Analysing the Impact of Inline Comments for the Task of Code Captioning

Vidas Bacevičius
Supervisor(s): Annibale Panichella, Leonhard Applis
EEMCS, Delft University of Technology, The Netherlands
24-6-2022

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

AI-assisted development tools use Machine Learning models to help developers achieve tasks such as Method Name Generation, Code Captioning, Smart Bug Finding and others. A common practice among data scientists training these models is to omit inline code comments from training data. We hypothesize that including inline comments in the training code will provide more information to the model and improve the model's performance for natural-language related tasks, specifically Code Captioning. We adjust one of these models, code2seq, to include inline comments in its data processing, then train and compare it to a commentless version. We find that including inline comments tends to increase the performance of the model by making it faster and producing more verbose results, and then reflect on the results of this work to formulate suggestions on how to improve upon this body of research.

1 Introduction

AI-assisted development has seen a rise in popularity in recent years, with tools such as GitHub Copilot, IntelliJ IntelliSense, OpenAI Codex and many more becoming gradually more accessible to the public. These tools help developers in many different tasks, among them being Code Completion, Documentation Generation and smart Bug Finding. Some of the underlying Machine Learning models behind these tools completely omit code comments from the training data, processing only pure code. This has the benefit of eliminating potentially useless and even harmful data that can decrease the performance of the model (possibly obsolete or incorrect commented out code, natural language comments that can potentially be parsed as code tokens). Removing comments, however, can also get rid of useful information, especially with regards to natural language generation tasks such as Code Captioning and Documentation Generation. In this research paper we will include code comments in an otherwise commentless ML model to see whether the model's performance increases for a specific task.

A particular model that removes comments during the processing of its training data is code2seq[1]. This ML model is used to generate natural language sequences given a piece of code. There are 3 possible tasks that the model can perform: Code Summarization, Captioning, and Documentation. The task of Code Captioning will be the focus of this paper; it involves generating a natural language sentence that describes a given code snippet.

In this research paper we analyse whether including comments in the training data of a model does positively change the outcome of the model's prediction. In particular, the research question to answer is: ***What is the impact of comments in the training code on the performance of code2seq for the task of Code Captioning?***, with the main hypothesis being: *Including comments in the training data will improve the performance of the model for the task of Code Captioning.*

We present the research in the following way: Section 2 introduces the work being done in the field of AI-assisted development, the background of code2seq and how it relates to our hypothesis. Section 3 describes the methodology used to change the model and prepare a dataset for experiments. The experimental setup, metrics used to evaluate the models and the results of the experiment are presented in Section 4. Section 5 explains responsible research and possible ethical issues related to the experiments. Section 6 outlines the existing related work in the field of AI-assisted development and automated comment generation, and how it could help with this research topic. Section 7 offers possible further work and ways to improve upon this research, and Section 8 concludes the work.

2 Background

This section outlines background information about Code Captioning and Documentation Generation, explains the main idea behind code2seq and defines the main problem of comment removal in code2seq.

2.1 Code Captioning vs. Documentation Generation

Although similar, Code Captioning and Documentation Generation are two different tasks. While Documentation Generation is a task that aims to generate a full Javadoc comment for a given method, Code Captioning instead takes smaller pieces of code and aims to describe it with smaller natural language sentences, usually only a few words long.

2.2 Code2seq

Solving natural language-based AI-assisted development tasks such as Code Captioning, Documentation Generation, Method Name Prediction can be seen as a problem of creating a relationship between source code and natural language [2].

There exist many different approaches to this problem. One of them is to consider the input code as a sequence of tokens, producing a sequence of natural language tokens as a result. This approach is aptly named seq-to-seq [3] and has been widely used to solve the aforementioned and similar natural language tasks [4] [5] [6].

Code2seq takes a different approach by representing the input code snippet as "a set of compositional paths over its abstract syntax tree (AST), where each path is compressed to a fixed-length vector using LSTMs"[1] and then using attention to select the relevant paths while decoding.

