Comment Or Not To Comment: The Effects Of Comments On Method Name Prediction

Nada Mouman
Supervisor(s): Dr. 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

Machine learning (ML) algorithms have been used frequently in the past years for Software Engineering tasks. One of the popular tasks researchers use is method name prediction, which helps them generate an identifier for methods with ML models such as Code2Seq. This model represents code snippets as Abstract Syntax Trees (AST) and it includes all code elements except comments. This paper introduces a novel approach to incorporating comments in Code2Seq. The reimplemented models integrate comments into the AST when preprocessing the data. We filtered comments to reduce the noise introduced in the model and improve the predictions using techniques such as TF-IDF and stopword removal. We implement several models for each filtering technique and one for raw comments. These models are trained and evaluated using the dataset java-small, provided by Code2Seq, with 700K examples and the following metrics: precision, recall, and F1. The results obtained from the evaluation of our model with raw comments show an improvement of 3.62% and 2.36% in precision and F1, respectively, on method name prediction compared to the original model. Furthermore, the model with stopword removal has a 6% and 3.52% gain in recall and F1. These improvements suggest that adding comments to Code2Seq is valuable for better method name predictions since they contain additional information about the methods.

1 INTRODUCTION

Over the past several years, Machine Learning (ML) techniques have shown a growing tendency to be used to automate Software Engineering (SE) tasks [1] such as code completion, code summarization, and defect detection [2]–[4]. These tasks are important because they help developers save time on typing or writing well-documented and comprehensible software.

One of the most commonly used tasks is method name prediction, which generates a word that summarises the code's behaviour of a method. Giving meaningful and conventional identifiers to methods and variables is crucial for a developer to understand a software system [5]. A study by Deissenboeck et al. observes that all identifiers (e.g. variable names, method names) comprise approximately 70% of the source code in terms of characters [6]. Methods with poor names might also reduce the readability of a program [7] because developers have only limited knowledge about the names already used in the system, and identifiers are subject to decay during the development process [6].

One machine learning model that can predict the method's name given a code snippet is Code2Seq [8], which uses Abstract Syntax Trees (AST) [9] to represent the code into a set of syntax paths. Currently, Code2Seq does not incorporate comments during the preprocessing step; therefore, they do not impact the predictions. Including comments in the model may affect the prediction of method names since they can improve the readability of the programs. However, they can also introduce dead code, wrong information about the code, and to-dos [10], [11].

Therefore, this research aims to study how including and removing the comments in the model affects the performance of method name prediction. First, to achieve this goal, we include the comments in the AST during preprocessing. Additionally, we remove stopwords from the comments to verify whether they influence the predictions. We also use TF-IDF (term frequency-inverse document frequency) [12] during preprocessing to create a model which only uses a set of keywords extracted from the comments. These keywords are considered the most relevant words in the comments, and they summarize the content of a comment. Next, we retrain the different models and evaluate them using the metrics: precision, recall and F1. We use the dataset *java-small* provided by Code2Seq, of which 24.50% of information is comments, to assess the models. After evaluation, we compare the results with the original model[1] and discuss the outcome.

The results demonstrate that comments improve the performance of method name prediction; this might be because comments introduce helpful information to the model that helps better predict the identifier. Removing stopwords has also shown a better score than including raw comments, as it might be possible for them to introduce noise to the model. Lastly, the TF-IDF model showed some precision improvement but a lower score on recall than the original model.

This paper is structured as follows: Section 2 covers the required background about ASTs and Code2Seq. Next, in Section 3 we present our changes to the preprocessing to include comments and the techniques used on comments, such as stopword removal and TFIDF. Section 4 explains the experimental setup and the metrics used to evaluate the models. In Section 5, the results obtained during the evaluation are demonstrated and discussed. Section 6 describes different aspects which might affect the study's validity. We reflect on the ethical aspects of the study, such as reproducibility, in Section 7. Finally, the conclusion and future work are explained in Section 8.

