Recommending Log Placement Based on Code Vocabulary

Konstantinos Lyrakis[1] , **Jeanderson Cândido**[2] , **Maurício Aniche** [3]
[1]TU Delft
k.lyrakis@student.tudelft.nl, j.candido@tudelft.nl, m.finavaroaniche@tudelft.nl

Abstract

Logging is a common practice of vital importance that enables developers to collect runtime information from a system. This information is then used to monitor a system's performance as it runs in production and to detect the cause of system failures. Besides its importance, logging is still a manual and difficult process. Developers rely on their experience and domain expertise in order to decide where to put log statements. In this paper, we tried to automatically suggesting log placement by treating code as plain text that is derived from a vocabulary. Intuitively, we believe that the Code Vocabulary can indicate whether a code snippet should be logged or not. In order to validate this hypothesis, we trained machine learning models based solely on the Code Vocabulary in order to suggest log placement at method level. We also studied which words of the Code Vocabulary are more important when it comes to deciding where to put log statements. We evaluated our experiments on three open source systems and we found that i) The Code Vocabulary is a great source of training data when it comes to suggesting log placement at method level, ii) Classifiers trained solely on Vocabulary data are hard to interpret as there are no words in the Code Vocabulary significantly more valuable than others.

1 Introduction

Logging is a regular practice followed by software developers in order to monitor the operation of deployed systems. Log statements are placed in the source code to provide information about the state of a system as it runs in production. In case of a system failure, these log statements can be collected and analyzed in order to detect what caused a system to fail.

Efficient and accurate analysis of log statements, depends heavily on the quality of the log data. Therefore, it is of vital importance that developers decide correctly on where and what to log. However, it is not always feasible to know what kind of data should be logged during development time. Log placement is currently a manual process, which is not standardized in the industry. Although there are various blog posts about logging best practices [1], [2], [3],[4],[5], they mostly concern how to create log statements, rather than defining where or what to log. As a result, developers need to make their own decisions when it comes to logging and they have to rely on their experience and domain expertise to do so.

In this paper, we focus on the "where to log" problem and propose a way to help developers to decide which parts of the source code need to be logged. Prior studies have proposed different approaches in order to achieve this goal.

Zhu et al. [6] proposed a way of suggesting log placement at block level(i.e. whether a block of code should be logged or not) and developed a tool named LogAdvisor. Their study focused on catch-blocks (from exception handling) and if-blocks with return statements. In order to develop LogAdvisor, different machine learning models were trained using structural, textual, and syntactic features derived from the source code.

Li et al. [7] related the log placement problem with the code's functionality. The rationale is that the functionality of a software component can affect the likelihood of this component being logged. Instead of working with specific blocks of code, this study focused on log placement at method-level, because usually each method in the code has one specific functionality. So, some metods are more likely to be logged than others (e.g. methods related to database connections vs getter methods).

What both papers have in common, is that some of the features they used to train their models are textual. In other words, they were derived by treating the source code as plain text. They both showed that there is value in textual features in order to predict whether a code snippet should be logged or not.

This research focuses solely on the value of textual features for recommending log placement at method level. We treat the source code as plain text and consider each method to be a separate document, which consists of words that belong to a vocabulary. Intuitively, we assume that each method's vocabulary can be used to decide whether it should be logged or not. We conjecture that some words from the vocabulary are more closely related to the logging of a method than others. Specifically, we focus on the following research questions:

RQ1: What is the performance of a Machine Learning

model based on Code Vocabulary for log placement at method level? This is the main focus of the research. A well performing classifier trained solely on Code Vocabulary would verify our assumption that Code Vocabulary can provide us with sufficient training data to recommend log placement.

RQ2: What value do different words add to the classifier? By answering this question we can explain why a classifier performs well or not by examining which words provide the most information. We could also detect patterns that lead developers to decide which methods need to be logged or not.

RQ3: How does the size of a method affect the performance of the trained classifier? The long term goal of this research is to help developers make better logging decisions. Answering this question would help us decide what kind of tool we would be able to build for this purpose. For example, if a classifier can accurately predict log placement regardless of the size of a method, we could implement a tool that suggests log placement in real time as the developer types the code of a method.

Paper Organization Section 2 describes the methodology that was followed in order to answer the research questions. Section 3 presents the results for each research question. Section 4 discusses the ethical implications of this research. Section 5 contains a discussion of the results. Finally, Section 6 concludes the paper and provides some suggestions for future work.

2 Methodology

This section describes the process that we followed to answer the research questions mentioned in the introduction. An overview of the methodology can be seen in Figure 1 Subsection 2.1 provides information about the studied systems. The processes described in subsection 2.2 and subsection 2.3 were followed for each system separately.

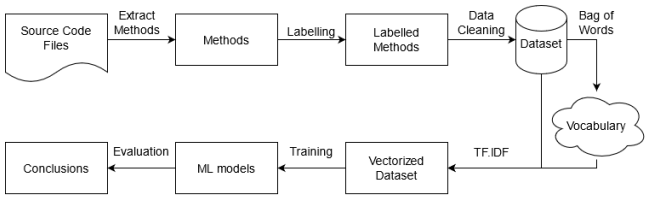

Figure 1: Methodology

2.1 Studied Systems

The systems that were studied during the research are Apache CloudStack[8], Hadoop[9] and Airavata[10]. All the three systems are open source projects built in Java. They were selected because they are large systems with many years of development, so we assume that they follow good logging practices, therefore they constitute a valuable data source for this research.

2.2 Creation of the dataset

The dataset that was used to conduct this research was created from the source code files of the CloudStack repository. For the purposes of this research, only the source code files were used, because the test files are expected to have different logging strategies which would be associated with a different vocabulary.

