Identifying Design and Requirement Self-Admitted Technical Debt using N-gram IDF

Supatsara Wattanakriengkrai*, Rungroj Maipradit†, Hideki Hata†,
Morakot Choetkiertikul*, Thanwadee Sunetnanta* and Kenichi Matsumoto†
*Mahidol University, Thailand
†Nara Institute of Science and Technology, Japan
Email: supatsara.wat@student.mahidol.ac.th, {morakot.cho, thanwadee.sun}@mahidol.ac.th
{maipradit.rungroj.mm6, hata, matumoto}@is.naist.jp

Abstract—**In software projects, technical debt takes place when a developer adopting a trivial solution containing quick and easy shortcuts to implement over a suitable solution that can take a longer time to solve a problem. This can cause major additional costs leading to negative impacts for software maintenance since those shortcuts might need to be reworked in the future. Detecting technical debt early can help a team cope with those risks. In this paper, we focus on Self-Admitted Technical Debt (SATD) that is a debt intentionally produced by developers. We propose an automated model to identify two most common types of self-admitted technical debt, requirement and design debt, from source code comments. We combine N-gram IDF and auto-sklearn machine learning to build the model. With the empirical evaluation on ten projects, our approach outperform the baseline method by improving the performance over 20% when identifying requirement self-admitted technical debt and achieving an average F1-score of 64% when identifying design self-admitted technical debt.**

1. Introduction

The main goal of all software projects is to deliver a high quality and defect-free software. However, in many situations, developers have to use shortcuts or temporary solutions in order to complete some urgent tasks. Ward Cunningham defines technical debt as "not quite right code which we postpone making it right" [1]. On the other hand, technical debt refers to a situation of taking shortcuts or temporary solutions to meet some short-term goals, but this phenomenon may increase the maintenance cost in the long run.

Technical debt can be incurred intentionally or unintentionally. Unintended technical debt refers to the technical debt being taken on unknowingly. In contrast, intended technical debt is a debt deliberately coined by developers (called self-admitted technical debt). Potdar and Shihab introduced the notion of self-admitted technical debt (SATD) as a technical debt that is intentionally established by developers [2]. This pioneer study in self-admitted technical debt pointed out that over 30% of source code file in a software project containing self-admitted technical debt. Currently, there are several technical debt works paying particular attention to the intended technical debt [3]; therefore, we determined to conduct an experiment to explicitly study this type of technical debt in this paper.

Prior works have shown that source code comments can be employed to successfully detect self-admitted technical debt [4]. A number of research works [5]–[7] used manual inspection of source code comments to detect self-admitted technical debt. The work in [2] introduced 62 patterns demonstrating the presence of self-admitted technical debt in source code comments derived after the manual inspection of 100k source code comments. Nevertheless, a previous study in [8] pointed out that the manual inspection of source code comments is not efficient in practice since it can lead to reader bias, susceptible to errors, and time-consuming.

Maldonado and Shihab reported that design and requirement debt are the most common types of self-admitted technical debt occurred in source code comments [5]. For design self-admitted technical debt, it ranged from 42% to 84% across projects and requirement self-admitted technical debt ranged from 5% to 45%. To clarify the differences between design and requirement self-admitted technical debt, we provide definitions and example comments of design and requirement self-admitted technical debt below.

- Self-admitted **design** debt refers to source code comments intentionally established by developers in order to indicate that there are problems remaining in the design of the code. Design debt comments can be comments about the violation of the principles of good object-oriented design, misplaced code, long methods, lack of abstraction, poor implementation, and shortcut solutions [9]. For example: *"TODO: - This method is too complex, lets break it up"* - [from ArgoUml] [5], and *"/* TODO: really should be a separate class */"* - [from ArgoUml] [5]. These source code comments directly stated the problems that should be fixed in order to enhance the design of the code. In contrast, the following comment indirectly conveyed the design problem: *"// I hate this so much even before I start writing it. // Re-initialising a global in a place where no-one will see* *it just // feels wrong. Oh well, here goes."* - [from ArgoUml] [5]. In this comment, the developer was concerned that the source code was not written as its best design but still left the design as it was.

- Self-admitted **requirement** debt can be defined as source code comments deliberately created by developers in order to demonstrate that some parts of the code are missing, incomplete, or cannot satisfy the requirement of clients. These following comments are examples of requirement debt comments: *"/TODO no methods yet for getClassname"* - [from Apache Ant] [5], and *"// TODO - should check that error has been logged..."* - [from Apache JMeter] [5]. Here, developers clearly recognize that there is incompleteness of the requirements (missing some methods) found in the source code.

