Testing the Performance of Automated Documentation Generation with Included Inline Comments

Balys Morkūnas
Supervisor(s): Annibale Panichella, Leonhard Applis
EEMCS, Delft University of Technology, The Netherlands

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

A number of Machine Learning models utilize source code as training data for automating software development tasks. A common trend is to omit inline comments from source code in order to unify and standardize the examples, even though the additional information can capture important aspects and better explain algorithms. We claim that models, utilizing the supplementary data, are able to produce more fluent translations for Automatic Documentation Generation task. We test this by creating two datasets and measuring the performance difference. The results show that there is a slight improvement in translation accuracy when a dataset contains inline comments, with stop words removed. Further research needs to be done to optimize the preprocessing of data and to more accurately detect the scope of inline comments.

1 Introduction

During software development, code documentation is an essential part of the process. Natural language descriptions reduce the time needed to understand and maintain code, as well as lower the chance of code defects [1]. Current advancements in Artificial Intelligence make Automatic Documentation Generation possible by utilizing Neural Machine Translation [2], Deep Reinforcement Learning [3], Natural Language Processing [4], [5], or Keyword Discovery [6], [7] techniques. Ongoing research explores the possibilities to allow developers to be more resourceful and optimize workflows by skipping manual documentation. Aforementioned techniques rely on analysing context-free grammars or by training models with data to make predictions.

An important observation, common throughout many models, is that commented out code, inline comments, or unreachable code are removed from the training examples. Models omit such information to unify and standardize data, even though these additions capture important aspects and sometimes can be crucial to understand the inner workings of an algorithm. We are specifically interested in the value that inline comments bring, as they often contain keywords that elaborate on ambiguous lines of code.

This work aims to measure and understand the impact that inline source comments have on the performance of one of the aforementioned models. More precisely, sequence-to-sequence model *code2seq* [2]. The model uses Abstract Syntax Trees (ASTs) to represent and analyse code structures. It was developed as a tool to generate natural language sequences given a piece of code and in addition to Code Documentation can also be used for Code Summarizing and Code Captioning. The focus of this work is Code Documentation, the hypothesis to be tested is formulated as follows:

Inline source code comments increase the performance and accuracy of machine learning models for Automatic Documentation Generation.

To test the impact of code comments, the training data is specifically encoded. Code and comments were tokenized and grouped based on specific rules to create ASTs. The tree, capturing additional information, is then used for model training. The performance of the trained model is compared to the original *code2seq* model by using BLEU [8] and F1 [9] scores, that evaluate machine translation and accuracy, respectfully.

The rest of this research paper will be organized as follows. Section 2 explains the relevant topics required to understand the research, Section 3 gives the methodology of the experiment, and Section 4 presents the obtained results. Interesting work, utilizing inline comments, is given in Section 5. The results are then discussed in Section 6. Finally, Section 7 raises ethical considerations and Section 8 concludes the work.

2 Background

This section focuses on explaining ASTs in the context of *code2seq* and the process it uses to generate documentation. The model solves a Neural Machine Translation (NMT) [10] problem, where the input is source code and the output is a natural language sequence, describing the input.

2.1 Abstract Syntax Trees

An AST represents a code snippet with terminal and non-terminal nodes. The former nodes abstract user defined values and are the leaf nodes in a tree, the latter, non-leaf nodes, are representations of structures in the given language, e.g., if statements, loops, etc. The applications for ASTs are broad and are used for compiler development, code duplication detection, and other static analysis tasks. A number of programming languages utilize ASTs when converting source code to machine code, for this reason all comments are removed and most AST implementations do not support comment recognition. While our focus is only on representing code, ASTs are also used to represent other types of structured data.

Figure 1 gives an example of a simple Java method and its corresponding AST. The terminal nodes are represented with boxes, while the non-terminals are in ovals. The *code2seq* model then uses the generated tree to produce paths between terminal and non-terminal nodes.

