From Source Code to Natural Language: An Exploration of Code Summarization Techniques

Gelei Xu
University of Notre Dame
gxu4@nd.edu

Ningzhi Tang
University of Notre Dame
ntang@nd.edu

1 Introduction

Programmers invest substantial time in reading code. Research suggests that, instead of trying to comprehend the entire project simultaneously, they typically concentrate on specific sections within the codebase (Singer et al., 2010). Nonetheless, traversing intricate source code to identify and understand particular portions can be a time-intensive and demanding endeavor. Consequently, the study reveals that programmers frequently skim code to save time (Starke et al., 2009). Skimming is valuable as it enables programmers to swiftly grasp the core essence of the codebase. However, sharing this acquired knowledge solely through skimming can pose challenges when disseminating it to other team members (Rodeghero et al., 2014).

To address this challenge, programmers create codebase documentation to explain its functionality. By summarizing the code's essence within concise documentation, programmers can enhance their comprehension of its structure and logic (Ahmad et al., 2020). Programmer-oriented software documentation is constructed from source code summaries. A summary provides a concise account of a code section, aiding programmers in comprehending its purpose and functionality without the need to delve into the code itself. These summaries provide a streamlined, high-level perspective of the code, serving as navigational tools to assist programmers in locating relevant sections and grasping the code's fundamental operations.

Programmers often desire effective documentation for their own use, but they frequently struggle to produce adequate documentation for others due to time pressure and language barriers (Haque et al., 2022). Additionally, manually generated summaries are prone to incompleteness and may not keep pace with code updates. As code undergoes revisions and improvements, these summaries can become outdated, potentially leading to misinformation. This discrepancy has the potential to impede rather than enhance code comprehension. Consequently, the demand for automated code summarization techniques has grown significantly (Rodeghero et al., 2014).

Automated code summarization seeks to connect the ever-changing codebase with the demand for accurate, current, and effortlessly generated summaries (McBurney and McMillan, 2014). These methods utilize natural language processing and machine learning algorithms to automatically extract essential information from the code, producing concise and coherent summaries (Wan et al., 2018). This process ensures that the summaries remain synchronized with the codebase's real-time alterations, regardless of their frequency. This automated approach not only enhances the precision of the summaries but also conserves programmers' time and effort that might otherwise be spent by manual summarization.

In this project, our aim is to explore both the classic and the state-of-the-art methods for code summarization. We first implemented the classic methods as the baseline, such as the Vector Space Model (VSM) (Haiduc et al., 2010) with the TF-IDF method (Sparck Jones, 1972). Then we tested some deep learning methods, such as Transformer (Ahmad et al., 2020) or GPT-4. We also innovatively integrated the TF-IDF encoding block into the transformer model. We compared the performance of these methods and analyzed the results.