Method Extraction

The first step towards the creation of the dataset, is to extract the methods from the source code. In order to do so, the JavaParser [11] was employed. For each method, we collected its statements and used CK [12] to count its Lines of Code. In total, 38509 methods were collected, out of which 4040 were logged.

Labelling

A vital step of the data preparation, is to label the methods that were collected. Each method was labeled as "logged" if it contained a log statement, otherwise it was labelled as "not logged". A log statement is a statement that contains a method call to a logging library. All the studied systems use the log4j [13] framework, which defines 6 levels of logging that correspond to similarly named methods: fatal, error, warning, info, debug, trace. So any statement that contains a call to these methods is considered to be a log statement.

Data Cleaning

The source code files, contain many methods that are very small, consisting of less than 3 lines of code. These methods, are less likely to be logged, as they mostly implement very simple functionalities (e.g. getters and setters). We decided to filter out these methods to avoid introducing bias to the classifiers. Table 1 shows that even after filtering out the small methods, the number of logged methods remains the same, so the removed methods would not provide extra information to the classifiers, as they all belong to the "not logged" class.

Methods	Logged	Filtered	Filtered Logged
38509	4040	15282	4040

Table 1: Number of Methods

The second step of Data Cleaning was to remove the log statements from the Logged Methods. Because the goal of this research is to recommend log placement, keeping the log statements would provide the classifiers with information that they would not have in a "real" setting.

Feature Extraction

In order to extract features from the collected methods, we used the Bag of Words model [14]. First, we tokenized the methods and created a vocabulary from the extracted tokens. Afterwards we removed the stop words from the vocabulary. For this research, the stop words constitute of Java's keywords, as all the studied systems are Java projects. Finally, we used TF.IDF [15] term weighting in order to vectorize each method. All of these steps were performed using scikit-learn[16].

Balancing Data

The dataset that was created after the vectorization of the extracted methods, contains very unbalanced data. 26.4% of the extracted methods were labeled as "logged" and the rest were labeled as "not logged". This could negatively affect the performance of the trained models, so we decided to balance the training data by oversampling the "logged" class using the Synthetic Minority Over-sampling Technique (SMOTE)[17]. To do so, we employed the imbalanced [18] tool.

2.3 Model Training

As mentioned before, this paper focuses on suggesting log placement based solely on textual features. As a result, we decided to treat log placement as a binary document classification problem [19], where each method constitutes a document and can be classified as "logged" or "not logged". We trained 5 different classifiers, namely Logistic Regression [20], Support Vector Machines (SVM) [21], Naive Bayes [22], Decision Tree [23] and Random Forest [24]. These classifiers were selected because they are known to perform well in this type of problem [25], [26], [27], [28], [29].

3 Results

This section describes how the results of the research were evaluated and answers the questions posed in section 1.

3.1 Evaluation

After creating the dataset, we employed scikit-learn to train and evaluate the selected classifiers. In order to avoid overfitting, all the classifiers were trained and evaluated on the whole dataset, by using 10-fold [30] cross evaluation. The metrics that were selected for evaluation are Precision (P), Recall (R) and the F1 score ($F1$). They are calculated in the following way:

$$P = \frac{T_p}{T_p + F_p},\ R = \frac{T_p}{T_p + F_n},\ F1 = 2 \cdot \frac{P \cdot R}{P + R}$$

where, T_p is the number of true positives, F_p is the number of false positives and F_n is the number of false negatives. In our case, "logged" is considered to be the positive class, while "not logged" is considered to be the negative class.

Both Precision and Recall are very important metrics when it comes to log placement. If Precision is too low, then the classifier predicts many false positives. Consequently, if the same classifier was used to suggest log placement in a software project, it could lead to unnecessary logging, which could cause unintended problems[31]. On the other hand, a classifier with low Recall, would miss methods that need to be logged. Therefore, it could lead to the loss of important runtime information that would be valuable to detect the cause of a system failure[32]. The F1 score was also used, because it summarizes Precision and Recall in one number, so it can be used to quickly compare many different classifiers.

3.2 Results of RQ1: Performance of the models

The selected models were trained and evaluated by using their scikit-learn implementations. As shown in Table 2, Table 3, Table 4, all the models performed well with regards to the selected metrics. Naive Bayes is a probabilistic classifier that implicitly assumes a strong (naive) independence among the features, which might have led to its lower performance. The dataset that was created for this research contains very sparse data, so this may have affected the performance of Logistic Regression which is known to face estimation difficulties in similar cases [33]. The other 3 models performed similarly well, with Random Forest having the best performance on average among the three studied systems.

Classifier	Precision	Recall	F1 score
Logistic Regression	0.724	0.878	0.793
SVM	0.906	0.860	0.874
Naive Bayes	0.728	0.631	0.675
Decision Tree	0.784	0.864	0.821
Random Forest	0.829	0.939	0.880

Table 2: Classifiers trained with Cloudstack's source code.

Classifier	Precision	Recall	F1 score
Logistic Regression	0.719	0.903	0.799
SVM	0.954	0.892	0.907
Naive Bayes	0.741	0.693	0.715
Decision Tree	0.779	0.894	0.831
Random Forest	0.835	0.968	0.896

Table 3: Classifiers trained with Hadoop's source code.

Classifier	Precision	Recall	F1 score
Logistic Regression	0.873	0.991	0.928
SVM	0.971	0.994	0.982
Naive Bayes	0.863	0.993	0.923
Decision Tree	0.917	0.987	0.950
Random Forest	0.925	0.998	0.960

Table 4: Classifiers trained with Airavata's source code.