2 BACKGROUND AND RELATED WORK

This section provides background information necessary to understand the study. The structure of Abstract Syntax Trees will be described since they are used to represent code in code2seq. Information about code2seq will also be provided to understand how the model is built and works.

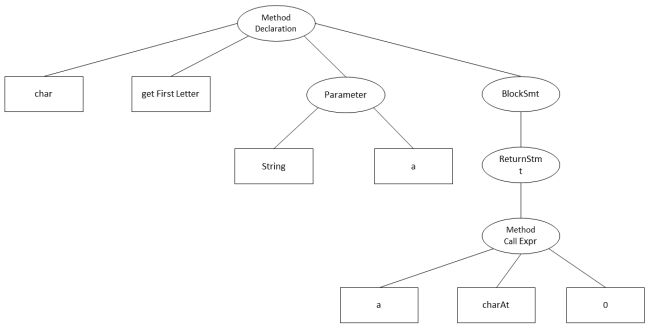
2.1 Abstract Syntax Trees

An Abstract Syntax Tree is a tree representation of the syntax structure of a program. Nodes in the AST represent a code element, and they are distinguished into terminal and non-terminal nodes. The terminal nodes are the leaves of the tree, and they represent the operands and user-defined values, such as identifiers, variables and function calls. The non-terminal

[1] https://github.com/tech-srl/code2seq

```
/** This method gets the first character of a string.
 */
char getFirstLetter(String a){
    return a.charAt(0);
}
```

(a)



(b)

Figure 1: An example code snippet (a) and its corresponding Abstract Syntax Tree (b). The non-terminal nodes are shown as ovals and terminal nodes as rectangles.

nodes stand for the operands of the program (e.g., loops, variable declaration, expressions). An example of an AST and its code is shown in Figure 1.

2.2 Code2Seq

Code2Seq is a neural network which can generate natural language sequences from a code snippet. It can do the following tasks: code summarization, code captioning, and code documentation.

The model uses ASTs to represent code snippets with context pair-wise paths between terminal nodes. They represent a sequence of the terminal and non-terminal nodes connected by up and down movements. In every training iteration, Code2Seq selects a certain number paths k because a code snippet can contain any amount of paths.

Code2Seq uses an encoder-decoder architecture which creates a vector representation for each path in the AST. Each node in a path is represented using a learned embedding matrix E^{nodes}, and then the entire path is encoded using a bi-directional LSTM. The values of the terminal nodes, which are tokens, are split into sub-tokens by CamelCase; for example, ArrayList is divided into Array and List. Then, a learned embedding matrix $E^{subtokens}$ models each sub-token, and the sum of the sub-token vectors represents the whole token.

Next, the token representations are concatenated with the path representation, and they are applied to a fully connected layer in the neural network. In the end, the decoder generates the output sequence by going through all the combined representations.

3 METHODOLOGY

This section describes how we approach the research. First, we analyze the implementation of Code2Seq to understand how it transforms a code snippet into an Abstract Syntax Tree (AST). Next, we change the model to embed the comments in the trees during preprocessing. After evaluating the model with the comments, we modify the content of the comments with techniques including TF-IDF and stopword removal.

3.1 Comment Embedding into Code2Seq

To include the comments in the model, we first needed to understand how a code snippet is represented to get a prediction and then change the representation to embed the comments and study their effect on the task.

Code2Seq represents each code snippet, during the preprocessing step, into an Abstract Syntax Tree. Each code element is represented as a Node in the tree. All nodes, except the root, have precisely one incoming relationship and zero-to-many outgoing relationships. The leaf nodes are the nodes which have zero outgoing relationships. The other nodes with outgoing relationships are branches and parents with one or more children nodes.