2 Methodology

Tradition code summarization methods include domain-specific methods, probabilistic grammars, and n-gram language models, and simple neural networks (Le et al., 2020). Recently, the Encoder-Decoder model (Sutskever et al., 2014) has been widely used in code summarization. The Encoder-Decoder model is a sequence-to-sequence model,

```python
def fibonacci(n):
    """A function that returns the nth value in the Fibonacci sequence using recursion."""
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Listing 1: Example input and output of code summarization.

where the encoder and decoder are two separate neural networks. The encoder encodes the input sequence (source code) into a vector, and the decoder decodes the vector into the output sequence (summary). The encoder and decoder are usually implemented with recurrent neural networks (RNNs) or transformers (Vaswani et al., 2017). State-of-the-art code summarization models also integrate graph structure (e.g., abstract syntax tree) (LeClair et al., 2020), file context (Bansal et al., 2023b), and biosignal data (Bansal et al., 2023a) into the encoder-decoder model.

In this section, we will introduce our exploration of the techniques of code summarization, which includes both traditional methods and deep learning methods. The traditional methods are based on the vector space model, which is a classic method in information retrieval. The deep learning methods are one vanilla transformer in the original paper. We also tried an innovative modification, i.e., integrating TF-IDF encoding into the transformer. Finally, we tried the GPT-4 model with a few-shot learning prompt.

2.1 Vector Space Model

We explored two variants of the Vector Space Model (VSM) (Haiduc et al., 2010) as baseline models, which is a classic technique to understand natural languages (Salton, 1989). The key idea of VSM is to represent the source code as vectors in a vector space. For the corpus of source code, we first construct a matrix, where each row is a sample of code (i.e., document), each column is a unique token in the vocabulary. Thus, each row can be represented as a vector of the code sample. For generating the summary using VSM, the terms in the code are ranked by their weights in the matrix, and then the top-k terms are selected as the summary. Regarding the selection of k, we use a linear regression model to predict the length of the summary y based on the length of the code x. The linear regression model is also trained on the training set.

The first variant of VSM is to randomly select k terms from the code as a summary, i.e., VSM with Random sampling. It assigns the same weight to each term in the code. The second variant of VSM uses the TF-IDF (Sparck Jones, 1972) to assign weights to the terms in the code, then selects the top-k terms as the summary, which is donated as VSM with TF-IDF sampling. TF-IDF is abbreviated from Term Frequency-Inverse Document Frequency, which is a numerical statistic that is intended to reflect how important a word is to a document in a corpus. The TF-IDF is calculated as follows:

$$w_{i,j} = tf_{i,j} \times \log(N/df_i) \tag{1}$$

where i is the term (i.e., token), j is the document (i.e., code sample). The $tf_{i,j}$ is calculated by the number of occurrences of term i in document j, and is normalized by the total number of terms in d. The IDF is calculated by the number of documents N in the corpus divided by the number of documents where the term i appears, and is further log-transformed. TF-IDF assigns a higher weight to terms that appear more frequently in the document but less frequently in the corpus.

It is worth noting that the summary generated by VSM is only some randomly ordered tokens, which is not a valid sentence. Its idea is only highlighting the important terms in the code, which is a super simple baseline for code summarization.

2.2 Transformer

The transformer is a deep learning model proposed by Vaswani et al. (Vaswani et al., 2017). It is a sequence-to-sequence model that uses the attention mechanism to capture the long-range dependencies in the input sequence. It is originally used for machine translation, but we just use it to do code summarization, i.e., translating the code into natural language. The transformer is a stack of encoder and decoder layers, which are composed of multi-head attention and feed-forward layers. The encoder takes the code as input, and the decoder takes the summary as input.

The transformer is trained to minimize the cross-entropy loss between the predicted summary and the ground truth summary, which uses masking to prevent the model from seeing the future tokens. The transformer also employs a positional encoding block to encode the position information of tokens in the input sequence.

2.3 Transformer with TF-IDF

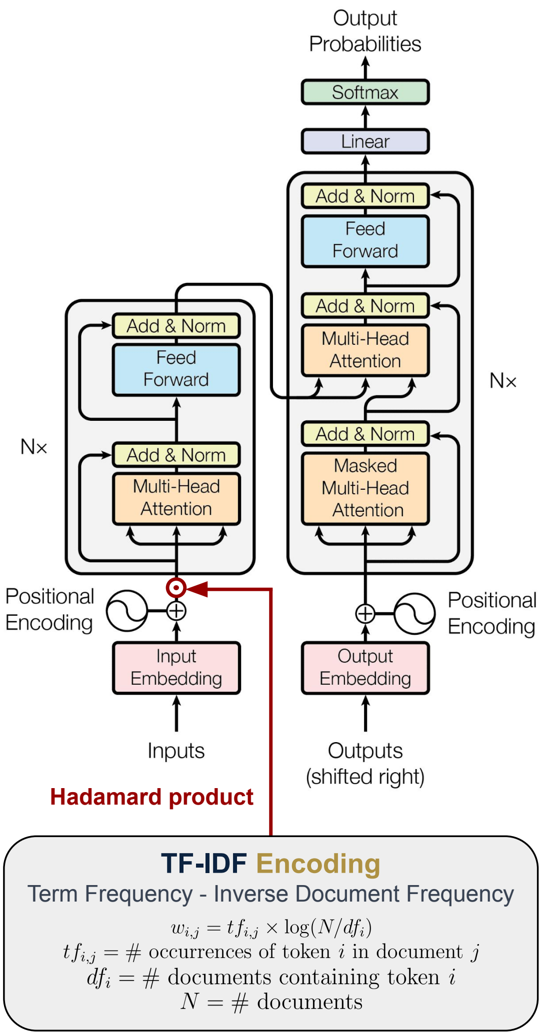

Figure 1: The architecture of the transformer with TF-IDF encoding.

