Enabling Log Recommendation Through Machine Learning on Source Code

Liudas Mikalauskas[1] , **Jeanderson Cândido**[1] , **Dr. Maurício Aniche**[1]
[1]TU Delft

Abstract

Logging is a common practice in software development that assists developers with the maintenance of software. Logging a system optimally is a challenging task, thus Li et al. have proposed a state-of-the-art log recommendation model. However, no further attempts exist to improve the model or reproduce their results using different training data. In this research, a model was developed using the methods of Li et al. to evaluate its performance when trained on a specific dataset. Some aspects of the model such as feature filtering were studied. It was concluded that the methods of Li et al. are reproducible and can produce a model that performs well with various training data. The study on feature filtering revealed that not filtering features results in an increase of all tested metrics.

1 Introduction

Logging is a procedure often used by software developers. Logging is applied to generate records of important runtime information. These records help in post-mortem analysis, debugging and maintenance [7, 9, 16]. These studies emphasize that while logging is important, it is a challenging task to log a system in a way that only necessary information is revealed and a performance overhead is not formed. To reduce the complexity of this task, Li et al. have proposed a state-of-the-art model for log recommendations at the block level. However, no further studies exist that confirm the results on different training data or try to improve the results by exploring a more progressive method of fusing syntactic (e.g. block type) and semantic (e.g. variable names) features or make improvements to the deep learning model [9]. These further studies should be carried out to advance log-recommenders and spread awareness about them in the scientific community.

In this research, I aim to answer this question: **what is the performance of a log recommendation model developed following the methods of Li et al., using CloudStack ® [13] source code as training data?** I plan to begin the research by creating a dataset and developing the model. Part of the research is allocated for fine-tuning the model. To conclude the research, I will evaluate the performance of this model and compare the results to those of Li et al.

Paper organisation Section 2 provides an overview of the log recommendation literature. Section 3 describes the methodology of this research. Section 4 will show the adjustments made to the original model. Section 5 describes the details of the evaluation setup and reports the results. Section 6 reflects the ethical aspects of the research and discusses the reproducibility of the methods. Section 7 answers the research question and reflects what has been concluded. Finally, section 8 discusses possible improvements.

2 Background

Several studies are expressing the importance of logging through surveys. In a survey conducted by Fu et al. 96% of surveyed developers strongly agreed "that logging statements are important in system development and maintenance" [5]. The survey of software developers conducted by Lal et al. reveals that developers who deal with logging find it difficult to choose the location and the level of a log [1]. Moreover, many developers expressed that a logging predictor can be helpful. Additionally, Fu et al. claim that the importance of logging comes from the variety of functions it can perform. Some examples are anomaly detection, error debugging, performance diagnosis, workload modelling etc. These claims provide motivation to automate the log placement problem.

Multiple contributions to automate the log placement problem exist. Two notable works in this area show achievements in log recommendations based on code vocabulary. In 2015 Zhu et al. have proposed a recommendation model for catch blocks and if blocks with return statements. In 2018 Li et al. have proposed log recommendations using topic modelling. The calculated topics represented the context and the functionality of the method. Using the topics they learned that a small number of topics (for instance network communication) are likely to be logged. The common advantage of recommenders based on code vocabulary is their independence of programming language since they base predictions on features that are universal (e.g. variable names, types); however, the recommendations provided by these models are somewhat rough because they are either at method level or limited to predetermined locations (e.g. catch blocks). For instance, a recommender like this would not be useful in a code base that has significantly long methods [7, 16]. A solution to this could be the state-of-the-art model proposed in 2020 by Li et al. The model was made for finer log recommendations at the

block level. Their approach involves syntactic and semantic feature extraction, word embedding and a Recurrent Neural Network layer. Since there are no further attempts to improve this approach I have selected to replicate this methodology and confirm the results on new training data. It is also worth noting that there are studies on log severity [8]; however, the scope of this project will not involve log severity and treat all logs at the same level.