2.3 Comment Removal

While generating paths over ASTs, code2seq automatically ignores and removes any comment tokens, generating a path that only contains code tokens (Figures 1, 2). This work looks for a way to change the input preprocessing mechanism to meaningfully incorporate comment tokens into the AST paths, which are then passed to the model.

```
void methodName() {
    // This is a comment!
    int a = 1;
}
```

Figure 1: Code snippet with a comment

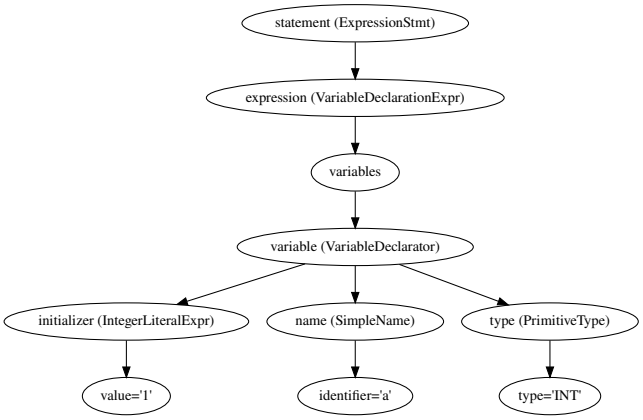

Figure 2: Resulting AST without a comment, during path generation

3 Methodology

This section describes our contribution to this research topic: how we include the inline comments into the AST paths and how the dataset for this task is chosen and processed.

3.1 Including Inline Comments in AST paths

The module responsible for preprocessing the input Java data is called *JavaExtractor*. Normally, this module creates ASTs from the input code using an external library, and then uses visitor classes to scan over the AST and create paths through it. The model then uses these paths to learn and make predictions.

The task of including comments into AST paths is achieved by altering this module. Normally, if an AST leaf represents a comment, it is simply ignored and not added to the path. We change it to check whether a particular node has a comment attached to it, whether that comment is an inline comment and whether it does not include code. If all these conditions are satisfied, this comment is added into the path as a leaf of the closest AST node (Figure 3). The comparison between the original code2seq workflow and the modified one can be seen in Figure 5.

There is a certain drawback that comes with this approach. The external library used to parse the AST nodes only considers the singular closest comment per node. Meaning that any additional comments that precede the node are considered orphans and are not added to the AST (Figure 4). This seemingly easy problem proved to be difficult to solve due to the nature of how *JavaExtractor* processes the AST nodes - the orphan comments are not in scope and are assigned to a semi-random node each time. Due to these reasons we decided that,

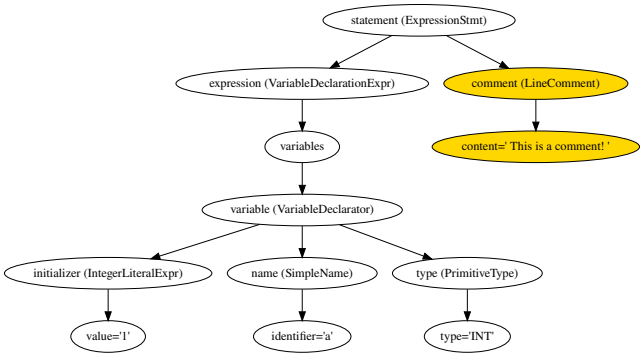

Figure 3: Resulting AST with a comment, during path generation

for the time being, having one comment per node was sufficient. Given more time, it would be possible to rewrite the *JavaExtractor* to correctly include orphan comments.

3.2 Dataset Choice

For any Machine Learning-related task, it is very important to pick a good dataset that represents the proposed problem well. The dataset that code2seq was trained on originally is tailored specifically for the task of Method Generation, focusing on methods and containing barely any inline or Javadoc comments. Therefore it is not suitable for the chosen task.