Based on the transformer, we proposed a novel modification, i.e., integrating TF-IDF encoding into the transformer. The intuition is that the TF-IDF score could represent the importance of each token in the input, and we want the model to pay more attention to the important tokens. The architecture of the transformer with TF-IDF encoding is shown in Figure 1. The TF-IDF encoding block is inserted before the input embedding layer. It computes the TF-IDF score of each token in the given input, and multiplies the normalized TF-IDF score with the input embedding.

2.4 GPT-4

GPT-4[1] is a large-scale transformer-based language model, which is the largest model ever trained by OpenAI. It is trained on a large number of web pages, books, and other sources of text on the internet with the goal of predicting the next word in a sentence. It is a general-purpose language model that can be used to perform a variety of tasks, such as question answering and summarization.

In this project, we used GPT-4 to generate code summaries. We used the technique of few-shot learning (Wang et al., 2020) to construct prompt. Few-shot learning is a machine learning technique that aims to learn a new task from a small number of examples. Technically, we randomly sampled k code snippets from the training set, and concatenated them as a prompt. Then, we asked the GPT-4 to generate the summaries for the code snippets in the test set.

We constructed a prompt as follows:

> *You are a software engineer working code summarization. Given a set of code snippets, you need to write a short summary for each of them in natural language. For example,*
>
> *Code snippets:*
>
> *[SAMPLED CODE SNIPPETS FROM TRAINING SET]*
>
> *Summaries:*
>
> *[CORRESPONDING SUMMARIES FROM TRAINING SET]*
>
> *Now, it is your turn. Please follow the example above, just write a short summary with all words in lower case. The punctuation marks should be separated from the words. For example, 'returns true if less then 5 % of the available memory is free .' is a good summary. Do not add any extra things.*
>
> *Code snippets:*
>
> *[CODE SNIPPETS FROM TEST SET]*
>
> *Summaries (You need to write):*

[1]https://openai.com/research/gpt-4

3 Experimental Setup

In this section, we describe the experimental setup for our experiments, including the dataset, evaluation metrics, and baseline models.

3.1 Dataset

We conducted our experiments on a Java dataset `tlcodesum` (Hu et al., 2018) and a Python dataset `python-method` (Wan et al., 2018), which are also used in several previous works (Wei et al., 2019; Ahmad et al., 2020). The statistics of the two datasets are summarized in Table 1, where Sum. is the abbreviation of Summary, Avg. is the abbreviation of Average, and Len. is the abbreviation of Length.

Table 1: Dataset Statistics

Dataset	`tlcodesum`	`python-method`
Records	87,136	92,545
Code Tokens	10,470,110	4,440,193
Sum. Tokens	1,544,578	877,626
Code Vocab.	36,202	159,968
Sum. Vocab.	28,047	27,197
Avg. Code Len.	120.16	47.98
Avg. Sum. Len.	17.73	9.48

`tlcodesum` contains Java methods extracted from 2015 to 2016 Java GitHub repositories. The comments, which are the first sentences of the JavaDoc, are used as ground-truth summaries. The methods with empty or just one-word comments are removed. The setter, getter, test methods, and overridden methods are also removed, since whose comments are easy to predict.

`python-method` is initially collected by (Barone and Sennrich, 2017) from GitHub repositories. The docstrings are used as the ground truth summaries. Semantically irrelevant spaces and newlines are removed from the docstrings. As mentioned in (Wan et al., 2018), the Python code is tokenized by . , " ' : ;) (! and space, and the summaries are tokenized by space.

Thus, the code and summary are converted into sequential text, which can be used as the input of the models. Following previous work (Wei et al., 2019), each data set is divided into training, validation, and test sets with the ratio of 8:1:1. The source code tokens of the form CamelCase in Java and snake_case in Python are further divided into separate subtokens from (Ahmad et al., 2020), significantly reducing the vocabulary size. We built separate vocabularies for the source code and summary.

Since a small portion of the code/comments are too long and seem to be wired (e.g., the comment is just a copy of the code), the cuda memory is not enough to train such pieces of data, i.e., large input embeddings. Thus, we filter out those data that are too long. The threshold is set as 1600 tokens, which larger than 99.9% quantile of the distribution of code snippets length. Thus, less than 0.1% of the data are filtered out but the training process can be finished.

3.2 Evaluation Metrics