Maldonado et al. proposed an automated model to identify design and requirement self-admitted technical debt using natural language processing and Stanford classifier [8]. Even though, Maldonado et al.'s work is the most recent acceptable approach for detecting the presence of design and requirement self-admitted technical debt in source code comments, the approach can achieve an average F1-score of 40% only [8]. Consequently, we have studied the causes of the difficulty in detecting design and requirement technical debts and found that source code comments indicating design and requirement technical debts are very distinct. This makes a simple model cannot handle them well.

Considering the previous result in [8] as a baseline, we propose our classification models to improve the accuracy by using N-gram IDF and auto-sklearn automated machine learning. Evaluation results indicated that our approach can outperform the baseline approach. For identifying design self-admitted technical debt, we achieve an average precision of 81%, a recall of 56%, and an F1-score of 64%. When detecting requirement self-admitted technical debt, our approach is able to enhance classification performance, reaching an average precision of 80%, a recall of 56%, and an F1-score of 63% that is 23% higher than the baseline. Lastly, to demonstrate that our proposed approach can handle a relatively small training set, we utilize our model to identify *defect* self-admitted technical debt. We discover that our approach can reach an average precision of 62% despite using a very small defect training set contains only 472 source code comments.

The main contributions of our work are the following:

- We design a classifier with automatic approach leveraging N-gram IDF and auto-sklearn automated machine learning to identify design and requirement self-admitted technical debt in source code comments.
- We compare our outcomes with the baseline utilizing publicly available data.

2. Preliminaries

2.1. Detection of Self-Admitted Technical Debt

To replace manual inspection process, several models have been proposed to automate the detection of self-admitted technical debt through code comments. The model of Maldonado and Shihab [5] utilized pattern matching technique in order to classify self-admitted technical debt into five categories: design, requirement, documentation, defect, and test. Farias et al. [10] demonstrated another model to identify self-admitted technical debt in source code comments using code tags and word classes to provide a SATD vocabulary. Huang et al. [11] proposed a text-mining-based model to automatically determine self-admitted technical debt. Liu et al. [12] extended their previous work to propose a tool that is able to detect self-admitted technical debt comments in the source code editor of Eclipse.

2.2. N-gram IDF

N-gram is all combinations of adjacent words of length n that appear in a source text. Generally, an n-gram has more semantic than an isolated word. For instance, the word "*ugly*" on its own does not provide much information; however, when we collect n-gram terms in a source code comment like "*// This is ugly; checking for the root folder.*" - [from Apache Ant] [5], the word "*ugly*" can be an indicator that this source code comment is a technical debt. Nevertheless, using all n-gram terms is quite not useful because they consume enormous memory spaces.

To overcome such problem of using n-grams, we utilize N-gram IDF, a theoretical extension of Inverse Document Frequency (IDF) introduced by Shirakawa [13]. IDF is normally employed to measure the rareness of terms; nonetheless, IDF cannot handle n-grams consisting of more than one word or n-grams with the length of more than 1, (i.e., phrases). N-gram IDF is capable of handling multiple words and phrases; therefore, we can expend N-gram IDF to extract only dominant n-grams of any length by comparing IDF weights of words and phrases. [14].

Terdchanakul [15] leveraged N-gram IDF to construct a bug reports classification model. Their work indicated that N-gram IDF is enabled to handle all of the valid n-gram words and select dominant n-grams to be features of the classifier. With using N-gram IDF, their approach can outperform the classification model based on topic modeling techniques in all cases. Our work is different from the mentioned previous work in that we do not use all n-gram words extracted from the n-gram weighting scheme tool since our n-gram dictionary is large. In contrast, we select dominant n-grams by comparing the weight1 score of n-gram terms in order to find only crucial features for the classification model.

2.3. Automated Machine Learning

For machine learning, two main problems are known: none of the machine learning give the best result in every

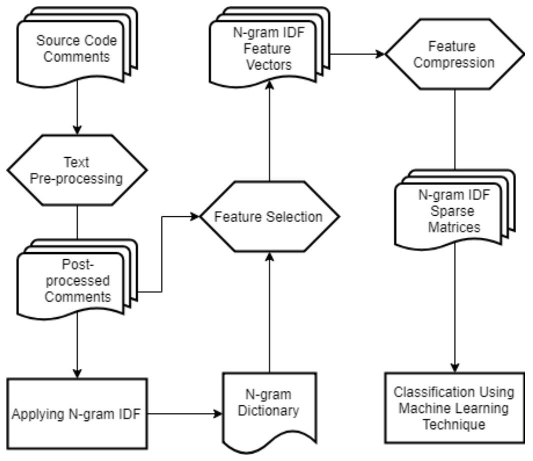

Figure 1. Overview of our source code comment classification model

dataset and some of the machine learning have to use hyper-parameters [16]. Automated machine learning addresses these problems by running multiple classifiers and tries different parameters to optimize the performance.