3.3 Results of RQ2: Value of Words

After training and evaluating the classifiers, we tried to determine which words of the source code's vocabulary are the most valuable (i.e. provided the most information to the classifiers). In order to do so, we calculated the information gain[34] for each word of the obtained vocabulary. As can be seen in Table 5, the top 20 words from Cloudstack's vocabulary provide almost the same information gain. Figure 2 shows the distribution gain for each word of Cloudstack's vocabulary. One can observe that most words from the vocabulary seem to have an information gain of 0.25-0.30. Therefore, we don't expect any word to significantly affect the performance of the trained classifiers.

Inspired by ablation studies common in the Deep Learning community, we followed a similar process to determine how would the classifiers perform if each one of the top 20 words was removed from the feature vector. For this experiment, we chose the Random Forest classifier, because it had

the best performance across the studied systems. As can be seen in Table 5, there was no significant change in the classifier's performance. We followed the same process for the other two systems, obtaining similar results. The interested reader can see the full results of RQ2 in Appendix A.

Word	Information Gain	F1 score
framework	0.289	0.919
nbe	0.289	0.919
param1	0.288	0.918
secstorage	0.288	0.918
yyyy	0.288	0.917
authenticators	0.288	0.919
localgw	0.288	0.918
ordered	0.288	0.918
chmod	0.288	0.918
spd	0.288	0.918
systemctl	0.288	0.919
thin	0.288	0.917
unbacked	0.288	0.918
ask	0.288	0.917
getstatus	0.288	0.920
hypervisorsupport	0.288	0.918
nop	0.288	0.918
preparing	0.288	0.917
processors	0.288	0.917
searchexludefolders	0.288	0.917

Table 5: Top 20 words by information gain in Cloudstack.

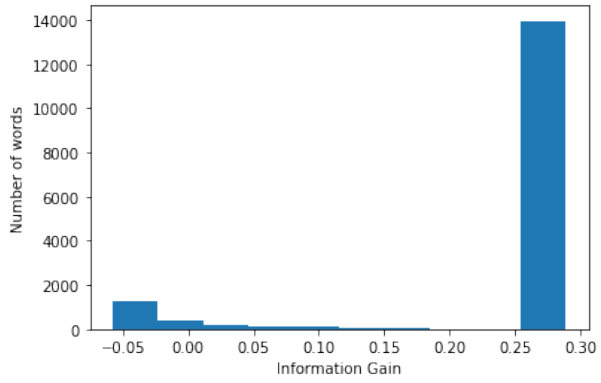

Figure 2: Information Gain distribution in CloudStack.

3.4 Results of RQ3: Method Size vs Performance

The last experiment of this study was to determine whether the size of a method affects the classifiers' ability to predict whether it should be logged or not. In order to do so, we split the test data according to the size of the methods in number of words. Most of the methods in the studied systems consist of 100 or less words, so we didn't consider the methods with more than 100 words for this experiment. We found that bigger methods result to better performance. However, the classifiers were able to perform well even for small methods that consist of 10 or less words. As an example, Figures 3, 4 and 5 show how SVM's performance improves as the methods' size increases. Appendix B contains the results for all the classifiers.

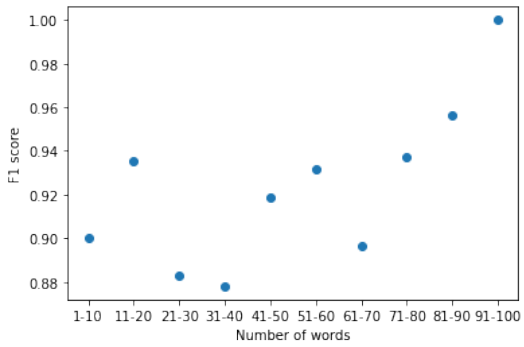

Figure 3: Performance of SVM against method size in Cloudstack.

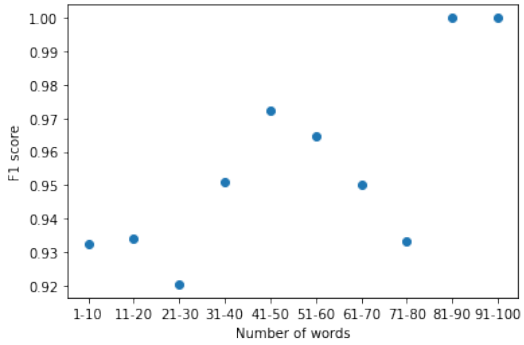

Figure 4: Performance of SVM against method size in Hadoop.

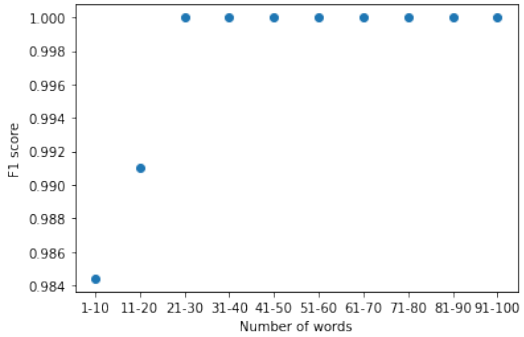

Figure 5: Performance of SVM against method size in Airavata.

4 Responsible Research

This section discusses the ethical implications of this research; subsection 4.1 explores potential threats to the validity of the study, while subsection 4.2 examines the reproducibility of the process that was followed.

4.1 Threats to validity

When conducting experimental research, it is essential to discuss issues that would threaten the validity of the obtained results. For this study, we decided to focus on internal, external and construct validity, as these are the types of validity that relate more to the chosen methodology.

Internal Validity

Internal validity refers to what extend we can be confident about the cause-effect relationship of the studied variables. Different software projects vary in the quality of their logging strategies. The human factor plays an important role in this case, as most projects are maintained by different developers who mostly rely on their own experience to decide which methods should be logged. Therefore, the quality of the trained models heavily depends on the quality of the training data. For our research, we assumed that the developers of the studied system follow good logging practices.