3 Methodology

Since the outcome of this research is an evaluation of a log recommendation model, implementing the model is a fundamental task. The recommendation model should be trained on a well-known open-source repository to capture the rules that developers use when choosing logging locations. The development of the model was divided into two sizable parts: the dataset creation and the deep learning model. Both of these parts will be discussed in detail below.

Note: in the following sections, I report on how I implemented a conceptual model described by Li et. al. I do not take any credit for designing the model.

3.1 Dataset Creation

File extraction

The first step in creating a dataset was extracting the Java source files from the repository. The files were extracted using Java NIO API [6]. After extracting the types of the files were determined by the patterns in their paths. ”Test”, ”/test”, ”test/”, ”mock/” or ”/mock” in the path meant the file is a test file, ”/docs/”, ”/examples/”, ”/sample” or ”sample/” in the path meant the file is a documentation file, ”build/” in the path meant the file is a build file, finally, files containing none of these patterns were labelled as production files. To create a dataset, only production files were used with the assumption that test, documentation and build files contain unusual logging practices that should not be mixed with conventional, production logging practices.

Labelling

To extract labels and features the Javaparser library [15] was used. First, an abstract syntax tree (AST) was built for each Java file. Then each AST was traversed to find all methods (the model is suggesting at the block level and that is always inside a method). For each method, the AST was further traversed to identify the blocks. As established in the manual logging study done by Li et. al, there are four types of blocks (try-catch, branching, looping and method declaration) where developers might consider inserting logging statements. These blocks were identified by finding their according nodes.

When a block is identified it is first labelled as logged or non-logged. To label a block, all expression statements containing a method call were checked using the `isLogStatement(String line)` method from the code metric calculator CK [3]. The method works by matching keywords related to logging (e.g. info, warn, debug) in a line of code. It is important to note that when labelling a block only statements directly inside the block were used, all statements inside nested blocks were discarded.

Feature extraction

The feature of a block is a sequence of tokens that represents the structure of the block. Each token is a simple class name of a node in the AST and the correct order is maintained by traversing the AST in a breadth-first manner. The sequences also contain the tokens from the beginning of the method to the beginning of the block. This is done because often the decision to log inside a block can be determined by code before it. Finally, some filtering is done. All tokens associated with log statements are removed to avoid biases in machine learning. Also, nodes for names, parameters, types and modifiers are removed since they do not provide any structural information. Lastly, all tokens of nested blocks are removed since the code inside nested child blocks should not influence the decision of whether to log the block at hand.

Figure 1 is a visual example of labelling and feature extraction scope. Block 1 is labelled as non-logged since there is no log statement inside it (the log statement in line 6 does not belong to Block 1). Block 2 is labelled as logged because of the log statement in line 6. Block 3 is labelled as non-logged because there is no log statement in it. In addition, the feature scope for each block is indicated. When building token sequences for these blocks only the lines in the feature scope were used to extract structural tokens.

3.2 Word embedding

Since only integer values can be input into a neural network, the tokens (strings) needed to be embedded. A popular approach is to add a trainable embedding layer to the deep learning model and train it on the token sequences on the go; However, I decided to isolate the training of the word embedding model. The advantage of isolating the word embedding layer is that after training it, the training states of the model can be discarded keeping just the vectors and their keys in the model object. This allows to save a much smaller and faster object that can be memory mapped “for lightning fast loading and sharing the vectors in RAM between processes” (Radim Řehůřek, 2021) [12]. Thus, in the deep learning part of the model, a keyed vector object is loaded and the vector-matrix is passed into an untrainable embedding layer. This decreases the training time of the deep learning model and allows effortless switching between a variety of different pre-trained embedding models.