In our approach, we use auto-sklearn for automated machine learning [17]. Auto-sklearn contains fifteen classification algorithms, fourteen feature pre-processing solutions, and four data pre-process solutions [17]. The components support developer not to set hyper-parameters and tune of machine learning. Auto-sklearn applies two steps to deal with the two problems: meta-learning and automated ensemble construction [17]. Meta-learning is added to be the first step to indicate a solution to use with the dataset. The automated ensemble is added as the last step, it collects information during training progress then construct ensemble which it supports automated machine learning from tuning the same hyper-parameter.

3. Methodology

A. Overview

The main goal of our approach is to accurately identify design and requirement self-admitted technical debt. To do that, firstly, we pre-process source code comments of 10 open source projects using text processing techniques. We then obtain all valid n-gram key terms of the pre-processed document utilizing the tool namely, Ngweight. After we get the output of this process, n-gram dictionary, we employ weight1 score to filter out non-crucial n-grams. For each source code comment, we enumerate the raw frequency of each n-gram term and then collect these values in vector elements. To compress features, we transform vectors into sparse matrices. Finally, these matrices are inputted as the training and testing sets of auto-sklearn automated machine learning. Fig 1 shows an overview of our approach and in the next sections, we describe more details.

B. Text Pre-processing

We convert all date formats appear in source code comments into general words. (e.g., “2007/12/05” changes to “abstractdate”) since previous studies have shown that Ngweight [14] will remove valid n-gram words appear once in a corpus and we also found that dates in the dataset mostly occur once that can result in low coverage of the n-gram dictionary. Next, we convert all special characters into simple words. For example, “?” changes to “questionmark” and “!” changes to “quote”. For the reason that Ngweight will remove all special characters during its process. Besides, special characters are meaningful for identifying design and requirement self-admitted technical debt from source code comments.

C. Applying N-gram IDF

To obtain n-gram terms, we use a library namely, Ngweight[1], computes N-gram IDF weights for all valid n-gram words in the given document. The result after applying Ngweight to pre-processed document is an n-gram dictionary, which consists of all n-gram key terms and other information such as n-gram ID, number of words in n-gram (term length), global term frequency, document frequency of n-gram, and document frequency of a set of words composing n-gram that will be used in the next step.

D. Feature Selection

Due to the large-scale dataset (62,275 documents), our n-gram dictionary is also immense (about 60,000 n-gram terms). It is difficult for auto-sklearn automated machine learning to handle all of them or we need to reserve enormous memory space for supporting the proposed approach. We solve the problem by filtering out n-gram terms that have less consequence on the classification model. Firstly, we remove n-grams appeared in only one source code comment or have global term frequency equal to 1. Then, we compute the weight1 score of each n-gram term. A weight1 score is generally utilized to measure the significance of a term; therefore, n-grams possessed of high weight1 score will be crucial for identifying self-admitted technical debt. We use the following equation to calculate weight1 score of each n-gram term:

$$Weight1 = log(\frac{|D|}{sdf}) * gtf$$

We use only 25% of n-gram dictionary (about 15,000 n-gram words). However, utilizing the small number of n-gram terms has a small impact on the performance of our classification model. After selecting n-gram key terms, we create vector elements, which act as features for our classification, from the pre-processed source code comments corpus and the filtered n-gram dictionary. Each feature vector contains comment ID and the raw frequency value of all dominant n-grams appears in each comment.

E. Feature Compression

1. https://github.com/iwnsew/ngweight

TABLE 1. DETAILS OF STUDY SUBJECTS

	Ant	ArgoUML	Columba	EMF	Hibernate	JEdit	JFreeChart	JMeter	JRuby	Squirrel	Total
Defect	13	127	13	8	52	43	9	22	161	24	**472**
Test	10	44	6	2	0	3	1	12	6	1	**85**
Documentation	0	30	16	0	1	0	0	3	2	2	**54**
Design	95	801	126	78	355	196	184	316	343	209	**2703**
Requirement	13	411	43	16	64	14	15	21	110	50	**757**
No Technical debt	3967	8039	6264	4286	2496	10066	4199	7683	4275	6929	**58204**
Total	**4098**	**9452**	**6468**	**4390**	**2968**	**10322**	**4408**	**8057**	**4897**	**7215**	62275

To utilize more n-gram key terms, we convert feature vectors into sparse matrices[2], which are matrices consist of mostly zero values. The zero values will be ignored, and only non zero values are utilized to process data [18]. This assists our proposed approach to save execution time and deal with more dominant n-grams (from 10% to 25% of n-gram dictionary). From this step, the sparse matrices serve as features of classification model instead of feature vectors.