External Validity

External validity is related to the potential of generalizing the obtained results. The results of this study should not be generalized to other systems beyond the three that were discussed in this paper. It would be beneficial to perform the same experiments in more systems in order to be more confident about the obtained results. However, the similarity of the results across the three studied systems leads us to expect that the answers to the research questions provided in this paper can be extended to other systems.

Our research used only the source code of the studied systems and excluded the test code. We preferred to make this distinction instead of using all the code in the repository, because source and test code serve different purposes and therefore were expected to have a different vocabulary. In addition to that, it is the source code that mostly contributes to the systems performance as it runs in production. However, it would be interesting to extend this study so that the test code is also included in the experiments. We expect future studies to train models that use both the source and the test code's vocabulary as training data to evaluate how this would affect their performance.

Construct Validity

Construct validity evaluates the appropriateness of the selected metrics for the given problem. In our case, we used Precision, Recall and the F1 score in order to evaluate the performance of the classifiers. These metrics were selected in order to estimate how a log-placement recommender system would perform if it was built on top of these models.

4.2 Reproducibility of the research

Reproducibility is a vital aspect of any research in order for the readers to be able to verify the obtained results [35]. In Empirical Software Engineering papers, the main reasons why the conducted research cannot be fully reproduced is the unavailability of the used data and unwillingness of the researchers to use the shared code. In order to confirm that the results obtained in this research are fully reproducible, we created a reproducibility package [36] that contains all of the necessary data and documented code. The interested reader is welcome to use our reproducibility package in order to verify the validity or our results.

5 Discussion

This section discusses the main outcomes of the paper.

5.1 Suggesting Log Placement Based on Code Vocabulary

The results of RQ1 show that there is great value in the code's vocabulary when it comes to recommending log placement. All the classifiers that were evaluated, performed well in predicting log placement at method level based solely on the code's vocabulary. Random Forest was the best performing classifier on average in the three studied systems in terms of Precision, Recall and F1 score. Therefore, a log recommendation tool based on a Random Forest classifier would be able to suggest log placement without missing many methods that need to be logged, but at the same time without suggesting too much logging. In addition to that, RQ3 showed that the recommender would be able to perform well regardless of the size of the methods.

5.2 Interpretability of Results

RQ2 showed that in all three systems there were no words significantly more valuable than others. This makes it very hard to explain the good performance of the trained classifiers. As in natural language, words in Code Vocabulary need to be in context in order to have a meaning, so different approaches in Feature Extraction, for example topic modelling [7], would enable us to train more interpretable classifiers.

5.3 Limitations

Our research was conducted on three well established and large Java Systems that could provide us with a sufficiently large dataset. However, many projects are either small or new, so they would provided limited data to a classifier. The quality of this data would also be questionable, as developers rely on their own experience when they decide where to put log statements, so it is expected that some would make poor logging decisions. In such cases, it would be valuable to do cross-project evaluation i.e. train a classifier in one system and use the model to predict log placement in another system. In our research, this was not trivial, as our approach would create different feature vectors for each system. We did not perform cross-project evaluation due to the limited time that was available for this research. However, we expect the classifiers to perform poorly in such cases, as different projects are expected to have significantly different vocabularies.

6 Conclusion

Log placement is vital but also difficult for software engineers. Current logging practices are closely tied to their application domain and in most cases developers rely on their own experience when they have to decide whether a code snippet needs to be logged or not. Prior studies have proposed different solutions to tackle this problems, which indicate that textual features derived from a system's source code can be used to train a Machine Learning model that recommends log

placement. In our research, we focused on the value of Code Vocabulary for suggesting log placement at method level. We treated code as plain text where each method constitutes a document that uses words taken from the Code Vocabulary. We showed that classifiers based on Code Vocabulary can accurately predict whether a method should be logged or not, but they are also hard to interpret.

Classifiers based on Code Vocabulary can accurately recommend Log Placement, but they are hard to interpret.

6.1 Future work

We aim to produce several other works as derivatives of this research. First, we aim to scale the code that was used in order to massively study the value of Code Vocabulary in log recommendation by applying the same process to many open source projects. The studied systems are built in Java, so we plan to study the performance of the classifiers in systems that are built in other Object-Oriented Languages as well.

The ultimate purpose of this work is to create a log-recommendation tool that will help developers make better logging decisions. Initially, this could be a static-analysis tool , where a Random Forest classifier is trained on existing source code in order to recommend log placement at method level. Ideally, we aim to be able to recommend log placement in real time (i.e. as a developer is writing a method's code).

References

[1] R. Kuc, "Java logging best practices: 10+ tips you should know to get the most out of your logs," 2020. [Online]. Available: https://sematext.com/blog/java-logging-best-practices

[2] J. Skowronski, "30 best practices for logging at scale," 2017. [Online]. Available: https://www.loggly.com/blog/30-best-practices-logging-scale

[3] D. McAllister, "Logging best practices: The 13 you should know," 2019. [Online]. Available: https://www.scalyr.com/blog/the-10-commandments-of-logging

[4] T. A. Gamage, "Enterprise application logging best practices (a support engineer's perspective)," 2020. [Online]. Available: https://betterprogramming.pub/application-logging-best-practices-a-support-engineers-perspective-b17d0ef1c5df

[5] L. Tal, "9 logging best practices based on hands-on experience," 2017. [Online]. Available: https://www.loomsystems.com/blog/single-post/2017/01/26/9-logging-best-practices-based-on-hands-on-experience

[6] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 415–425.

[7] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, Oct 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9595-8

[8] (2021) Apache cloudstack: Open source cloud computing. [Online]. Available: https://cloudstack.apache.org

[9] (2021) Apache hadoop. [Online]. Available: https://hadoop.apache.org

[10] (2021) Apache airavata. [Online]. Available: https://airavata.apache.org/

[11] "Javaparser," 2021. [Online]. Available: https://javaparser.org/

[12] M. Aniche, "Java code metrics calculator (ck)," 2015, available in https://github.com/mauricioaniche/ck/.