For the embedding technique, Skip-gram was chosen over others like One-hot encoding because of two advantages. Firstly, similar words have vectors that are close together and secondly, the Skip-gram vectors capture the context of a word [11]. An assumption was made that these characteristics will benefit the performance of the deep learning model. In table 1 an example of word similarity is provided. It is visible that the embedding of the method declaration token is similar to tokens for variable declaration and block statement which usually appear after a method is declared. It can also be seen that a return statement embedding is similar to a throw statement and assert statement embeddings which usually appear at the end of a method before a return statement. It is reasonable that the empty word is most similar to the return statement since it was used to pad the features of the shorter methods.

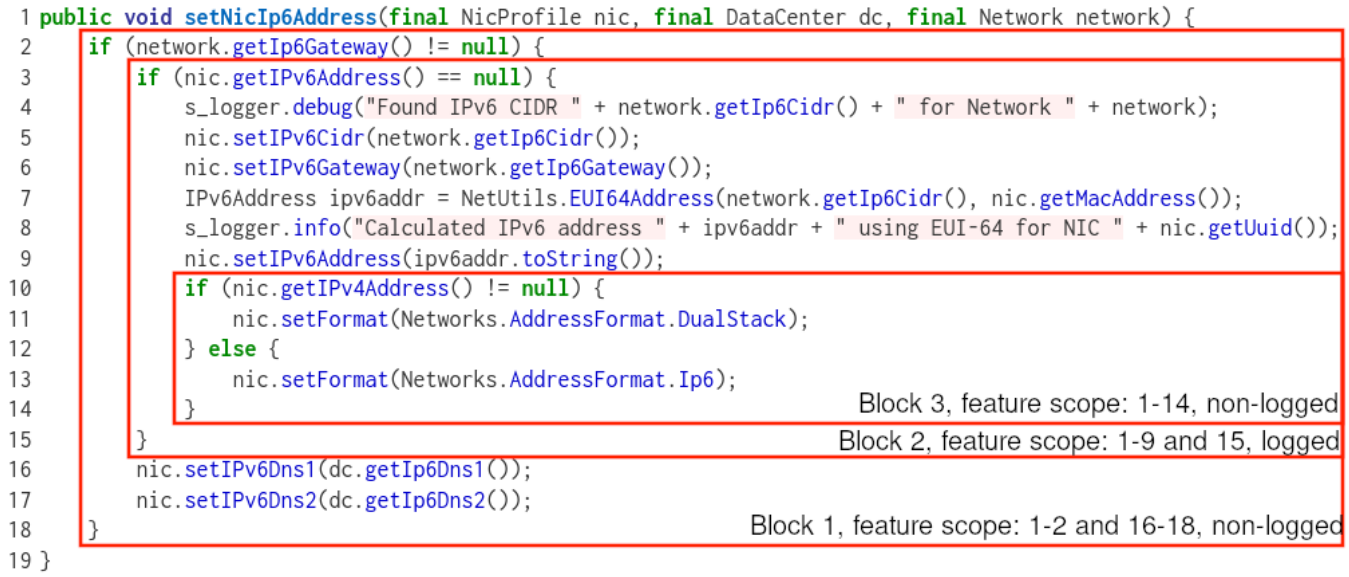

Figure 1: Example of a method extracted from the repository. The red boxes indicate identified branching blocks. The label and feature scope for each branching block is shown.

Word	Similarity
'BooleanLiteralExpr'	0.3267
'MarkerAnnotationExpr'	0.2549
'ThisExpr'	0.2217
'SuperExpr'	0.2023
'NameExpr'	0.1609
'VariableDeclarationExpr'	0.1520
'VariableDeclarator'	0.1519
'NullLiteralExpr'	0.1408
'MethodCallExpr'	0.1300
'BlockStmt'	0.1163

(a) 'MethodDeclaration'

Word	Similarity
''	0.4120
'ThrowStmt'	0.3190
'JavadocComment'	0.2875
'FieldDeclaration'	0.2781
'AssertStmt'	0.2769
'InstanceOfExpr'	0.2588
'NormalAnnotationExpr'	0.2434
'InitializerDeclaration'	0.2225
'EnclosedExpr'	0.2221
'BlockComment'	0.2197

(b) 'ReturnStmt'