Comments are represented as leaf nodes and are associated with a parent node.They are distinguished into LineComment, BlockComment, and JavadocComment. We consider these categories crucial as they might affect the model's performance differently. Each node, except comments, can be associated with at most one comment. The relationship between node and comments is bi-directional. Block and Javadoc comments are assigned to the following node on the same or successive line in the code. Line comments, instead, are attributed to the node preceding them when they are on the same line as another node or the node following them if they are on their own line. Some comments may not be associated with any node because no nodes are preceding or following them. These comments are called Orphan Comments, which are associated with an arbitrary node in the code or the parent of the expected node. Nodes can have multiple orphan comments attributed to them.

We included the comments in the model by visiting each node and verifying whether the node is associated with a comment during the preprocessing step. In case a comment is attributed to a node, we normalized the comment's content by converting it to lowercase. Next, the comment was preprocessed as a node and added to the leaves. In Figure 2, an example of a code snippet with comments and the corresponding AST is shown.

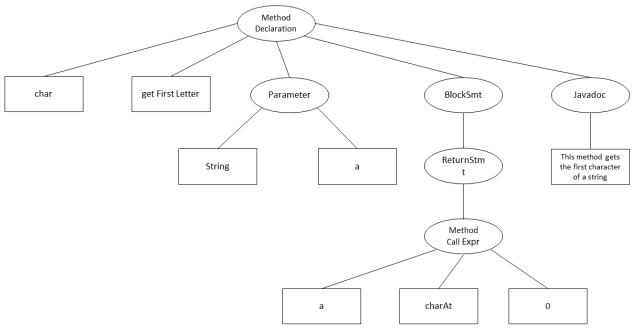
After including the comments in the model, we also wanted to study whether removing stopwords from the comments affects the predictions. Usually, stopwords do not add critical information to the content, decrease the data size and add noise to the data [13], [14]; because of these reasons, we preprocessed and trained a new model without stopwords in the comments. In order to do so, we used Blei et al. [15] dataset containing the stopwords from the English language and went through each comment and removed any word present in the dataset.

3.2 TF-IDF

Other than removing stopwords from the comments, we also studied whether including only a certain amount of words from the comments affected the performance of the model.

```
/** This method gets the first character of a string.
 */
char getFirstLetter(String a){
    return a.charAt(0);
}
```

(a)



(b)

Figure 2: An example code snippet with comments (a) and its corresponding Abstract Syntax Tree (b).

Thus, we used TFIDF (term frequency-inverse document frequency), which is a statistical measure that evaluates the importance of a word to a document in a collection of documents. With TF-IDF, we calculated the relevance of each word in a comment and selected the most important ones as keywords [16].

In TF-IDF, the relevance increases proportionally with the number of times a word appears in a document, but it is balanced with the number of documents that contain the word. The score of a word is calculated by multiplying two metrics: term frequency (TF) and inverse document frequency (IDF).

$$TF.IDF(t, d, D) = TF(t, d) * IDF(t, D) \quad (1)$$

TF measures the number of times a word appears in a document. It is calculated by dividing the word frequency in a document *d* by the total number of words in *d* (see E.q. 2).

$$TF(t, d) = \frac{\textit{Number of times t appears in d}}{\textit{Total number of terms in d}} \quad (2)$$

IDF estimates how common or rare a word is. This metric is calculated by dividing the total number of documents *N* in a collection *C* by the number of documents possessing the word and calculating the natural logarithm of the result (see E.q. 3).

$$IDF(t, D) = \ln\left(\frac{N}{\textit{Number of documents containing t in D}}\right) \quad (3)$$

In our implementation, a comment is considered as a document and the collection of documents is the set of comments contained in a method. We calculate the TF-IDF score for each word in a comment, select a certain number *k* of words with the highest scores, and then remove the remaining terms from the comment. We view these top *k* words as the keywords for a comment, which summarize the information of the comments and might improve the model's performance.

4 EMPIRICAL STUDY

This section showcases the research questions we needed to answer for this research, the parameters required to reproduce the study, and the evaluation metrics and dataset used to evaluate the models.