There exist a few different datasets suitable for the task of Code Captioning. The original code2seq research paper demonstrates the task of Code Captioning by learning on the CodeNN [4] dataset. However, the authors of code2seq raise awareness that this dataset "was very difficult to train the model on" and they recommend "not using this dataset"[1]. Other potential datasets include CONCODE [7], Structured Neural Summarization [8] and CodeSearchNet [9]. The most promising candidate for the dataset is FunCom [10] dataset. This dataset is split into Java methods and their Javadoc comments, is easily preprocessable due to the multiple formats that it provides (filtered and tokenized), and is big enough for our task, as it contains around 2 million Java methods.

Another reason to use this dataset is that the tokenized version of the dataset includes only the first line of corresponding Javadoc comments as labels. This first line is usually a concise description of what that method does, which is exactly what the task of Code Captioning tries to achieve. Of course,

[1] https://github.com/tech-srl/code2seq/issues/17issuecomment-522636556

```
void methodName() {
    // This is an orphan comment.
    // This is a comment!
    int a = 1;
}
```

Figure 4: Code snippet with an orphan comment

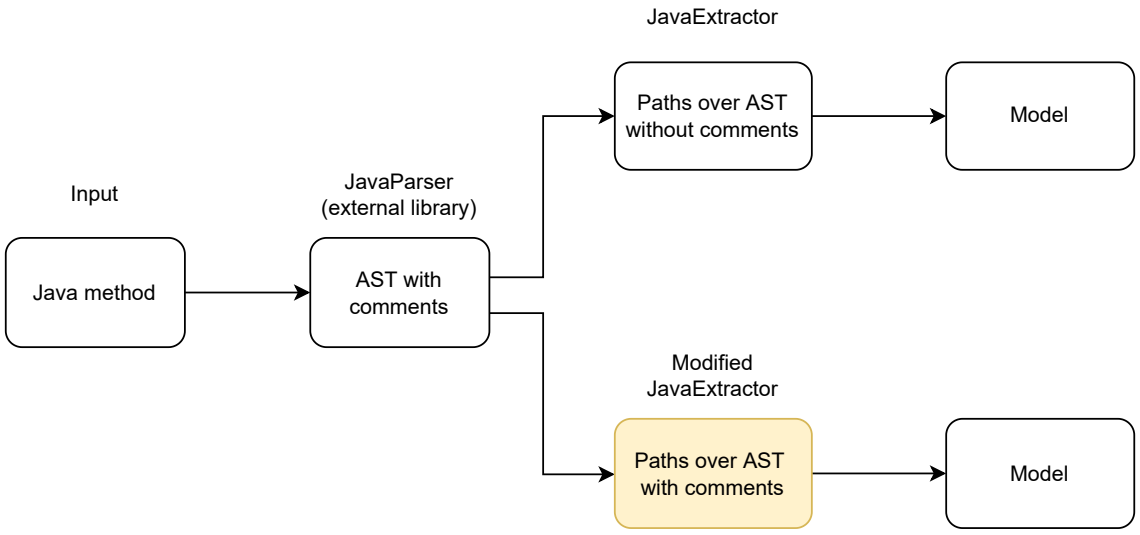

Figure 5: Original and modified code2sec workflows

as not all Javadoc comments start with a concise method explanation, naturally not all labels will provide us with a useful description of the method. Having these factors in mind, we still choose this dataset as the most useful and adaptable to code2seq.

3.3 Processing the Dataset

The FunCom dataset comes in 3 different formats: raw, filtered and tokenized.[2] The filtered dataset contains unprocessed Java method and comment pairs; the tokenized dataset has tokenized methods paired with the tokenized version of the *first line of the respective Javadoc comment*. Since code2seq's *JavaExtractor* requires unprocessed Java source code as input *and* the tokenized version of the comments work really well as labels, we combine the two formats of the dataset. The resulting dataset consists of the unprocessed version of Java methods plus the tokenized version of first line of the respective Javadoc comments.

The dataset is then processed so that each datapoint resembles a Java method - each tokenized comment line was reinserted into the code as a Javadoc comment.