F. Classification Using Machine Learning Technique

To identify design and requirement self-admitted technical debt, we determine to use auto-sklearn[3], an automated machine learning presented by Matthias [17]. Auto-sklearn will automatically find the best classifier and adjust hyper-parameters of some classification algorithms (e.g., random forest) to achieve the best ones.

4. Evaluation

In this section, we describe how we manage the dataset for the evaluation and explain an approach utilized to assess the performance of our classification model.

4.1. Dataset

We derive the dataset[4] from a previous study [8] and process them with the same way. The dataset consists of the project name, label of the comment that is manually determined using the rules based on prior work by Alves et al [2], and source code comments from 10 open source projects namely, Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. There are six main categories of source code comments: design, requirement, defect, documentation, test, and no self-admitted technical debt code comments (total is 62,275 code comments). After we gathered the dataset, we determine to separate source code comments into two groups, design and requirement self-admitted technical debt, which are the particular focus of our study. The first group contains design and no self-admitted technical debt code comments (60,907 code comments) that we employ to identify design self-admitted technical debt. On the contrary, we utilize another group consists of requirement and no self-admitted technical debt code comments (58,961 code comments) to detect requirement self-admitted technical debt.

2. https://docs.scipy.org/doc/scipy/reference/sparse.html
3. https://automl.github.io/auto-sklearn/stable
4. https://github.com/maldonado/tse.satd.data

4.2. Evaluation Setting

To evaluate our proposed approach, we apply a leave-one-out cross-project validation to two dataset groups, similar to the evaluation of the baseline, by splitting the dataset into nine projects for training and one project for testing. The validation process is repeated 10 times and all projects act as testing data once. We then report an average value of F1-score, precision, and recall after 10 rounds of leave-one-out cross-project validation. Afterwards, we compare the performance of our classification model to the baseline that we construct by following the methodology established in the prior work.

4.3. Result

In this section, we report the performance of our classification model based on leave-one-out cross-project validation setups and use three standard metrics in automating classification: F1-score, precision, and recall to be units of measurement. As we see in Table 2, our approach can outperform the work of Maldonado et al. in many cases with an average precision of 81%, a recall of 56%, and an F1-score of 64% when identifying design self-admitted technical debt. For detecting requirement self-admitted technical debt, our approach is able to improve classification performance, reaching an average precision of 80%, a recall of 56%, and an F1-score of 63%. Accordingly, we can enhance the performance of our model over 20% although we reduce the dictionary size and formats of design and requirement self-admitted technical debt code comments are opposed.

5. Discussions

5.1. Threats to Validity

Relying on the labeled dataset of the prior study. This is obviously a threat to construct validity. Since the labeled dataset established by manual inspection, it can have common errors produced by humans, (e.g., mislabelling and prejudicing) that can result in decreasing classification performance of the proposed approach. For instance: *"//why do we do nothing?"* - [from Apache Ant] seems to classify as no self-admitted technical debt, but actually, this source code comment is categorized as design self-admitted technical debt. Even though the previous