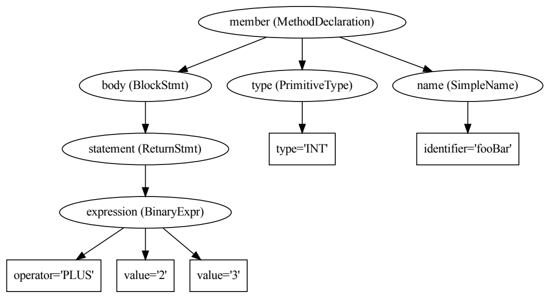
The ability to organize and attach comments is important to our research. We aim to take the generated ASTs and supplement them with inline comments. The original preprocessing implementation will have to be altered, as it currently ignores any type of comments.

2.2 Neural Machine Translation

NMT was initially designed to improve Deep Neural Networks for human language translation. It uses an artificial neural network to predict the likelihood of a sequence of words, like sentences. A popular application of NMT is Google Translate. In this scenario, *code2seq* utilizes the same methods to translate code samples into natural language sentences. The NMT approach consists of an encoding and a decoding layer. The encoding layer maps code tokens $\boldsymbol{x} = (x_1, ..., x_n)$ to an intermediate sequence representation $\boldsymbol{z} = (z_1, ..., z_m)$. The decoder generates an output sequence $\boldsymbol{y} = (y_1, ..., y_k)$ given $\boldsymbol{z}$.

```java
public int fooBar() {
    return 2 + 3;
}
```

(a) Java method.



(b) Corresponding AST.

Figure 1: Java method and an Abstract Syntax Tree representing the methods structure.

The *code2seq* model aims to solve an NMT problem and follows an encoder-decoder architecture [11]. The model prepares the input code by generating an AST and extracting paths between terminal and non-terminal nodes, a generated path is denoted by $\boldsymbol{x}$. For the encoding, *code2seq* uses a bidirectional Long-Short Term Memory (LSTM) [12] to create vector representations for each generated path. LSTMs are a special kind of Recurrent Neural Networks (RNNs) [13], capable of learning and remembering dependencies between long sequences. By having an additional internal state, LSTMs are able to weigh the importance of observed information and discard or keep only what is contextually relevant. The LSTM nodes take a path $\boldsymbol{x}$ and encode it as $\boldsymbol{z}$. Each path element can be represented by a learned embedding matrix, which the LSTM uses to encode the entire sequence. The use of LSTMs are important for this research, as Automatic Documentation Generation attempts to produce relatively long sequences.

Finally, an LSTM decoder generates the output sequence in natural language by using an attention model [14], [15]. Decoding with attention means that each part of the input has an attention weight. The weights denote how relevant the part of the input is for the current decoding step in the given context. Instead of having a single context vector, generated from the encoders last hidden state, attention allows for the decoder to focus on parts that are the most important when generating output. The decoder *attends* the output $\boldsymbol{z}$ of the encoder to generate the final output $\boldsymbol{y}$. For formal and more concrete implementation details, please refer to the *code2seq* paper [2].

3 Methodology

The process to test the hypothesis can be split into four steps. The first step concerns the data used for training the model. The second step is the preparation of the data. The third and fourth steps are model training and evaluation.

3.1 Dataset

The dataset used for this experiment was taken from the CodeSearchNet project [16]. This dataset was chosen as a replacement because the majority of the original *code2seq* data did not have a corresponding Javadoc comment. The CodeSearchNet dataset consists of $454k$ (*comment*, *code*) pairs in the Java programming language. More specifically, a *comment* is a top-level function or method Javadoc description, and *code* is the function or method that the *comment* belongs to. Table 1 provides statistics for the used CodeSearchNet dataset. It shows that out of all training examples, around $109k$ contain inline comments. Furthermore, the calculated length statistics for *code* and *comment* examples are illustrated in Table 2. The mode and median metrics are calculated by counting the number of words in each example.