JavaExtractor is modified to take in this new format of input data. Instead of recursively walking through Java project directories, it is changed to parse a singular *jsonl* text file, in which each line contains one Javadoc + method pair. Lastly, *JavaExtractor* is updated to parse the newly included Javadoc comments as labels, instead of method names, as it was previously configured.

4 Experimental Setup and Results

Testing the raised hypothesis includes setting up the experiment on a supercomputer, choosing the evaluation metrics, running the experiment and evaluating the results.

[2]http://leclair.tech/data/funcom/

4.1 Setup

We train the code2seq model twice: the No Comments model is trained with no modifications to the preprocessing of the data, while the Comments model includes inline comments in its AST paths. Both models are trained on the DelftBlue supercomputer using a NVIDIA Tesla V100S 32 GB graphics card. Since the task of Code Captioning requires a short code descriptions, the prediction size of the model is set to a maximum of 15 words.

4.2 Evaluation Metrics

To evaluate and compare the performance of the two models, two different metrics are used: F1 score and BLEU score. Wilcoxon Rank Sum Test is then used to evaluate whether the test results are statistically significant.

F1 Score

The F1 score is defined as the harmonic mean of a test's precision and recall, measured between 0 and 1. Precision in this case is the ratio of identical words between the generated and original comment over the amount of words in the generated comment. Recall is the ratio of identical words between the two comments over the amount of words in the original comment. The formula used for F1 score calculation can be seen below:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{1}$$

This score is chosen as one of the measures because it is hypothesized that including inline comments will improve the precision and recall of the model. Moreover, the code2seq model itself uses the F1 score as the measurement of performance between different training epochs.

BLEU Score

BLEU (Bilingual Evaluation Understudy) is a score for evaluating the quality of machine translation between two natural

Table 1: Results of model evaluation: code2seq with inline comments and without

	BLEU	**F1**	**Precision**	**Recall**	**Training Time**	**Epochs**
No Comments	**15.35**	0.442	0.469	0.418	72h	40
Comments	14.98	**0.461**	0.508	0.422	24h	12

languages [11]. Despite its original purpose, it is also very commonly used to evaluate natural language generation tasks. BLEU works by calculating matching *n*-grams between machine generated and original (human-created) sequences.

BLEU score is calculated using the following formula[3]:

$$BLEU = BP \times exp\left(\sum_{n=1}^{N} w_n \, log \, p_n\right) \tag{2}$$

w_n is calculated as $1/n$ and stands for *n-gram weight*. p_n is defined as the modified precision for *n*-gram. *BP* stands for Brevity Penalty and is used to penalize short machine translations. Is is calculated as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{3}$$

where *c* and *r* stand for the lengths of candidate and reference translations.

We use BLEU score for the evaluation because the original code2seq paper uses this score to evaluate the Code Captioning task, therefore the results of this research and the original can be compared. We do recognize the drawbacks of the BLEU score, namely it being too simplistic for complex natural language generation tasks [12]. For that reason we use two metrics to evaluate the models and suggest more in the Further Work section.

Wilcoxon Rank Sum Test

The Wilcoxon rank sum test is a nonparametric statistical test that compares two paired groups, in this case BLEU scores of each prediction of No Comments and Comments models. We use this test to evaluate the difference between the two sets of predictions and see whether the difference in BLEU scores is statistically significant.

4.3 Results

The results of the experiment are presented in Table 1. The No Comments model reaches a higher BLEU score, but the Comments model performs slightly better in regards to F1 score. Neither of the models, however, reach the original reported BLEU score of 23.04 for Code Captioning. One possible explanation is that the hyperparameters of the model are not optimally tuned for the presented task and dataset. The hyperparameters used are based on the configuration used in the code2seq paper originally for the Code Captioning task. Due to time constraints, there was no option of performing hyperparameter tuning, hence the hyperparameters used are not optimal. Another reason for this issue may be the fact that the dataset originally used for the task of Code Captioning was a C language dataset, as opposed to Java.