Since the summary generated by VSM is not a valid sentence, we use **Precision** and **Recall** of the terms in the generated summary compared to the ground truth summary as the evaluation metrics. Precision is calculated by the number of tokens in the generated summary that also appear in the ground truth summary divided by the number of terms in the generated summary. Recall is divided by the number of terms in the ground truth summary.

Except for those two metrics, we also use **BLEU** (Papineni et al., 2002), **METEOR** (Banerjee and Lavie, 2005), and **ROUGE-L** (Lin, 2004) as the evaluation metrics following the previous works (Ahmad et al., 2020). Although the sequence order of generated summary is not considered in the VSM, we still use those metrics, since they are widely used in code summarization tasks. They are implemented using the `nltk` (Loper and Bird, 2002) and `rouge` (Lin, 2004) packages.

3.3 Baseline Models

We take VSM, vanilla transformer, and GPT-4 as our baseline models. Regarding the selection of k of VSM, we found that the length of the summary is positively correlated with the length of the code. The Pearson correlation coefficient is 0.1156 (p-value < 0.0001) and 0.0570 (p-value < 0.0001) for the `tlcodesum` and `python-method`. We use a linear regression model to predict the length of the summary y based on the length of the code x. The result is as follows:

$$y_{Java} = 0.0143 \times x_{\text{Java}} + 16.0064 \quad (2)$$

$$y_{Python} = 0.0067 \times x_{\text{Python}} + 9.1413 \quad (3)$$

About the hyperparameters of the transformer, we refer to setting as the original paper (Vaswani et al., 2017) and homework 2 of the course[2]. The encoder has 4 layers and the decoder has 1 layer. The number of heads is 4, the hidden size is 256, the feed-forward network size is 1024, the dropout rate is 0.1. We use the Adam optimizer with initial learning rate of 0.0003, β_1 of 0.9, β_2 of 0.98, and ϵ of 10^{-9}. The batch size is 16 because of the limitation of the GPU memory. We apply padding `<pad>` to the input and output sequences to make them have the same length. The model is trained for 10 epochs and the best model is selected based on the validation loss. We would like to thank the PyTorch tutorial "Language Translation with `nn.Transformer` and `TorchText`"[3] for the code reference of implementing transformer.

We also refer to homework 5 of the course[4] to implement three sampling methods of the transformer:

- **Greedy sampling:** select the token with the highest probability at each decoding step.
- **Top-*k* sampling:** select the top-k tokens with the highest probability at each decoding step, and sample from them with normalized probability. The default value of k is 100.
- **Ancestral sampling:** sample from the probability distribution at each decoding step.

About the few-shot learning of GPT-4, we use five examples to prompt the model to generate the summaries for the test dataset. The examples are randomly selected from the training dataset. For convenience, we just generate the first 100 lines of the test dataset for evaluation.

4 Experimental Results

In this section, we present the results of our experiments. We first compare the performance of different models on the two datasets under different metrics. Then, we analyze one example output of each model to understand the strengths and weaknesses of them.

4.1 Performance Comparison

Table 2 shows the results of the code summarization models on the `tlcodesum` and `python-method` datasets. The best results are highlighted in bold.

The results show that the VSM with TF-IDF sampling outperforms the VSM with Random sampling on all the evaluation metrics, which indicates that the TF-IDF actually helps to select the important tokens in the code. However, the performance of VSM with TF-IDF sampling is still far from the state-of-the-art models, i.e., GPT-4, and it lacks the inherent ability to generate valid sentences.

The results show that the Transformer with TF-IDF generally underperforms compared to other Transformer models. It usually scores lower in different evaluation metrics. However, in some cases, especially when using the greedy method, the Transformer with TF-IDF surpasses the original Transformer models. This is observed in the Python dataset, indicating that TF-IDF's performance varies with context and might be better in certain applications.

One reason that the baseline methods have higher BLEU scores but lower METEOR and ROUGE scores is that the baselines always generate much shorter summaries compared to the transformer methods. BLEU is a word-precision-based metric, but METEOR and ROUGE more emphasize the semantics of the summary. For the baselines, we used a linear regression model to predict the length of the summary, which on average is about 10. However, for the transformer methods, the length of the summary is set as at most 100 tokens, except for generating the EOS token.