4.1 Research Questions

This research aims to study the effects of comments on the results of method name prediction for Code2Seq. The main question **RQ1** we want to answer is: *What is the impact of comments on the performance of code2seq for method name prediction?*. Furthermore, we also want to investigate whether applying various filtering techniques on the content of the comments would affect the model. Thus, we define the following sub-questions:

RQ2 *How does "including Javadoc comments" impact the performance of code2seq for method name prediction?*

RQ3 *How does "including inline comments" impact the performance of code2seq for method name prediction?*

RQ4 *How does "filtering the content of the comments" impact the performance of code2seq for code method name prediction?*

4.2 Dataset

We use the dataset *java-small* provided by Code2Seq to evaluate the reimplemented models. Java-small consists of 11 large GitHub Java projects, nine of which are used for training and the other two for validation and testing. This dataset contains around 700K examples for training, 23K examples for validation and 56K for testing. In total, the dataset contains around 12M lines of code of which 3M are used for comments. Around 24.50% of new information from the dataset is added into the model. The Javadoc comments comprise 53.93% of the comments. Instead, the line and block comments are respectively 44.78% and 1.29%. About 15.58% of comments are orphans, hence not used to predict method names.

4.3 Evaluation

We resolve **RQ1** by comparing the model with embedded comments with the original model without them. We use the following metrics, used by Alon et al. [8], to compare the models: precision, recall and F1 score. These metrics measure the quality of a method name prediction using the sub-tokens composing the identifier. During evaluation, the predicted sub-tokens are classified into four groups: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). Precision quantifies how many correct positive predictions the model made, and it is calculated as:

$$precision = \frac{TP}{TP + FP} \quad (4)$$

Table 1: Comparison between original and reimplemented code2seq models on method name prediction using java-small

Model	Precision	Recall	F1
Original Code2Seq	47.30	36.92	41.47
Code2Seq + comments	**49.01**	37.44	**42.45**
Code2Seq + javadoc	44.44	35.59	39.53
Code2Seq + inline comments	47.75	37.30	41.88
Code2Seq without stopwords	47.52	**39.14**	**42.93**
Code2Seq + TFIDF	**48.36**	35.88	41.19

Table 2: Comparison between original and reimplemented code2seq model with comments using java-med

Model	Precision	Recall	F1
Original Code2Seq	57.29	45.42	50.67
Code2Seq + comments	**58.09**	**46.37**	**51.57**

Recall is the ratio of correctly positive predictions of sub-tokens to all the correct predictions the model made, and it is defined as:

$$recall = \frac{TP}{TP + FN} \tag{5}$$

F1 score is a metric which combines both precision and recall into a weighted average. It is determined as:

$$F1 = 2 * \frac{recall * precision}{recall + precision} \tag{6}$$

An unknown sub-token of a predicted method name is regarded as a False Negative. For example, the prediction objectGet for the method getObject has maximum precision and low recall since it is an exact match. However, the prediction getObjectCount has maximum recall and low precision.

We answer **RQ2** by including only the documentation of each method during the preprocessing of the data and then training a new model. We compare this model with the model without comments using the metrics mentioned above. The same procedure is followed to answer **RQ3**, but we include only the inline comments of the methods instead of the Javadoc comments. For **RQ4**, we trained two models which modify the content of the comments by using the following techniques during preprocessing: stopword removal and TF-IDF. Next, we train new models for each filtering technique and compare them with the original model.

4.4 Experimental Setup

The values of all the parameters for preprocessing and training were set up as the same ones used for method name prediction for the original model of Code2seq. The reimplemented models were trained for 8 epochs since they converged and got the best results around 5-8 epochs. We trained models with 2, 4 and 10 keywords for TF-IDF and the results showed minor differences in the F1 scores. Therefore we used 4 keywords for the TF-IDF model because we wanted to have a small group of keywords to represent the content of the comments. Due to time limitations, we could not experiment with other values for the number of keywords The experiments have been carried out using the DelftBlue Supercomputer [17].