It is important to mention that only the first sentence of the Javadoc comment was used as a label, as it typically summarizes the functionalities of the methods. This decision is based on the official Javadoc documentation[1]. HTML tags, line separators, comments with only a single word are removed. The data is partitioned into training, validation, and testing sets. Lastly, the gathered data was only taken from open source libraries.

3.2 Preprocessing

During the preprocessing step, we had to decide on how the inline comments will be encoded into the *code* part of the training data. Using a Java parsing library[2], we classify observed comments into inline, and block comments. The comments are stripped of any stop words (e.g., *the, and, but*) as they bring no important information. We check each comment string against a list of most popular stop words[3]. Additionally, we check to omit commented out code, as such information is not the focus of this experiment and could influence the performance in unforeseen ways. We search for commented code by matching a regex string. If the string starts with an open comment symbol and ends in a semi-column, it is classified as code. In addition, we look for keywords like *void*, *int*, *string*, to capture lines that do not end in a semi-column.

Since the library is able to generate an AST from the given input, it is possible to attach the inline comments to specific nodes. This is important since the generated paths would make little sense when they would include the comment but not the node it refers to. The Java parser library assumes that the inline comment node parent is the code below the comment or the code on the same line, if any. Some inline comments, that are not above or on the same line as code, are classified as orphan comments, meaning they exist without a parent node. As the model generates paths between connected nodes, orphan comments cannot be utilized.

[1]https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

[2]https://javaparser.org/

[3]https://www.computerhope.com/jargon/s/stopword.htm#basic

Table 1: Dataset example statistics.

Pairs	**Inline Comments**	**Unique Code Tokens**	**Unique Comment Tokens**
454,451	109,457	2,042,229	345,202

Table 2: Raw code and comment length statistics.

Code			
Avg Lines	Avg Words	Mode	Median
17	114	37	67

Comments			
Avg Lines	Avg Words	Mode	Median
5	45	4	29

Figure 2 illustrates how an inline comment attaches to an AST node. The comment, marked in blue, is a direct child node of the return statement. While this is a primitive example, it is easy to see that when generating paths between terminal and non-terminal nodes, the model training data is complimented with additional keywords. To accommodate the supplementary information, we increase the maximum length of a path in case a comment node extends the depth of a tree.

3.3 Training

As the main task that *code2seq* solves is method name prediction (Code Summarization), the model first had to be reconfigured to take in Javadoc comments as target labels instead of method names. Previously, the preprocessing step would replace the original method name with a temporary masking keyword. In our case, we keep the method name and alter the program to add the Javadoc comment as the target label.

After the preprocessing, we are able to train the model. The training was done on a high-performance computer containing Intel XEON E5-6248R 24 Core 3.0 GHz processor and an NVIDIA Tesla V100S 32 GB graphics card. It took around ten hours for the CodeSearchNet Java set to converge. Converging is defined as performance not increasing for ten consecutive training epochs.

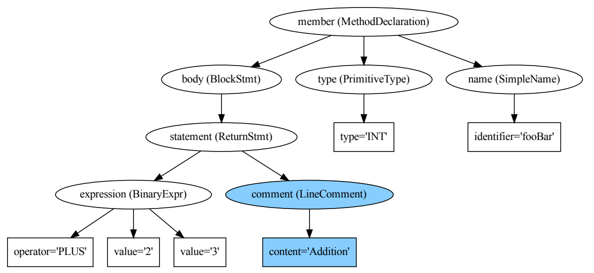
It is important to mention that we also prepare a baseline model with the original *code2seq* preprocessing script and a model that contains inline comments with stop words included to act as comparison units. Both trained on the CodeSearchNet dataset.

3.4 Evaluation

To evaluate and measure the difference between a model with inline comments and without, we use two metrics. The following subsections will explain the BLEU and F1 scores in more detail. Furthermore, we explain two statistical tests to see if the observed results are significant.

```java
public int fooBar() {
    // Addition
    return 2 + 3;
}
```

(a) Java method with an inline comment.