Table 1: Top 10 most similar words to words 'MethodDeclaration' and 'ReturnStmt' returned by the Word2Vec model

3.3 Deep learning

The visualisation of the deep learning model can be seen in figure 2. When viewing the dimensions keep in mind that a "None" value means that the model can accept input with any size at that dimension. The embedding layer converts integers to integer vectors. Note that the layer is not trainable but only uses the weights that are loaded from a pre-trained word embedding model. The size of the vector depends on the loaded model. The recurrent layer was chosen because of its ability to handle sequential input. Specifically, the long-short term memory unit was included as the recurrent layer because of its memory cell that enables maintaining information in memory for longer periods [4]. Two dropout layers with a rate of 0.2 were introduced to reduce overfitting [14]. The dense layer is used to convert the output of size 100 (number of internal recurrent states) to a binary prediction.

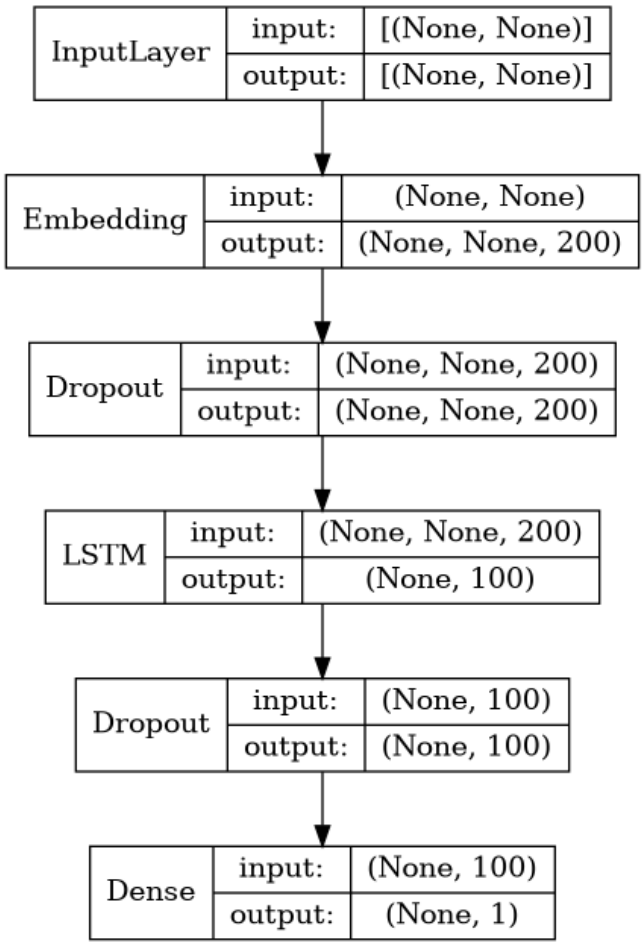

Figure 2: Neural network model

4 Adjustments to the model

Some adjustments, either based on experiments or assumptions, were performed to the original model to increase the performance.

4.1 Feature type

Li et al. extracted three types of features: syntactic, semantic and fused. They concluded that a model trained on syntactic features achieves the best results and encouraged future studies to employ a more sophisticated way of fusing syntactic and semantic features. The research done for this study provided no hope in improving feature fusing. Therefore it was decided that this study will use only syntactic features to achieve the best performance.

4.2 Feature filtering

Li et al. performed filtering on the syntactic features where they removed all nodes that do not provide structural information [9]. To test how that affects performance first the model was trained on the dataset created as described in subsection 3.1. Meaning the feature sequences had tokens for names, parameters, types and modifiers removed. After that, the model was trained on the dataset without any tokens removed. Each model was trained for five epochs then the average accuracy, precision and recall were calculated. The results can be seen in Table 2. The metrics are not defined yet so please refer to subsection 5.1 for their definitions.