Running the Wilcoxon rank sum test over the two sets of BLEU scores results in a *p-value* of 0.12. For a difference between two distributions to be considered statistically significant, this value should be less than 0.05. We therefore conclude that the comparison of BLEU scores between these two predictions is not statistically significant enough and measures other than BLEU should be used to compare these two models.

It is important to note the training time and epochs reached of each model. The training itself stops after 10 epochs of F1 score not improving. While the No Comments model keeps training even after 40 epochs, with F1 score steadily increasing, the Comments model does not. In fact, the best F1 score that the Comments model achieves is reached after only 3 epochs, then the F1 score starts decreasing each epoch. We reason that as the model gets trained with more data, the included inline comments start hindering the Code Captioning prediction. This phenomenon can be seen as a positive, though, as the Comments model reaches a higher F1 score and a comparably high BLEU score in just 24 hours of training, as opposed to 72hrs, which the No Comments model takes.

Prediction Analysis

Analysis of predictions of both models reveals that the Comments model generally produces longer and more verbose predictions, with a tendency to be more human-readable than the No Comments prediction (Table 2, Ex. 1, 2), and occasionally even the reference comment (Table 2, Ex. 3, 4). This also explains the lower BLEU score in contrast to a faster training time and a better F1 score - if the original comment is lacking in descriptiveness (e.g. "unset a variable"), the BLEU score rewards predictions that are closer to it ("removes a variable from the map"), and punishes better, more descriptive predictions ("removes the specified variable from the map"). To fix this, a dataset with more descriptive reference captions should be used to train the model.

Both models tend to occasionally loop and predict a repeating comment, usually connected with articles, adpositions and conjunctions (Table 2, Ex. 5, 6), although it is apparent that the Comments model suffers from this more. One potential way of tackling this issue is stopword filtering, discussed in Section 6.

[3] https://leimao.github.io/blog/BLEU-Score/

Table 2: Examples of generated code captions by both models, compared to the original code comment

Example No.	Original Comment	No Comments Prediction	Comments Prediction
1	returns the ith bspoint	return the ith point	returns the point at the specified index point
2	unset a variable	removes a variable from the map	removes the specified variable from the map
3	initialize shared state	initializes the super class	initializes the internal state of the thread
4	the class of this item	returns the type of the object	returns the type of this object
5	returns a copy of the objects size	returns the size of the queue	returns the size of the size of the size of the size
6	converts music band object to artist for search results object	converts the given code artist band code to the appropriate code code	converts the given band to the given band

5 Related Work

This section provides an overview of existing research in the area of source code comment analysis that can provide more insight into the proposed research question.

A paper by Chen et al. [13] uses random forests to automatically detect the scope of Java source code comments. It reaches a high accuracy of 81%, as well as claims to have provided a solution to comment-code mapping. This approach could be used in conjunction with code2seq to improve its inclusion of comments in the AST paths, since the current comment inclusion in code2seq is quite primitive - as explained previously, the closest code line to the comment gets assigned that comment, with addition of orphan comments being dropped. The approach from Chen et al. could instead assign comments to AST paths based on their automatically predicted code scope.

There has also been work done into classifying code comments according to their different intentions. One such paper by Zhang et al. [14] uses supervised learning to create a taxonomy that classifies Python code comments into different categories, such as *Summary*, *Development Notes*, *Todo*, and so on. This could be helpful in regards to our research by classifying and filtering out certain types of inline comments to fine-tune the data we want to train the model with.

Lastly, a paper analysing source code comments by Geist et al. [15] found that code comments do contain important information about the underlying software system, and by leveraging Machine Learning it should be possible to classify comments and transfer valuable information from the source code into documentation. However, a survey performed by Song et al. [16] found that, in practice, automatic comment generation tools are not that accurate yet, and provides multiple possibilities of how to improve it, such as exploring the synergy between deep neural network and other models.

6 Further Work

This section presents different ways this research can be expanded and improved upon.