(b) Corresponding AST.

Figure 2: Java method with an inline comment and an Abstract Syntax Tree representing the method's structure.

BLEU

Bilingual Evaluation Understudy is a method used to compare a machine produced candidate translation with an existing human reference sequence. BLEU is a popular metric and often used for evaluation of NMT tasks [17]. The main idea behind BLEU is to count the matching n-grams in the machine produced and human reference sequences. The n-gram denotes the granularity of the sequences, where $n = 1$ means that each token is matched and $n = 2$ mean that each word pair is matched. The order of the produced sequence is not relevant for the BLEU score.

The BLEU score is calculated by using the following formula:

$$BLEU = BP \times exp\left(\sum_{n=1}^{N} w_n \, log \, p_n\right) \tag{1}$$

The BP stand for Brevity Penalty, which is a value that depends on the difference of the generated and reference sequence lengths. If c and r are the lengths of the candidate and reference translations, then the value of BP is:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{2}$$

Furthermore, w_n is the n-gram weight calculated as $1/n$, and p_n is the ratio of the number of subsequences of length n in the candidate translation that are also in the reference string. As the generated sequences are relatively long, we have selected the value of n to be equal to 4.

F1

The F1 score is a combination of two other scores – precision and recall. Precision is defined as the ratio between the number of shared words, separated by spaces, in two sequences and the total amount of words in the generated sequence. Recall is the ratio between the number of shared words and the total amount of words in the reference sequence. Combining the two, F1 measures the harmonic mean of precision and recall and is calculated with the following formula:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \quad (3)$$

We calculate precision and recall by tokenizing the generated sequence and checking if the tokens are contained in the original string. This is done so that the order of the tokens would not be a confounding factor in the calculations. The reason we decided to include the F1 measure is that by having supplementary information, in the form of inline comments, might lead to higher recollection. Additionally, the wider range of context options might reduce the precision of the model.

Wilcoxon Rank Sum Test & Cliff's Delta Effect Size

The Wilcoxon Rank Sum test [18] is a non-parametric, statistical test to compare two groups and prove if there is significant difference between their distributions. The tested null hypothesis is that two populations have the same distributions. We reject the null hypothesis when we have sufficient evidence that one distribution is shifted, implying the difference in populations and significance in findings. The test requires responses to be ordinal, and it assumes that the data from both groups is independent and continuous. Our data samples are obtained by independently training and evaluating the models. The test allows us to compare and make claims about the observed differences in results.

The Cliff's Delta [19] measures the number of times a value from one group is larger than one from the other group. Values closer to ± 1 imply that there is little overlap, and values closer to 0 show strong overlap between the given sample distributions. We use the result of Cliff's Delta to compliment the null hypothesis test and help quantify the size of the difference, beyond the calculated p-values.

4 Results

The section presents and discusses the obtained results of the experiment. To begin, we give an overview of the experimental setup. Then, the scores together with statistical analysis are presented. Lastly, interesting examples that portray how inline comments affect the generated sequences are provided.

4.1 Setup

By following the process in Section 3 we prepared the target labels of the dataset by taking the first sentence, removing unicode symbols, and process the code to leave or remove inline comments, etc. We create three datasets, called `NOCO` for a baseline, no inline comment model, `ICOS` for inline comments with stop words left in, and `ICO` for a dataset with processed inline comments. These datasets are used to train their corresponding models. We limit the training to produce comments of thirty seven words maximum and use four LSTM encoders and two LSTM decoders, as per the recommendations of the *code2seq* authors[4]. Additionally, the maximum amount of contexts was left unchanged and set to 100, the training algorithm samples a different subset of this size at each epoch. Results presented in the following subsections were calculated using the test set.

Table 3: Result comparison of with and without comment models. BLEU is measured from 0 to 100 and the rest are from 0 to 1.