[13] (2021) Log4j; download apache log4j 2. [Online]. Available: https://logging.apache.org/log4j/2.x/download.html

[14] Wikipedia contributors, "Bag-of-words model — Wikipedia, the free encyclopedia," 2021, [Online; accessed 7-June-2021]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bag-of-words_model&oldid=998080237

[15] L. Jure, R. Anand, and D. U. Jeffrey, *Importance of Words in Documents*. Boston, MA: Cambridge University Pres, 2020, pp. 19–21. [Online]. Available: http://infolab.stanford.edu/~ullman/mmds/booka.pdf

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[17] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *CoRR*, vol. abs/1106.1813, 2011. [Online]. Available: http://arxiv.org/abs/1106.1813

[18] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 559–563, Jan. 2017.

[19] D. Mladeni, J. Brank, and M. Grobelnik, *Document Classification*. Boston, MA: Springer US, 2010, pp. 289–293. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_230

[20] V. Bewick, L. Cheek, and J. Ball, "Statistics review 14: Logistic regression," *Critical care (London, England)*, vol. 9, pp. 112–8, 03 2005.

[21] N. Cristianini and E. Ricci, *Support Vector Machines*. Boston, MA: Springer US, 2008, pp. 928–932. [Online]. Available: https://doi.org/10.1007/978-0-387-30162-4_415

[22] D. Hand and K. Yu, "Idiot's bayes: Not so stupid after all?" *International Statistical Review*, vol. 69, pp. 385 – 398, 05 2007.

[23] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, Mar 1986. [Online]. Available: https://doi.org/10.1007/BF00116251

[24] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[25] J. R. Brzezinski and S. Lytinen, "Logistic regression for classification of text documents," Ph.D. dissertation, 2000, aAI9971843.

[26] L. M. Manevitz and M. Yousef, "One-class svms for document classification," *J. Mach. Learn. Res.*, vol. 2, p. 139–154, Mar. 2002.

[27] S. Ting, W. Ip, and A. Tsang, "Is naïve bayes a good classifier for document classification?" *International Journal of Software Engineering and its Applications*, vol. 5, 01 2011.

[28] W. Noormanshah, P. Nohuddin, and Z. Zainol, "Document categorization using decision tree: Preliminary study," *International Journal of Engineering and Technology*, vol. 7, pp. 437–440, 12 2018.

[29] M. Klassen and N. Paturi, "Web document classification by keywords using random forests," in *Networked Digital Technologies*, F. Zavoral, J. Yaghob, P. Pichappan, and E. El-Qawasmeh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 256–261.

[30] F. Azuaje, I. Witten, and F. E, "Witten ih, frank e: Data mining: Practical machine learning tools and techniques," *Biomedical Engineering Online - BIOMED ENG ONLINE*, vol. 5, pp. 1–2, 01 2006.

[31] C. Eberhardt. (2014) The art of logging. [Online]. Available: https://www.codeproject.com/Articles/42354/The-Art-of-Logging

[32] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 293–306. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/yuan

[33] D. Walker and T. Smith, "Logistic regression under sparse data conditions," *Journal of Modern Applied Statistical Methods*, vol. 18, pp. 2–18, 09 2020.

[34] J. Lutes, "Entropy and information gain in decision trees," 2020. [Online]. Available: https://towardsdatascience.com/entropy-and-information-gain-in-decision-trees-c7db67a3a293

[35] L. Madeyski and B. Kitchenham, "Would wider adoption of reproducible research be beneficial for empirical software engineering research?" *Journal of Intelligent & Fuzzy Systems*, vol. 32, pp. 1509–1521, 2017, 2. [Online]. Available: https://doi.org/10.3233/JIFS-169146

[36] K. Lyrakis, "Java logging best practices: 10+ tips you should know to get the most out of your logs," 2020. [Online]. Available: lyrakisk/log-recommendation: ReproducibilitypackageformyBachelor' sresearchproject

A Results of RQ2

Word	Information Gain	F1 score
framework	0.289	0.919
nbe	0.289	0.919
param1	0.288	0.918
secstorage	0.288	0.918
yyyy	0.288	0.917
authenticators	0.288	0.919
localgw	0.288	0.918
ordered	0.288	0.918
chmod	0.288	0.918
spd	0.288	0.918
systemctl	0.288	0.919
thin	0.288	0.917
unbacked	0.288	0.918
ask	0.288	0.917
getstatus	0.288	0.920
hypervisorsupport	0.288	0.918
nop	0.288	0.918
preparing	0.288	0.917
processors	0.288	0.917
searchexludefolders	0.288	0.917

Table 6: Top 20 words by information gain in Cloudstack.

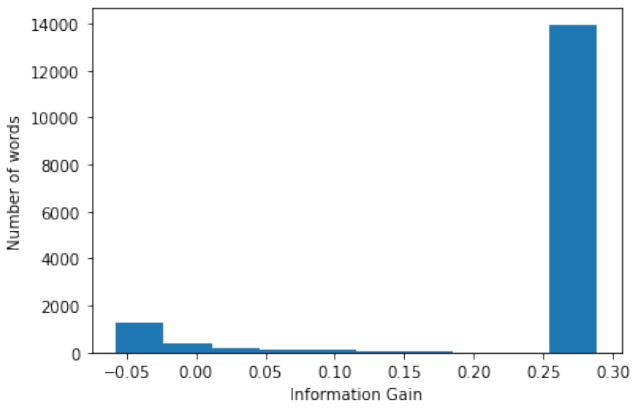

Figure 6: Information Gain distribution in CloudStack.