As mentioned in Section 3.1, code2seq's *JavaExtractor* ignores orphan comments. This poses a problem of multi-line comments being ignored and only the last comment being taken into consideration. Including orphan comments would provide the model with more realistic data of how code is commented. There are two ways to do that - it is possible to rewrite the logic of *JavaExtractor* to consider orphan comments and their appropriate scope, either adding them as separate comments or concatenating them into a single long comment node. Another way of doing this without modifying *JavaExtractor* would be to preprocess the dataset so that multiple consecutive inline comments are merged into one comment.

The training hyperparameters for this task were not optimal. For further improvements, either manual or automatic hyperparameter tuning should be done to make the model more appropriate for the dataset and task.

During prediction, the model occasionally loops and predicts a repeating comment. This phenomenon appears to be more prominent in the comments model. A potential way to combat this would be to create a list of common English stopwords and filter them out from the inline comments during AST path creation.

Analysing the model predictions reveals that the dataset labels (the first extracted line from a corresponding Javadoc comment) are not ideal for this type of task. These reference comments are usually incomplete, cut off and lack general context. For a better training approach, a dataset with more complete code captions should be selected. Section 3.2 already discusses other possible Code Captioning datasets. To run the updated model on a different dataset, steps described in Section 3.3 should be done to preprocess it.

BLEU and F1 scores are not enough to properly test and

evaluate a complex natural language generation model [12], especially when the reference labels are subpar. F1 and BLEU scores measure the lexical overlap between original and generated sentences. Another set of lexical overlap metrics that could be used is ROUGE [17]. In contrast of BLEU score measuring the precision of the model, ROUGE is used to measure recall. In addition to measuring lexical overlap, one could also measure distributional similarity of model's predictions. One of such metrics is BERT-Score [18], which evaluates the model based on the cosine distances between sequence tokens. A combination of these metrics should be used to broadly evaluate the model's strengths and weaknesses.

7 Responsible Research

This section outlines the ethical considerations and reproducibility of this body of research.

7.1 Ethical Issues

The FunCom dataset that we conducted the research on is based on the UCI Sourcerer project [19]. This project collected the data from open source GitHub projects. Nevertheless, these open-source files can still contain sensitive information. This data can possibly include names, dates, or just code that the authors would prefer not to be used for AI training purposes. A way to minimise the risk of this problem is to use leaked credential finding tools such as truffleHog[4] to find and replace sensitive information.

Another ethical issue can arise if the dataset's code comments contain profanity or socially sensitive topics such as gender or race. Since the updated model learns from inline comments, the model can inadvertently learn to generate socially insensitive comments. There are tools such as Profanity Filter [5] that censors or removes profanities, and Debiaswe [20] that significantly reduces gender bias in word embeddings. In the event of this body of research being reproduced or applied commercially, these tools should be used to minimize social harm that an irresponsibly trained model can bring.

7.2 Reproducibility

Sections 3 and 4 describe in detail the steps taken to preprocess the data, modify the model and perform the experiment. The FunCom dataset is licensed under GNU General Public License v3.0, therefore changes to the forked code2seq repository and the FunCom dataset processing scripts are open-sourced and uploaded to GitHub[6] and Zenodo[7]. It is worthy to note that in order to replicate this experiment, the user should be familiar with computer science and Machine Learning related terminology.

The reproducibility of this research highly depends on the dependencies of the modules used as well as the hyperparameters of the preprocessing and training scripts. In order to reproduce the experiment as accurately as possible, all dependencies and model configurations have been specified in the repository.

As the experiment was performed on the DelftBlue supercomputer, the exact running time and parallelizability of the experiment differ depending on where the experiment is performed. Should the experiment be replicated on the DelftBlue supercomputer again, the *Slurm* training scripts are also provided in the repository.

It is important to note that even with setting up the experiment as similarly as possible, identical or closely similar results can be difficult to reach due to the stochastic nature of Machine Learning models. Small model configuration or dataset changes can cause differing results. Nevertheless, both supporting and opposing results are welcome in expanding the general knowledge about this research topic.