	Filtering	No filtering
Accuracy	0.9373	0.9511
Precision	0.5461	0.5665
Recall	0.2784	0.3049

Table 2: Comparison of metrics (5 epoch average) when trained on filtered features v. unfiltered features

The model trained on the unfiltered dataset performed slightly better. The most notable increase was to recall. In the end, the filtering was removed with hopes for better performance at the cost of a higher training time.

Note: some of the increase in performance could have been caused by the stochastic nature of the model.

4.3 Limiting the block length

Another adjustment made to the model was limiting the length of sequence tokens. Namely all sequences that have more than 150 tokens were removed. As can be seen from the appendix section A that means that only a trivial amount of uncommonly large sequences was removed. The result was a great reduction in training time and a small decline in performance metrics.

5 Experimental setup and results

This section will provide details on the metrics chosen, the experimental setup and the results obtained.

5.1 Metrics

The main evaluation metric chosen for this model was F-Measure (FM). It was chosen for its ability to reveal the model's capability of predicting the minority class. This advantage is important since the model tries to predict a class in a very imbalanced dataset (only 9% of blocks are logged) [2].

To calculate the FM first a confusion matrix must be made that enumerates true positives (TP), false negatives (FN), false positives (FP) and true negatives (TN) [10]. The authors of this review further inform that to calculate FM first the Precision and Recall metrics need to be calculated.

Precision (p) "Precision is used to measure the positive patterns that are correctly predicted from the total predicted patterns in a positive class." [10] In the context of logging it measures the correctly predicted logs from the sum of correctly and incorrectly predicted logs.

$$\frac{TP}{TP + FP}$$

Recall (r) "Recall is used to measure the fraction of positive patterns that are correctly classified." [10] In the context of logging it measures the number of correctly identified logged blocks from the sum of correctly identified logged blocks and correctly identified non-logged blocks.

$$\frac{TP}{TP + TN}$$

F-Measure (FM) "This metric represents the harmonic mean between recall and precision values." [10] This review indicates that FM is a metric that combines both precision and recall.

$$\frac{2 * p * r}{p + r}$$

5.2 Setup

To set up for the experiment, all the development in section 3 was completed. Then the model was trained on the dataset created from CloudStack®source code. The model was trained on 80% of the dataset. The remaining 20% were used for validation. After each epoch of training the F-Measure, precision and recall metrics were calculated using the validation set to record how the model changes over time.

If one wishes to reproduce the results please refer to the documentation in https://github.com/luidas/log-placement. The discussion on reproducibility in section 6.2 may also be relevant.

5.3 Results

The results of the experiment can be seen in figure 3. The model was trained for 15 epochs because that's the highest number of epochs with which the training time is reasonable. At epoch 0 the precision and recall are 0. At that epoch, F-Measure was represented as 0 for simplicity although it would be more accurate to not represent it as a number. Taking this into account, the F-Measure of epoch 0 was ignored when interpreting the graph.