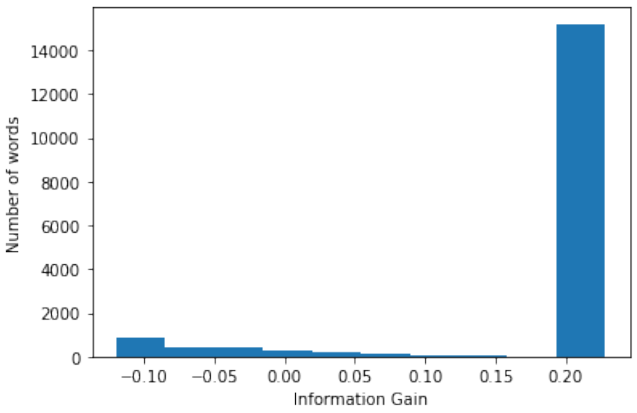

Figure 7: Information Gain distribution in Hadoop.

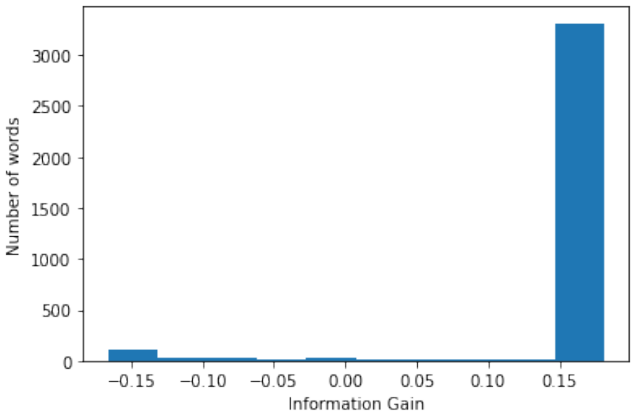

Figure 8: Information Gain distribution in Airavata.

Word	Information Gain	F1 score
hierarchical	0.228	0.941
progressing	0.228	0.940
proc	0.227	0.941
sections	0.227	0.940
sink	0.227	0.940
sleeptime	0.227	0.940
al	0.227	0.941
amtype	0.227	0.940
colons	0.227	0.941
dlisting	0.227	0.940
perspective	0.227	0.940
privileged	0.227	0.940
purged	0.227	0.940
qp	0.227	0.940
restarts	0.227	0.941
scheduling	0.227	0.940
skyline	0.227	0.941
subcluster	0.227	0.941
tmpjars	0.227	0.940
wake	0.227	0.940

Table 7: Top 20 words by information gain in Hadoop.

Word	Information Gain	F1 score
subject	0.181	0.974
authenticator	0.181	0.973
node	0.181	0.974
look	0.180	0.974
request	0.180	0.973
verify	0.180	0.974
action	0.180	0.973
ahead	0.180	0.973
atleast	0.180	0.974
core	0.180	0.974
driver	0.180	0.974
initially	0.180	0.974
jsch	0.180	0.973
ls	0.180	0.974
making	0.180	0.973
mark	0.180	0.974
modify	0.180	0.973
os	0.180	0.973
projects	0.180	0.974
ps	0.180	0.973

Table 8: Top 20 words by information gain in Airavata.

B Results of RQ3

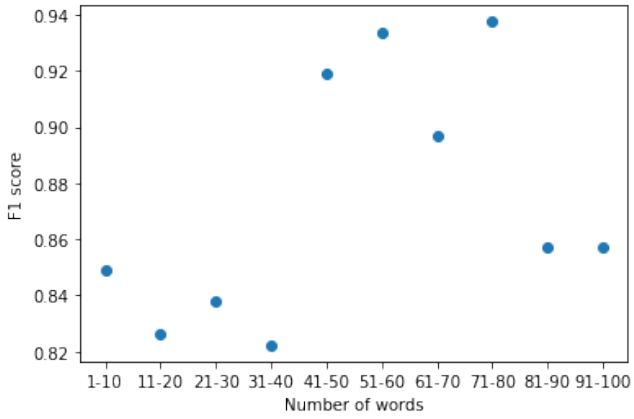

Figure 9: Performance of Logistic Regression against method size in Cloudstack.

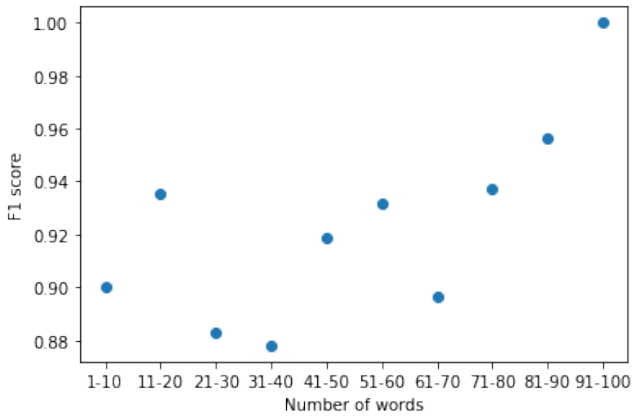

Figure 10: Performance of SVM against method size in Cloudstack.

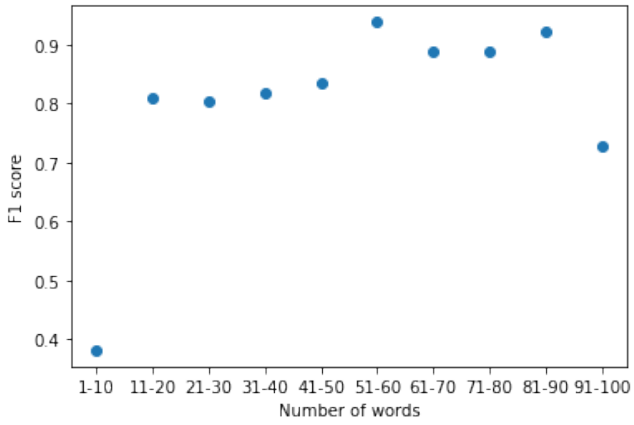

Figure 11: Performance of Naive bayes against method size in Cloudstack.