	BLEU	**Precision**	**Recall**	**F1**	**Time**
`NOCO`	6.22	0.428	0.387	0.406	11h
`ICOS`	5.97	0.437	0.370	0.400	**8h**
`ICO`	**6.28**	**0.450**	**0.391**	**0.418**	12h

4.2 Scores

Table 3 portrays the results achieved by the two models. The `ICO` model, containing inline comments, slightly outperformed the `NOCO` model for both the BLEU and F1 measures. We use the Wilcoxon Rank Sum test to better understand the underlying meaning of the results and identify if there are differences in the distributions. To apply the test, we calculated Jaccard distance and BLEU score between the predicted and reference test set sequences for both models. Both tests returned a p-value lower than 0.05. The p-values indicate that we can reject the null hypothesis and state that there is significant difference in the distributions of the model results. The Cliff's delta, however, suggests that the significance is weak, as both tests returned a value close to zero. It is important to note that the model with inline comments and no stop word filtering (`ICOS`) performed noticeably worse than both of its counterparts. The lower BLEU scores suggests that the stop words only populate the possible options for the model to choose from, but provide no meaning to the overall generated sequence.

The results show a slight improvement in model performance and accuracy with included inline comments, given that the stop words are removed (BLEU of 6.28 compared to 6.22). It must be noted that even though the models achieved results that were significant, compared to previous research, the BLEU score is low. Alon et al. [2] achieve a BLEU score of 14.53 for Documentation Generation task. This is reflected in a considerable amount of incomprehensible generation attempts by our model.

4.3 Generated Examples

To better visualize the differences between comments, we provide examples in Table 4. The first column is the original documentation string, written by a human, the following two columns are generated by the `ICO` and the `NOCO` models. Overall, it is evident that both models are able to pick up the general meaning of the given input and produce a somewhat understandable sequence. In some cases, the model with inline comments is able to better depict the meaning of the code sequence as compared to the model without. It is also impor-

[4]https://github.com/tech-srl/code2seq/issues/45#issuecomment-624539251

Table 4: Generated documentation examples with author highlights. Green indicates an overlap between the original docstring and the translation. Red indicates meaningless or inaccurate translations.

Original Docstring	**ICO Docstring**	**NOCO Docstring**
`buffer when possible`	`this method is called when the buffer is read`	`reads up to code len bytes from stream`
`gets the children of this directory`	`gets the list of children of the current project`	`returns the list of all children from the directory`
`returns a host specifier built from the provided specifier`	`returns the name of the host`	`create a host from a string`
`returns the innermost cause of code throwable`	`returns the root cause of the given throwable`	`cause the cause of the given throwable`
`returns an unmodifiable view of code iterable`	`returns an unmodifiable view of the specified iterable that wraps the specified iterable`	`returns an iterable that contains the given iterable`

tant to mention that the human created comment is not necessarily meaningful in every example.

A common trend throughout the generated examples is that they are long or contain repeating information. We speculate that this is due to an erroneous hyperparameter set up. The length for generated sequences was set to 37 which is higher than most input labels. This causes the examples to often repeat the same word. Such looping behaviour could also suggest overfitting. Additionally, some target labels contain Javadoc *links*, that associate a comment with other classes or packages and populate the comments with out-of-place keywords (e.g., *code*, *see*).

5 Related Work

This section aims to provide an overview of relevant research done with code comments and place our contribution within the appropriate context. In particular, the described work shapes the field of code comment analysis and gives suggestions on possible applications and methods.

5.1 Code Comments in Software Defect Prediction

Recent work by Huo et al. [20] analyses the impact of code comments on the accuracy of Software Quality Assurance tasks. They claim that analysing only source code to detect defects rarely produces optimal results. The work proposes that code comments bring additional semantic features and can supplement the existing feature extraction methods that predict bugs in software. Their introduced Deep Learning model called CAP-CNN uses Convolution Neural Networks in order to abstract additional comments into training examples. The performance of the model indicates that the extra information improves the results of vulnerability detection and is an improvement over other state-of-the-art methods. While the CAP-CNN model focuses on top-level documentation for additional information, the work is still an indicator that natural language supplements lead to more sophisticated models. Since comments and source code are made up of different structural semantics, the work uses distinct encoders for comments and source code. Our work only adds inline comments as an addition to the code part. Implementing a layer that handles comment feature extraction separately might lead to additional improvements.

