApplicationWorkbenchAdvisor.java
ApplicationWorkbenchWindowAdvisor.java
ApplicationWorkbenchActionBarAdvisor.java
BioclipsePerspective.java
→ BioclipsePlugin.java
PerspectiveOpenPreferencePage.java

Example 2. Method in the training set seen by both approaches. Note list of files is similar to Example 1 because both are built with the same GUI platform. Project context helps attendgru-pc detect this similarity.

here, one may note the similarity between these files and the files in Example 2 (method ID numbers may be used to recover these files in the dataset). Example 2 is from the training set. The content of the method itself is quite different from Example 1 (aside from the method signature), so `attendgru` has difficulty seeing the two methods as similar – there are many methods named `stop()` that have nothing to do with plugins. But `attendgru-pc` has access to the project context and can identify the similarity of the files in this context. As a result, `attendgru-pc` predicts the summary that it has learned during training, which is correct.

We caution that we selected these examples as a demonstration of what project context can offer, and may not be representative of how the model always behaves. The model can make incorrect predictions – recall from Section V-A that there is a subset of methods for which project context outperforms the baseline, and a subset where it does not. However, when the project context models go astray, it tends to be because they are recognizing patterns in the context, and we found that ensemble models can generate better summaries.

Our intent in this paper is to propose a technique for encoding the project context of source code. While project context is quite expansive, our technique can capture enough of this context to be useful for the task of source code summarization. Essentially what have shown is that even a small amount of this project context – just f=10 in this paper – can lead to significant improvements in the aggregate BLEU scores of several baselines. In ensemble models, the benefit increases further. However, as we observe in RQ_5, the costs of including even limited project context mushroom rapidly. Future work aims to capture more of this context and interactions among the code components such as dependency relationships, and to demonstrate the benefit of context to other areas using neural models of source code.

VII. Reproducibility

We strongly encourage reproducibility. We provide the following online appendix to facilitate reuse of this technology by practitioners and other researchers. Our code, dataset scripts, and operating instructions may be found at:

`https://github.com/aakashba/projcon`

Acknowledgment

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

References

[1] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[2] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[3] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*. ACM, 2002, pp. 26–33.

[4] D. Kramer, "Api documentation from source code comments: a case study of javadoc," in *Proceedings of the 17th annual international conference on Computer documentation*. ACM, 1999, pp. 147–153.

[5] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[6] F. Zhao, J. Zhao, and Y. Bai, "A survey of automatic generation of code comments," in *Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences*, 2020, pp. 21–25.

[7] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 795–806.

[8] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=Xh5eMZVONGF

[9] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[11] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 3931–3937.

[12] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 378–389. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568252

[13] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[14] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[15] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 1993, pp. 482–498.

[16] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2017, pp. 287–292.

[17] Y. Lu, Z. Zhao, G. Li, and Z. Jin, "Learning to generate comments for api-based code snippets," in *Software Engineering and Methodology for Emerging Domains*. Springer, 2017, pp. 3–14.

[18] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.

[19] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 2269–2275.

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

[21] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 397–407.

[22] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[23] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *International Conference on Learning Representations*, 2019.

[24] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[25] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A neural model for method name generation from functional description," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 414–421.

[26] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 925–936.

[27] P. Nie, R. Rai, J. J. Li, S. Khurshid, R. J. Mooney, and M. Gligoric, "A framework for writing trigger-action todo comments in executable format," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 385–396.

[28] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," *arXiv preprint arXiv:2005.06980*, 2020.

[29] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[30] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," *International Conference on Mining Software Repositories*, 2020.

[31] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid {gnn}," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=zv-typ1gPxA

[32] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 279–290.

[33] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "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)*, 2016, pp. 2073–2083.

[34] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *International Conference on Learning Representations*, 2019.

[35] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *28th ACM/IEEE International Conference on Program Comprehension (ICPC'20)*, 2020.

[36] J. Krinke, "Effects of context on program slicing," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1249–1260, 2006.

[37] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–37, 2014.

[38] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 2002, pp. 271–278.

[39] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 255–265.

[40] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE, 2001, pp. 103–112.

[41] G. Neubig, "Survey of methods to generate natural language from source code."

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, pp. 3104–3112, 2014.

[43] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[44] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, p. 92, 2018.

[45] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53 040–53 065, 2019.

[46] Y. Zhu, R. Li, Y. Yang, and N. Ye, "Learning cascade attention for fine-grained image classification," *Neural Networks*, vol. 122, pp. 174–182, 2020.

[47] Y. Wang, H. Shen, S. Liu, J. Gao, and X. Cheng, "Cascade dynamics modeling with attention-based recurrent neural network." in *IJCAI*, 2017, pp. 2985–2991.

[48] M.-C. Sun, S.-H. Hsu, M.-C. Yang, and J.-H. Chien, "Context-aware cascade attention-based rnn for video emotion recognition," in *2018 First Asian Conference on Affective Computing and Intelligent Interaction (ACII Asia)*. IEEE, 2018, pp. 1–6.

[49] F. Li, R. Feng, W. Han, and L. Wang, "Ensemble model with cascade attention mechanism for high-resolution remote sensing image scene classification," *Optics Express*, vol. 28, no. 15, pp. 22 358–22 387, 2020.

[50] K. Wang, X. Zeng, J. Yang, D. Meng, K. Zhang, X. Peng, and Y. Qiao, "Cascade attention networks for group emotion recognition with face, body and image cues," in *Proceedings of the 20th ACM International Conference on Multimodal Interaction*, 2018, pp. 640–645.

[51] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th international conference on Software engineering*. ACM, 2014, pp. 390–401.

[52] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[53] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A human study of comprehension and code summarization," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 2–13.

[54] S. Haque, A. Bansal, L. Wu, and C. McMillan, "Action word prediction for neural source code summarization," *28th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021.

[55] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[56] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: http://dx.doi.org/10.3115/1073083.1073135

[57] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[58] E. Chatzikoumi, "How to evaluate machine translation: A review of automated and human metrics," *Natural Language Engineering*, vol. 26, no. 2, pp. 137–161, 2020.

[59] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit." in *ACL (System Demonstrations)*, 2014, pp. 55–60.

[60] A. Tharwat, "Classification assessment methods," *Applied Computing and Informatics*, 2020.

[61] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[62] E. Garmash and C. Monz, "Ensemble learning for multi-source neural machine translation," in *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, 2016, pp. 1409–1418.

[63] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.