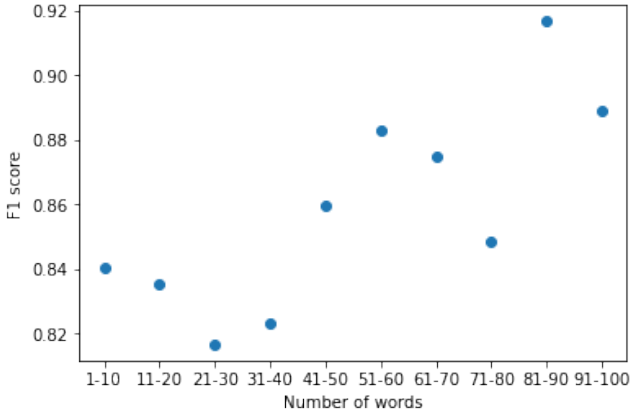

Figure 12: Performance of Decision Tree against method size in Cloudstack.

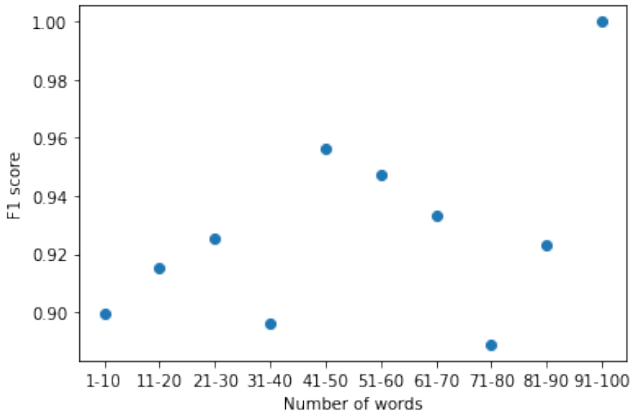

Figure 13: Performance of Random Forest against method size in Cloudstack.

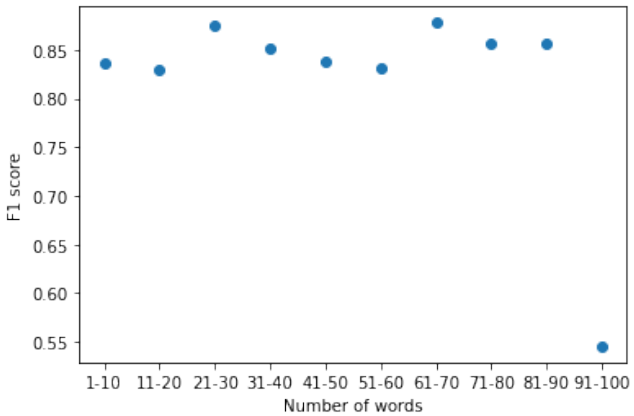

Figure 14: Performance of Logistic Regression against method size in Hadoop.

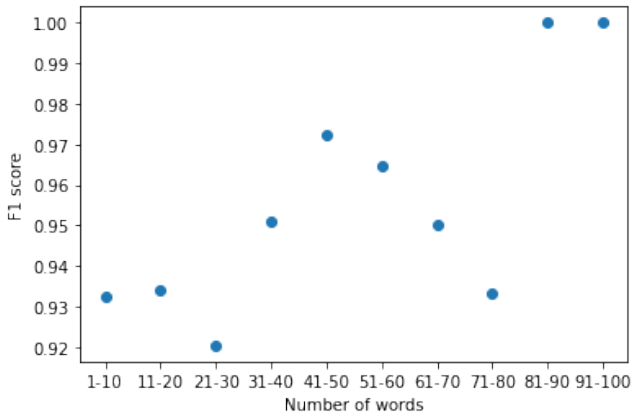

Figure 15: Performance of SVM against method size in Hadoop.

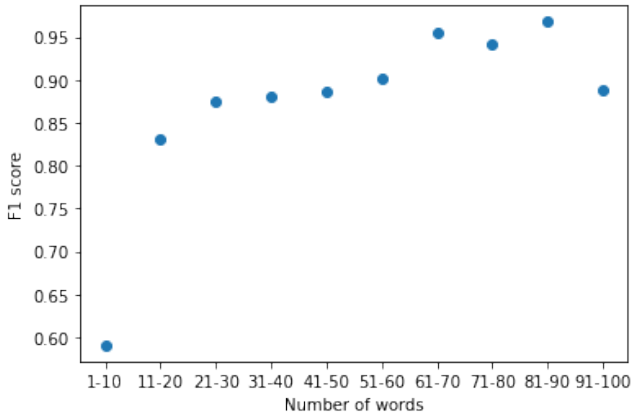

Figure 16: Performance of Naive Bayes against method size in Hadoop.

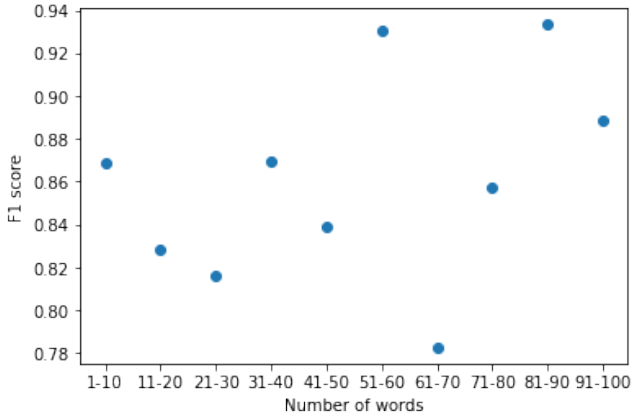

Figure 17: Performance of Decision Tree against method size in Hadoop.