5.2 Code Redocumentation

An interesting approach to code redocumentation was suggested by Geist et al. [21]. Code redocumentation is the process of identifying, reevaluating, and refurbishing documentation of a given code base. Such techniques are usually applied on large scale legacy projects, where manual analysis and classification is economically infeasible. The process is also used during code migrations, reengineering, or software maintenance. For such tasks, the information in source code, like inline code comments, is viewed as a valuable resource, by the authors. While not all redocumentation techniques utilize such data due to the additional work required (e.g., annotating legacy software), the authors state that comments are extremely important and carry the intent behind various design choices. The research proposes multiple heuristic and Deep Learning models to classify inline or block comments as useful for the redocumentation task. Being able to measure the relevance or usability of an inline comment could prove useful if implemented in the preprocessing step of our work. Currently, some inline comments are bits of commented code, URLs, or hardly understandable sequences.

5.3 Source Code Comment Scope

By design, Javadoc comments describe the contents of a method or a class that they are related to. Inline and block comments, however, usually appear in more random places throughout the code base. Chen et al. suggest a method for classifying the scope of such comments by identifying comment-code relationships [22]. Using machine learning, the authors are able to outperform heuristic approaches for scope detection. Typical use cases are detecting outdated comments and mining repositories for comment generation. The latter, is specifically of interest to our research, as having numerous inline comments with correctly associated scopes, could lead to higher performance for the *code2seq* model. Currently, comments with unidentified scopes are classified as orphan and omitted from the training examples.

6 Discussion & Future Work

The main objective of this work was to measure the effect of inline comments for software engineering tasks. The results suggest that there is a relatively small benefit, with specific caveats. Mainly that the comments have to be filtered properly, and have stop words removed. Currently, the filtering process is very straightforward and could be improved by utilizing more advanced keyword extraction techniques. The work shows that the keywords that are extracted create more sophisticated context pools for the model to make predictions from. Overall, we recommend to leave inline comments in, assuming there are enough time resources to allocate for filtering.

For future work, the most important aspect to consider is measuring how effective various keyword extraction techniques are for inline comments. Furthermore, more attention must be given to orphan comments. Methods to detect the scope of a comment can be applied in conjunction with this work to obtain a training corpus that contains a higher number of examples with more precise associations. Lastly, the results suggest that the hyperparameters also require adjustments. Measuring the effect of the target label lengths and increasing or decreasing the size of context pools must be done with hyperparameter optimization techniques.

7 Responsible Research

This section includes a discussion on the responsibility aspects of the research. It explores the ethical concerns related to the experiment and describes any validity issues related to the outcome. Finally, the section ends with a discussion on reproducibility.

7.1 Ethical Aspects

From an ethical point of view, it is important to consider how the research impacts the environment and its users. Foremost, the prolonged and expensive computations have an environmental cost. Recent studies show that energy consumption for Deep Learning tasks has a noticeable carbon footprint [23]. One of the mitigations mentioned in the article is equitable access to computational resources. Having a centralized computational unit, shared between researchers, is more cost-effective than renting cloud computing services. Inline with the recommendations, we used a server, provided by the Technical University of Delft, to carry out our experiments. Another worthy consideration regarding energy consumption is that the dataset with inline comments is 7.2% larger than its commentless counterpart. This raises the question if the extra energy for processing and training is worth the 0.9% increase in performance scores.

An ethical aspect related to users is data reuse. By creating a model that automatically generates documentation, it creates a situation where future research might reuse the generated comments for other experiments, constructing a feedback loop. While feedback loops often lead to increased performance, developers need to be wary of any biases that might occur. As this research only spans the field of programming, we do not consider it to be a critical issue.