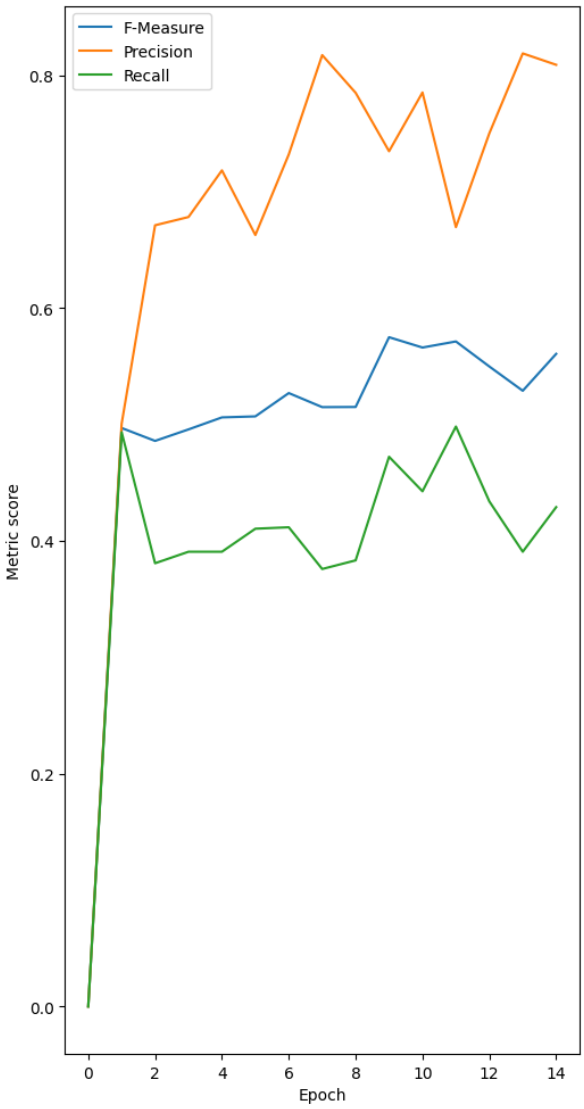

Figure 3: FM, Precision and Recall over 15 epochs

F-measure peaks at epoch 9 with a value of 0.57 (with a precision of 0.73 and recall of 0.47). The average FM value between epochs 1 and 14 is 0.53, while the precision has an average of 0.72 and recall of 0.42. The model also shows potential for learning with a 13% increase in FM between epochs 1 and 14. Another possible evidence of learning is that the sum of differences between every two neighbouring FM values is positive.

6 Responsible Research

This section will reflect on the ethical aspects of the research and discuss the reproducibility of the methods.

6.1 Ethical aspects

The most important ethical issue of this research is the harm that the misuse of this recommendation model can bring.

More precisely, if the future user of this model does not acknowledge its limitations (s)he may overly rely on the model's recommendations and use it in improper situations. Thus a future user needs to acknowledge all the limitations that generally stem from the training data used. It is important to understand that the model cannot perceive code. Its only purpose is to identify the structure of a block and try to mimic what CloudStack developers would do in a block of a similar structure. The legitimacy of the recommendations relies on the fact that CloudStack is widely used in the industry. Another limitation is that the model can only handle recommendations for the Java programming language.

Another ethical issue may arise from the fact that the model and the data extraction process was built on many high-level interfaces. Meaning that if an open-source tool existed for a needed task it was used instead of building a new tool. This considerably saves time but raises some concerns. The main concern is that the internal workings of these open-source tools were not studied but rather treated as a black box. That is because the hidden processes are very complex, and proving their correctness may require immense efforts. This raises a threat to the correctness of the model itself.

6.2 Reproducibility

One way to reproduce the results of this study would be to replicate the implementation described in section 3. To increase reproducibility, the methodology was described as clearly and thoroughly as possible (without exceeding sensible length). However, replicating the methodology might be a complex task for a person with limited experience in machine learning or computer science. What is more similar results are not guaranteed due to the stochastic nature of AI algorithms. For instance, small changes in feature extraction, hyperparameters or structure of the neural network can cause different results.

With these concerns in mind, it was decided that the experiments will be most reproducible if the code was accessible online. Thus a public repository (URL in section 5.2) was created containing the dataset and the code of the deep learning model. The results can be replicated by running a few commands and documentation is included.

The final measure to achieve reproducibility was applying a clever dependency management approach. To do that, a `Dockerfile` was included in the repository, which specifies exactly what software and packages should be installed for the project to run. Instead of struggling with dependencies, all the user needs to do is use the Docker software to build an image using the Dockerfile. The built image contains a base image and all the necessary dependencies for the project installed.

7 Conclusion and Discussion

Research question: what is the performance of a log recommendation model developed following the methods of Li et al., using CloudStack ® [13] source code as training data?

Answer: over 15 epochs the model achieved a peak FM score of 0.57 and an average of 0.53.

Comparing the results with those of Li et al., it can be concluded that the model developed in this study achieved similar performance. The model of Li et al. trained on syntactic features (averaged over all repositories they studied) achieved an FM score of 0.55.