The result shows that greedy sampling generally outperforms both the top-k and ancestral sampling, suggesting that the greedy sampling has distinct advantages in text generation tasks. The greedy sampling, by design, selects the most likely next word at each step in the generation process. This approach tends to produce more coherent and contextually appropriate outputs, as it consistently chooses the highest probability option. This consistency is likely a key factor in its superior performance. The results indicate that, in most cases, this method yields outputs with higher precision, indicating a greater relevance of the generated text to the given context.

In contrast, the top-k sampling introduces a degree of randomness and diversity in the generated text. While this can sometimes lead to more creative or varied outputs, it appears that this randomness often comes at the cost of overall coherence and relevance, as reflected in lower performance metrics compared to the greedy method. Similarly, the ancestral sampling also introduces variability in

[2]https://www3.nd.edu/~dchiang/teaching/nlp/2023/hw2.html
[3]https://pytorch.org/tutorials/beginner/translation_transformer.html
[4]https://www3.nd.edu/~dchiang/teaching/nlp/2023/hw5.html

Table 2: Performance Comparison of Different Methods

	Method	Sampling	Precision	Recall	BLEU	METEOR	ROUGE-L
`tlcodesum`	VSM	Random	0.1323	0.1802	0.1802	0.0982	0.0967
		TF-IDF	0.1524	0.2142	0.2232	0.1199	0.1195
	Transformer	Greedy	0.3465	0.2205	0.1311	0.1477	0.1947
		Top-k	0.2283	0.2171	0.1382	0.1315	0.1407
		Ancestral	0.2028	0.1965	0.1205	0.1156	0.1188
	Transformer with TF-IDF	Greedy	0.2617	0.1764	0.1032	0.0987	0.1243
		Top-k	0.1936	0.1624	0.0983	0.0902	0.1002
		Ancestral	0.1637	0.1465	0.0893	0.0789	0.082
	GPT-4	-	**0.3234**	**0.3122**	**0.2721**	**0.2116**	**0.24**
`python-method`	VSM	Random	0.1081	0.113	0.1869	0.0691	0.0923
		TF-IDF	0.1412	0.1493	0.2175	0.0874	0.1209
	Transformer	Greedy	0.2137	0.1916	0.0682	0.0955	0.1274
		Top-k	0.2499	0.2315	0.0932	0.1294	0.0978
		Ancestral	0.222	0.206	0.0731	0.1093	0.0699
	Transformer with TF-IDF	Greedy	0.3189	0.244	0.0885	0.1452	0.132
		Top-k	0.2283	0.2158	0.0763	0.1154	0.0816
		Ancestral	0.2125	0.1984	0.0662	0.1039	0.0628
	GPT-4	-	**0.2101**	**0.2307**	**0.2673**	**0.1985**	**0.204**

the text generation process. This approach, while potentially capturing a wider range of linguistic possibilities, seems to be less effective in consistently generating high-quality, contextually relevant text compared to the greedy sampling.

Regarding the output of GPT-4, it consistently emerges as the top performer across all evaluated metrics. Its superiority in various aspects of text generation - coherence, relevance, fluency, and adherence to the context - establishes it as a reliable standard in the field. The consistent excellence of GPT-4's output across diverse datasets and metrics highlights its advanced language understanding and generation capabilities. This consistency underscores the effectiveness of large-scale language models in handling complex and varied natural language processing tasks, cementing their place as a benchmark in the field.

4.2 Example Outputs

Listing 2 shows an example source code and its corresponding reference summary from the test set of `python-method` dataset. The target of this code snippet is to "store a temporary file ." The example output of each model is shown in Table 3. The GPT-4 model generates the best summary, which even better than the reference summary with more details like "returns its path". However, GPT-4 uses the word "stores" instead of "store", which is the third-person singular form of the verb. Such a problem is meaningless for the human judgment but will be penalized by the word overlap based metrics such as BLEU.

The outputs of the VSM models are unreasonable in terms of the syntax. The VSM with TF-IDF sampling just samples the tokens with the highest TF-IDF scores without considering the order of the tokens, which may reveal the importance of the tokens but is not a reasonable summary. The VSM with random sampling model is even worse. The outputs of the Transformer models are reasonable in terms of the syntax. However, in the example, the Transformer model only generates the last half of the summary from GPT-4. The Transformer with TF-IDF model generates more details than the Transformer model, but the summary is still not as good as the GPT-4 model. However, they did not capture the tokens in the reference summary.

Despite sometimes lagging behind the VSM in numerical NLP metrics, the output of the Transformer with TF-IDF is often more reasonable from a human evaluation standpoint. This discrepancy shows a limitation in current NLP metrics, as they may not capture all qualitative aspects of text generation that human evaluators notice. It implies that although these metrics are helpful for quantitative assessment, they may not fully represent the nuances and contextual appropriateness that a human reviewer can identify. Previous study (Haque et al., 2022) also found that the semantic similarity metrics such as the cosine similarity between the