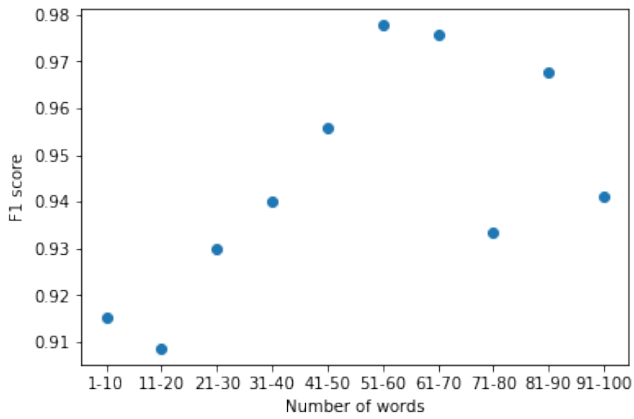

Figure 18: Performance of Random Forest against method size in Hadoop.

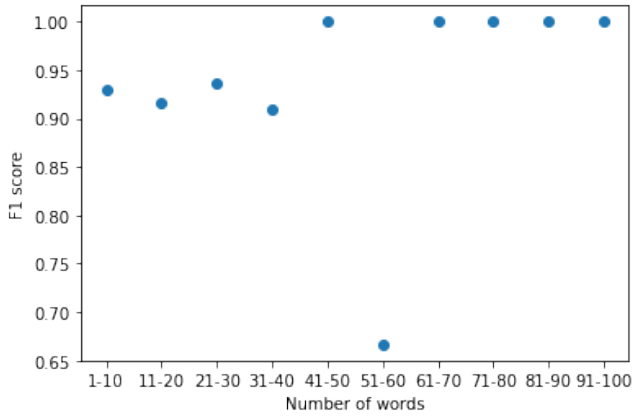

Figure 19: Performance of Logistic Regression against method size in Airavata.

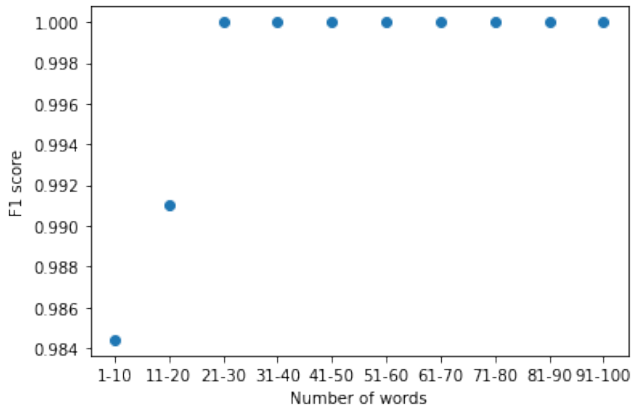

Figure 20: Performance of SVM against method size in Airavata.

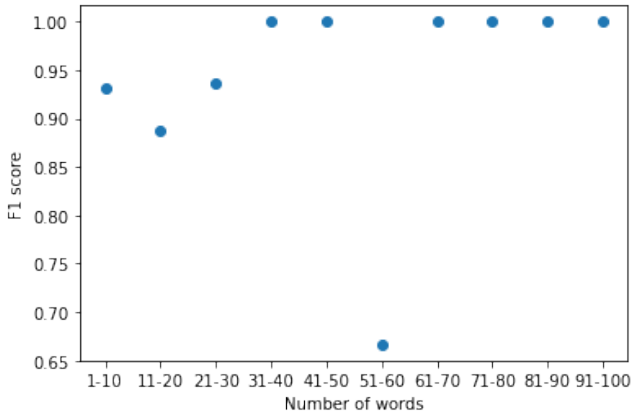

Figure 21: Performance of Naive Bayes against method size in Airavata.

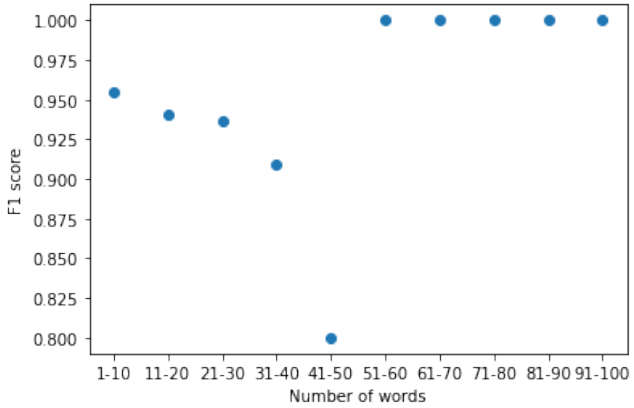

Figure 22: Performance of Decision Tree against method size in Airavata.

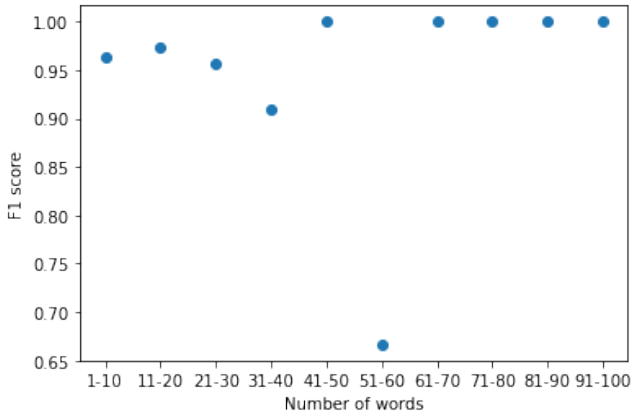

Figure 23: Performance of Random Forest against method size in Airavata.