7.2 Threats to Validity

To strengthen the argument we make, it is important to think about the possible threats that might influence the outcome of the experiment. The most easily identifiable threat is the size of the dataset. The *code2seq* authors used three datasets of varying sizes. The largest was around fifteen million entries, more than 40 times larger than CodeSearchNet dataset. By using a significantly smaller dataset, our observed delta is influenced. Another aspect to be wary of is that we only used a single dataset, made up of Java examples. Expanding the possible range of data might have an impact on the outcome of the experiment.

7.3 Reproducibility

The initial *code2seq* project was forked, and all changes made to the codebase can be observed online[5]. The usage of version control allows for complete transparency regarding the implementation changes and the testing setup. Additionally, the codebase and the dataset are openly available to download and are licenced under the MIT licence. One can reproduce the results by cloning the repository or by following Section 3. The current implementation strives to produce a clean and regular collection of examples.

8 Conclusions

The paper raised the hypothesis that including inline comments in Automatic Documentation Generation model training data would increase the performance and produce more accurate natural language sequences. It formulated the methodology of attaching inline comments for specific nodes of Abstract Syntax Trees (ASTs). The paths from the ASTs were used to train a model that produces an output sentence given source code as input. The hypothesis was confirmed as the model, trained with the additional information, outperformed the baseline by 0.9% and showed significance in difference of distributions. However, including inline comments without any filtering results in worse performance. We encourage other researchers to explore more sophisticated keyword extraction techniques and implement comment scope detection.

[5]https://zenodo.org/record/6659797

References

[1] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The javadocminer," in *International Conference on Application of Natural Language to Information Systems*, Springer, 2010, pp. 68–79.

[2] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX.

[3] 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*, 2018, pp. 397–407.

[4] M. P. Arthur, "Automatic source code documentation using code summarization technique of nlp," *Procedia Computer Science*, vol. 171, pp. 2522–2531, 2020, Third International Conference on Computing and Network Communications (CoCoNet'19), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.04.273. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920312655.

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

[6] 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*, 2014, pp. 279–290.

[7] 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*, 2010, pp. 43–52.

[8] 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 of the Association for Computational Linguistics*, 2002, pp. 311–318.

[9] Y. Sasaki, "The truth of the f-measure," *Teach Tutor Mater*, Jan. 2007.

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

[11] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] S. Dupond, "A thorough review on the current advance of neural network structures," *Annual Reviews in Control*, vol. 14, pp. 200–230, 2019.

[14] R. Desimone and J. Duncan, "Neural mechanisms of selective visual attention," *Annual review of neuroscience*, vol. 18, no. 1, pp. 193–222, 1995.

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

[16] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[17] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," *arXiv preprint arXiv:1701.02810*, 2017.

[18] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945, ISSN: 00994987. [Online]. Available: http://www.jstor.org/stable/3001968 (visited on 2022-06-07).

[19] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions.," *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[20] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *2018 IEEE international conference on data mining (ICDM)*, IEEE, 2018, pp. 1049–1054.

[21] V. Geist, M. Moser, J. Pichler, R. Santos, and V. Wieser, "Leveraging machine learning for software redocumentation—a comprehensive comparison of methods in practice," *Software: Practice and Experience*, vol. 51, no. 4, pp. 798–823, 2021.

[22] H. Chen, Y. Huang, Z. Liu, X. Chen, F. Zhou, and X. Luo, "Automatically detecting the scopes of source code comments," *Journal of Systems and Software*, vol. 153, pp. 45–63, 2019.

[23] E. Strubell, A. Ganesh, and A. McCallum, *Energy and policy considerations for deep learning in nlp*, 2019. DOI: 10.48550/ARXIV.1906.02243. [Online]. Available: https://arxiv.org/abs/1906.02243.