5 RESULTS

We changed the Code2Seq model and trained the original and new models to answer the research questions. We evaluated the models using the java-small dataset and used the original model as a baseline for comparison with the other models. We trained the models 3 times with random seed 239 to compare them fairly and get accurate results. Table 1 shows the average results of the three iterations of the original and reimplemented models.

From the results, we can notice some improvement in the model's performance by including the entire content of the comments with no filtering technique. Both precision and F1 are significantly higher for the model with the comments than for the original model. This result might suggest that comments add beneficial information to the model that improves the prediction of the names of the methods.

Surprisingly, the model with only the documentation had a lower score than the original model. There are two likely reasons for this result: the documentation might not follow the rules, which might have introduced noise to the model. Usually, the documentation includes the parameters and the description of the method, but this information could be not up-to-date and provide erroneous data. Alternatively, the amount of documentation in the dataset is not sufficient to affect the model's performance.

The model with inline comments shows a slight performance improvement; this could be because the amount of inline comments in the dataset is insufficient for a significant improvement. Approximately 66.80% of inline comments in the dataset are used to evaluate the models since the rest is orphaned.

Removing stopwords from the content of the comments led to a higher score in both recall and F1 compared to the original model. The plausible explanation for this result is that the stop words might have introduced noise to the model and therefore increased the number of unknown sub-tokens in the dictionary. Unknown sub-tokens are False Negatives, and thus, if the number of FNs increases, the recall decreases. Indeed, we can notice that only recall has changed in a better result, and the precision has remained nearly the same as the original model. Using stopword removal has decreased the number of terms used in the comments by 36.27% and this piece of data might have contained noise as we mentioned above.

The TF-IDF model shows a slight gain in precision but a lower score for the recall than for the original model. One explanation might be that the number of keywords selected from the comments is inadequate to represent the actual content of the comments as roughly 11.28% of the terms in the comments were left after applying TF-IDF to the dataset. The selected keywords could have also introduced unknown sub-tokens and thus, increased the number of False Negatives and decreased the recall.

Additionally, we run the model with the raw comments

with the dataset *java-med*, which has around 3M examples. The results, shown in Table 2, are consistent with those obtained using *java-small*. Due to time limitations, we could not use *java-med* with the rest of the models.

6 DISCUSSION

The current reimplemented models do not include the orphaned comments because they are not associated with a specific node but random nodes. Thus, multiple comments in the dataset might have been excluded during preprocessing; their information would probably affect the model's performance. It would be valuable to include the orphan comments in the model and study their effects using a meaningful manner, such as finding the scope of the comments and associating them with the correct node in a future study.

We used a dataset with 700K methods to train and evaluate the models, but a larger dataset might affect the validity of our research. Because of the time constraints of this research and complications during the training of the models, we could not test our models with the more extensive datasets *java-med* and *java-large* provided by the developers of Code2Seq. These datasets have respectively 3M and 12M examples. A further research is needed to ensure the validity of our results.

Currently, the model with TF-IDF extracts four keywords from the comments, but a different amount of keywords might affect the prediction of method names. Therefore, additional research would be helpful to study the effects of different numbers of keywords for TF-IDF on method name prediction. K-cross validation might be useful to choose the best number of keywords, which gives the best results.

7 RESPONSIBLE RESEARCH

We considered the following ethical aspects during the research: reproducibility, data usage, and energy consumption. We ensured the reproducibility of our research by making the source code used during the study available on a GitHub repository[2]. The code is documented, as every person can use it to reproduce our study. The preprocessing and training times are also included in the documentation with the machine's specifications used during the research. Additionally, the parameters used to train and evaluate the models are listed in Section 4.

We used two datasets for our study: *java-small* and Blei's stopwords. Both datasets are publicly available on GitHub and can be used by anyone. Therefore, we do not affect any ethical issues by using these datasets since everyone can access them and use them to reproduce our study.