It can be concluded that the developed model can correctly recommend logging locations. It can also be concluded that the methodology of Li et al. is reproducible and an acceptable performance can be achieved with other training data. Also, the effects of feature filtering were studied and it was concluded that no filtering yields better performance.

One specific aspect of the results may be of interest. While the FM score of both models is similar, the precision and recall metrics differ significantly. While the model developed in this study had an average precision of 0.72 and recall of 0.42, the model developed by Li et al. had an average precision of 50.6 and recall of 61.8. This contrast was possibly caused by some small alterations made to the data extraction and deep learning processes when reproducing the methodology.

8 Future Work

The model could not be fully studied due to limited computing resources. Future studies could investigate more demanding configurations of the model like training the model for more than 15 epochs, using a larger neural network, not limiting the length of a block, using bigger word vectors.

Also, the problem of log severity was not tackled in this research. Future studies could explore using structure token sequences for multiple class predictions that represent different log levels (e.g. debug, warn).

Furthermore, future studies could examine if the model trained on the CloudStack dataset performs well when predicting log locations in other codebases.

9 Acknowledgments

I would like to acknowledge my supervisor Jeanderson Cândido for helping to formulate the research question, finding a project to analyse, providing base code for dataset creation (including code to label files as production, test, documentation or build) and providing guidance throughout the project. I would also like to thank Dr. Maurício Aniche for his valuable insights during our meetings. Finally, I would like to thank my peers Kostas Lyrakis and Erwin van Dam for sharing ideas.

References

[1] Optimizing Contemporary Application and Processes in Open Source Software:. Advances in Systems Analysis, Software Engineering, and High Performance Computing, pages 74–75. IGI Global, 2018.

[2] Implementing the Macro F1 Score in Keras: Do's and Don'ts, May 2021.

[3] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[4] Amulya Arun Ballakur and Arti Arya. Empirical Evaluation of Gated Recurrent Neural Network Architectures in Aviation Delay Prediction. In *2020 5th International Conference on Computing, Communication and Security (ICCCS)*, pages 1–7, Patna, India, October 2020. IEEE.

[5] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, Hyderabad India, May 2014. ACM.

[6] Ron Hitchens. *Java NIO*. O'Reilly, Beijing ; Sebastopol, CA, 1st ed edition, 2002.

[7] Heng Li, Tse-Hsun Chen, Weiyi Shang, and Ahmed E. Hassan. Studying software logging using topic models. *Empirical Software Engineering*, 23(5):2655–2694, October 2018.

[8] Heng Li, Weiyi Shang, and Ahmed E. Hassan. Which log level should developers choose for a new logging statement? (journal-first abstract). In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–468, Campobasso, March 2018. IEEE.

[9] Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang. Where shall we log?: studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 361–372, Virtual Event Australia, December 2020. ACM.

[10] Hossin M and Sulaiman M.N. A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):01–11, March 2015.

[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]*, September 2013. arXiv: 1301.3781.

[12] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[13] Navin Sabharwal and Ravi Shankar. *Apache CloudStack Cloud Computing*. Packt Publishing, 2103. OCLC: 968116048.

[14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[15] Danny van Bruggen, Federico Tomassetti, Roger Howell, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch (@skirsch79), Simon, Johann Beleites, Wim Tibackx, jean pierre L, André Rouél, edefazio, Daan Schipper, Mathiponds, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Marvin Wyrich, Ricardo Morais, Maarten Coene, bresai, Implex1v, and Bernhard Haumacher. javaparser/javaparser: Release javaparser- parent-3.16.1, May 2020.

[16] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to Log: Helping Developers Make Informed Logging Decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 415–425, Florence, Italy, May 2015. IEEE.

A Sequence length histogram

The histogram is represented in figure 4.