```python
def store_temp_file(filedata, filename, path=None):
    """ store a temporary file . """
    if not filename:
        filename = get_filename_from_path(filename)
    if len(filename) > 100:
        filename = filename[:100]
    options = Config()
    if path:
        target_path = path
    else:
        tmp_path = options.cuckoo.get('tmppath', '/tmp')
        target_path = os.path.join(tmp_path, 'cuckoo-tmp')
    if not os.path.exists(target_path):
        os.mkdir(target_path)
    tmp_dir = tempfile.mkdtemp(prefix='upload_', dir=target_path)
    tmp_file_path = os.path.join(tmp_dir, filename)
    with open(tmp_file_path, 'wb') as tmp_file:
        if hasattr(filedata, 'read'):
            chunk = filedata.read(1024)
            while chunk:
                tmp_file.write(chunk)
                chunk = filedata.read(1024)
        else:
            tmp_file.write(filedata)
    return tmp_file_path
```

Listing 2: An example of source code and corresponding summary from the test set of python-method dataset.

Table 3: Example Outputs of Different Methods

Method	Summary
Reference	store a temporary file .
VSM (Random)	[path os config def
VSM (TF-IDF)	1024 file target path tmp
Transformer (Top-k)	read the path of a file .
Transformer with TF-IDF (Top-k)	return list of the file path .
GPT-4	stores a temporary file and returns its path .

BERT embeddings are more correlated with human judgment than the word overlap based metrics. We plan to explore them in the future work.

5 Conclusion

In this project, we explored both traditional and deep learning methods for code summarization. We also proposed a novel modification of the original transformer, which is the TF-IDF encoding block. We conducted experiments on two datasets, and evaluated the performance of different methods using five metrics. The results show that the GPT-4 model performs the best on both datasets and all metrics. However, the TF-IDF encoding block does not improve the performance of the Transformer.

We have several takeaways from this project. Firstly, there is the mismatching between word overlap based metrics and human judgment, especially the BLEU score. Exploring more metrics such as semantic similarity metrics that are more consistent with human judgment is necessary. Secondly, there is a high correlation between code and summary due to numerous identical tokens, which is quite different from the machine translation task, for which the original transformer was designed. Third, integrating prior knowledge into model training is challenging. Although we think the TF-IDF encoding block is reasonable since it could represent the importance of each token in the input, the performance is not ideal. Lastly, GPT-4 is always the best, which is not surprising since it is a state-of-the-art model. But one issue is that the training data is not available to the public, which may include the data leakage from the test set, thus the comparison is somewhat unfair.

Furthermore, we spent much time debugging the mask settings of the decoder in the transformer model. We believe the experience of implementing the model by hand is helpful for us to understand the details of the model and the training process.

6 Reproducibility

The data and code of our project are available at https://github.com/TTangNingzhi/code-summarization for reproducibility.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Aakash Bansal, Bonita Sharif, and Collin McMillan. 2023a. Towards modeling human attention from eye movements for neural source code summarization. *Proceedings of the ACM on Human-Computer Interaction*, 7(ETRA):1–19.

Aakash Bansal, Chia-Yi Su, and Collin McMillan. 2023b. Revisiting file context for source code summarization. *arXiv preprint arXiv:2309.02326*.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

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

Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 36–47.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge.

Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3):1–38.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Edward Loper and Steven Bird. 2002. Nltk: The natural language toolkit. *arXiv preprint cs/0205028*.

Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, pages 390–401.

Gerard Salton. 1989. Automatic text processing: The transformation, analysis, and retrieval of. *Reading: Addison-Wesley*, 169.

Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, pages 174–188.

Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21.

Jamie Starke, Chris Luce, and Jonathan Sillito. 2009. Searching and skimming: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 157–166. IEEE.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

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 the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407.

Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.