Since we included around 24.50% of new data in the model, we looked at the experiment's impact on energy consumption. The preprocessing of the reimplemented models took 30-40 minutes more than the original model. Instead, the training and evaluation times were equivalent for all the models. Hence, we conclude that the gain we achieved from including comments is worth the slight growth in preprocessing time and energy usage. However, a bigger dataset might affect the preprocessing and training times differently.

[2]https://github.com/MrsHan23/code2seq

8 CONCLUSION AND FUTURE WORK

In this paper, we presented a method to include comments in the ML model Code2Seq to study their effects on the performance of the method name generation task. Currently, Code2Seq uses Abstract Syntax Trees to represent a code snippet and comments are not included in the AST during the preprocessing and training steps. To include the comments in the model, we navigated through each node in the AST and verified whether they were associated with a comment. If the node contained a comment, we included the comment as a node in the tree's leaves. Filtering techniques such as TF-IDF and the removal of stop words on the comments were also employed to reduce the noise and dataset size.

The reimplemented models were trained and evaluated using the metrics: precision, recall, and F1. We compared these models with the original model code2seq using the same parameters provided by the documentation of the ML model. The results showed that including comments in the model improves the performance of code2seq for the task method name prediction. Removing stopwords from the comments also had higher scores; an explanation would be that stopwords in comments introduce noise in the model and reduce the recall value.

In the new models, the orphaned comments are excluded because they are not associated with any node. It would be valuable to include them in the model and verify whether the performance is affected. Future research is needed to study the effects of the number of keywords extracted from comments when using TF-IDF. Finding an optimal number might improve the performance of method name prediction.

References

[1] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang, "Deep learning in software engineering," 2018. DOI: 10.48550/ARXIV.1805.04825.

[2] S. Lu, D. Guo, S. Ren, *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021. DOI: 10.48550/ARXIV.2102.04664.

[3] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," 2019. DOI: 10.48550/ARXIV.1910.00577.

[4] Y. Bai, L. Zhang, and S. Yan, "Automatic generation of code comments based on comment reuse and program parsing," pp. 380–388, 2019. DOI: 10.1109/IICSPI48186.2019.9095968.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," ESEC/FSE 2015, pp. 38–49, 2015. DOI: 10.1145/2786805.2786849.

[6] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006. DOI: 10.1007/s11219-006-9219-1.

[7] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," pp. 31–35, 2009. DOI: 10.1109/WCRE.2009.50.

[8] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," 2018. DOI: 10.48550/ARXIV.1808.01400.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," 2018. DOI: 10.48550/ARXIV.1803.09544.

[10] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988. DOI: 10.1109/32.6171.

[11] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," pp. 227–237, 2017. DOI: 10.1109/MSR.2017.63.

[12] L.-P. Jing, H.-K. Huang, and H.-B. Shi, "Improved feature selection approach tfidf in text mining," vol. 2, 944–946 vol.2, 2002. DOI: 10.1109/ICMLC.2002.1174522.

[13] M. Kawahara and H. Kawano, "Mining association algorithm with threshold based on roc analysis," 8 pp.-, 2001. DOI: 10.1109/HICSS.2001.926303.

[14] A. Sarkar and M. S. Hossen, "Automatic bangla text summarization using term frequency and semantic similarity approach," pp. 1–6, 2018. DOI: 10.1109/ICCITECHN.2018.8631934.

[15] D. M. Blei and J. D. Lafferty, "Visualizing topics with multi-word expressions," 2009. DOI: 10.48550/ARXIV.0907.1013.

[16] Y. Qin, "Applying frequency and location information to keyword extraction in single document," vol. 03, pp. 1398–1402, 2012. DOI: 10.1109/CCIS.2012.6664615.

[17] D. H. P. C. C. (DHPC), *DelftBlue Supercomputer (Phase 1)*, https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.