8 Conclusion

The goal of this paper was to test whether changing an AI-assisted development model to include inline comments in the training data improves its performance for the task of Code Captioning. To that end, we chose and preprocessed a Code Captioning dataset and modified the code2seq model to include inline comments in its AST path generation. We then set up an experiment by training and comparing two models with and without inline comments. The No Comments model resulted in a higher BLEU score, however the Comments model reached a higher F1 score in just 1/3rd of the time. A manual prediction inspection revealed that the Comments model produces longer, more verbose and more human-readable captions, which was not captured by either of the used metrics. We therefore conclude that **including inline comments improves the performance of the model**. Our results open up new ways to address comments in training data, and we therefore invite other researchers to improve and expand upon this body of work using different datasets and models.

References

[1] 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.

[2] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *Proceedings - International Conference on Software Engineering*, pp. 837–847, Jun. 2012. DOI: 10.1109/ICSE.2012.6227135.

[3] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, 2014. DOI: 10.48550/ARXIV.1409.3215. [Online]. Available: https://arxiv.org/abs/1409.3215.

[4] https://github.com/trufflesecurity/trufflehog

[5] https://github.com/rominf/profanity-filter

[6] https://github.com/bacevicius/code2seq_codecaptioning

[7] https://zenodo.org/record/6659835

[4] 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)*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. DOI: 10.18653/v1/P16-1195. [Online]. Available: https://aclanthology.org/P16-1195.

[5] M. Allamanis, H. Peng, and C. Sutton, *A convolutional attention network for extreme summarization of source code*, 2016. DOI: 10.48550/ARXIV.1602.03001. [Online]. Available: https://arxiv.org/abs/1602.03001.

[6] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, *A neural architecture for generating natural language descriptions from source code changes*, 2017. DOI: 10.48550/ARXIV.1704.04856. [Online]. Available: https://arxiv.org/abs/1704.04856.

[7] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, *Mapping language to code in programmatic context*, 2018. DOI: 10.48550/ARXIV.1808.09588. [Online]. Available: https://arxiv.org/abs/1808.09588.

[8] P. Fernandes, M. Allamanis, and M. Brockschmidt, *Structured neural summarization*, 2018. DOI: 10.48550/ARXIV.1811.01824. [Online]. Available: https://arxiv.org/abs/1811.01824.

[9] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, *Codesearchnet challenge: Evaluating the state of semantic code search*, 2019. DOI: 10.48550/ARXIV.1909.09436. [Online]. Available: https://arxiv.org/abs/1909.09436.

[10] A. LeClair and C. McMillan, *Recommendations for datasets for source code summarization*, 2019.

[11] 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*, Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. [Online]. Available: https://aclanthology.org/P02-1040.

[12] S. Gehrmann, E. Clark, and T. Sellam, *Repairing the cracked foundation: A survey of obstacles in evaluation practices for generated text*, 2022. DOI: 10.48550/ARXIV.2202.06935. [Online]. Available: https://arxiv.org/abs/2202.06935.

[13] 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, ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.03.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121930055X.

[14] J. Zhang, L. Xu, and Y. Li, "Classifying python code comments based on supervised learning," in *Web Information Systems and Applications*, X. Meng, R. Li, K. Wang, B. Niu, X. Wang, and G. Zhao, Eds., Cham: Springer International Publishing, 2018, pp. 39–47, ISBN: 978-3-030-02934-0.

[15] 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. DOI: 10.1002/spe.2933. [Online]. Available: https://onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/abs/10.1002/spe.2933.

[16] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: Algorithms and techniques," *IEEE Access*, vol. 7, pp. 111 411–111 428, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2931579.

[17] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013.

[18] T. Zhang*, V. Kishore*, F. Wu*, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=SkeHuCVFDr.

[19] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241–259, 2014, Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010), ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2012.04.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016764231200072X.

[20] T. Bolukbasi, K.-W. Chang, J. Zou, V. Saligrama, and A. Kalai, *Man is to computer programmer as woman is to homemaker? debiasing word embeddings*, 2016. DOI: 10.48550/ARXIV.1607.06520. [Online]. Available: https://arxiv.org/abs/1607.06520.