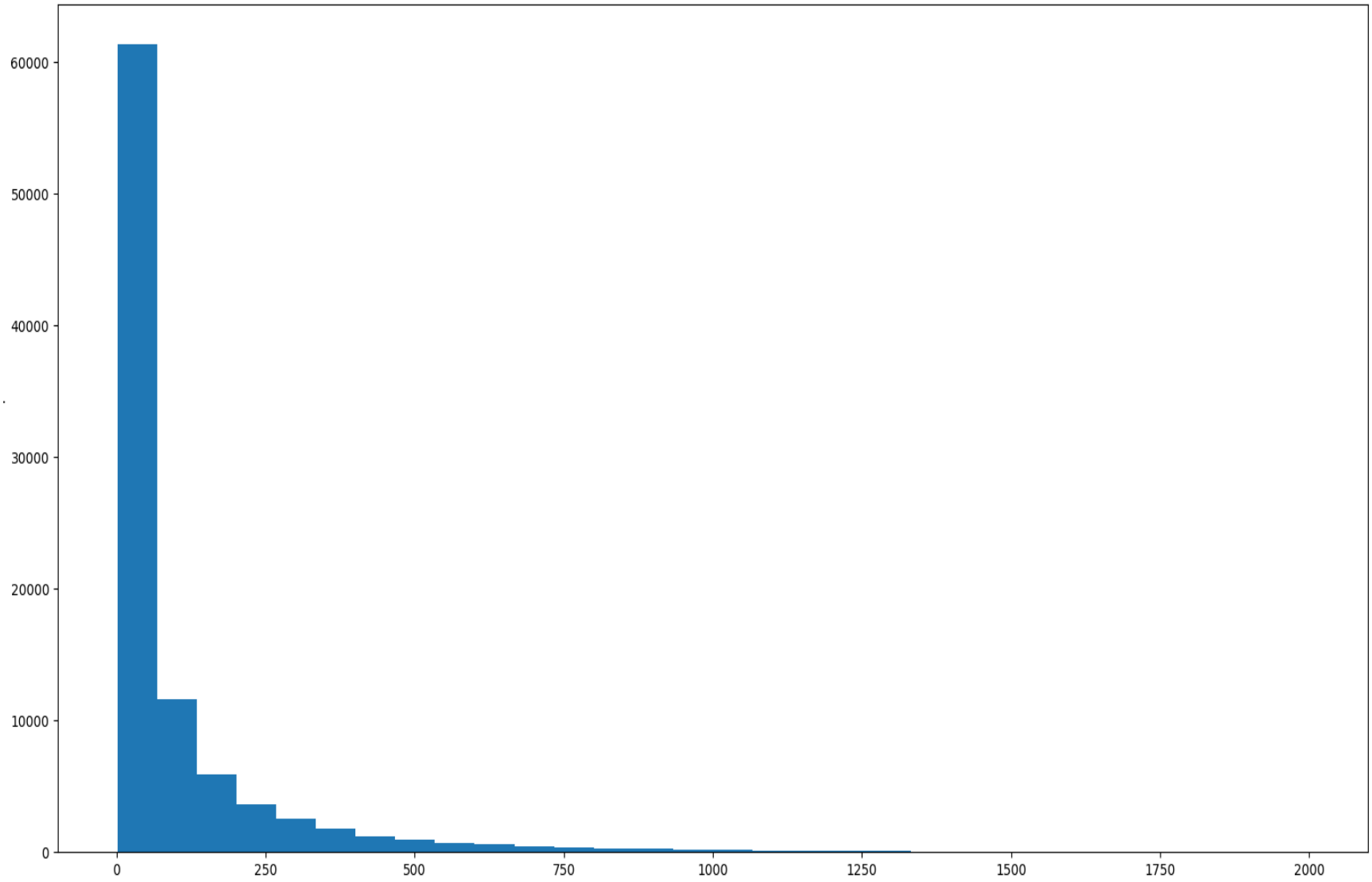

Figure 4: Histogram showing the lengths of token sequences. The lengths are indicated on X-axis, the number of sequences are indicated on Y-axis.