TABLE 2. COMPARISON OF PRECISION, RECALL, AND F1 BETWEEN OUR APPROACH AND BASELINE FOR DESIGN AND REQUIREMENT SATD

	Design Debt						**Requirement Debt**					
	Our approach			**Maldonado et al.**			**Our approach**			**Maldonado et al.**		
	Precision	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Ant	0.676	0.301	0.360	0.554	0.484	0.517	0.650	0.136	0.226	0.154	0.154	0.154
ArgoUML	0.784	0.703	0.741	0.788	0.843	0.814	0.779	0.762	0.771	0.663	0.540	0.595
Columba	0.765	0.940	0.842	0.792	0.484	0.601	0.781	0.935	0.851	0.755	0.860	0.804
EMF	0.802	0.501	0.604	0.574	0.397	0.470	0.826	0.682	0.747	0.800	0.250	0.381
Hibernate	0.833	0.450	0.583	0.877	0.645	0.744	0.809	0.435	0.566	0.610	0.391	0.476
JEdit	0.943	0.701	0.810	0.779	0.378	0.509	0.937	0.715	0.811	0.125	0.071	0.091
JFreeChart	0.872	0.250	0.390	0.646	0.397	0.492	0.846	0.280	0.421	0.220	0.600	0.321
JMeter	0.706	0.420	0.530	0.808	0.668	0.731	0.693	0.418	0.522	0.153	0.524	0.237
JRuby	0.856	0.750	0.801	0.798	0.770	0.783	0.859	0.749	0.800	0.686	0.318	0.435
SQuirrel	0.903	0.630	0.740	0.544	0.536	0.540	0.848	0.535	0.656	0.657	0.460	0.541
Average	**0.814**	**0.564**	**0.640**	**0.716**	**0.560**	**0.620**	**0.803**	**0.565**	**0.637**	**0.482**	**0.416**	**0.403**

study inspected data with fixed rules introduced by Alves et al. [9], mistakes might still occur since they depend on an individual perspective. On the other hand, if there are different rules were utilized, our proposed model might produce distinct outputs.

N-gram extraction tool. This is a threat to the internal validity of our approach, for Ngweight excludes some significant words, which appear once, from the n-gram library. For example, words indicate project name, (e.g., Ant, Columba, and SQL) and programming language, (e.g., Java). These words may be a part of n-gram key terms in the n-gram library. In addition, Ngweight is unable to detect some words that are not English words (e.g., noi18n). These have an impact on the coverage of n-gram library.

Utilizing top n-gram words. This limitation is also a threat to internal validity. According to our large-scale dataset, we have to remove n-gram terms that are not important, from the n-gram library. This is able to affect our classification model works inadequately. To minimize the threat, we use the sparse matrix that is capable of compressing our features, then we can use more n-gram key terms (from 10% to 25% of n-gram library). Nevertheless, to achieve higher accuracy of our classification model, we need to find an approach to solving this problem in the future.

5.2. Can Our Approach Work Well with a Relatively Small Training Set?

Table 1 shows numbers of source code comments in each type of self-admitted technical debt. As we see in Table 1, design self-admitted technical debt has the largest training set (2,703 source code comments) following by requirement self-admitted technical debt (757 source code comments), which they are appropriate for training our model. However, we would like to know that can our proposed approach work well with a relatively small training set? Consequently, we utilize our model to identify defect self-admitted technical debt, which has a very small training set comparing to design and requirement self-admitted technical debt (472 source code comments). We do not use documentation or test self-admitted technical debt to be the sample of the experiment because in the corpus, documentation, and test self-admitted technical debt do not appear in some project, (e.g., documentation SATD does not occur in EMF project and test SATD does not be found in Hibernate project), that can cause problems when using leave-one-out cross-project validation to evaluate outcomes.

TABLE 3. COMPARISON OF AN AVERAGE F1-SCORE, PRECISION, AND RECALL BETWEEN AUTO-SKLEARN AND RANDOM FOREST FOR DEFECT SATD

	Precision	**Recall**	**F1-score**
Random forest	0.550	0.110	0.170
Auto-sklearn	0.620	0.316	0.333

Firstly, we establish a new group of source code comments, which contains only defect and no self-admitted technical debt, and we then follow our methodology to identify defect self-admitted technical debt from source code comments. To evaluate classification performance, we perform leave-one-out cross-project validation or training on 9 projects and testing on 1 project. We report an average value of precision, recall, and F1-score after 10 rounds of leave-one-out cross-project validation.

The evaluation results show in Table 3. Even though we use a relatively small defect training set in the proposed approach, the performance of our classification model is still acceptable with an average precision of 62%, recall of 31%, and F1-score of 33%.

We presume that N-gram IDF is crucial to improving the classification performance when detecting defect self-admitted technical debt since N-gram IDF is enabled to extract dominant key terms varying in both lengths and contexts [15]. To prove this hypothesis, we replace auto-sklearn with random forest classifier. Table 3 demonstrates evaluation results compares between auto-sklearn and random forest classifier. Although we use random forest classifier that is not adjusted hyper-parameters, an average value of precision is acceptable. Therefore, we can summarize that N-gram IDF is able to be utilized as features that make a positive contribution to our proposed approach. However, to achieve high efficiency, auto-sklearn is also a significant part of our classification model.

6. Conclusion

In this paper, we proposed an automatic model based on N-gram IDF-based technique and auto-sklearn automated machine learning to identify design and requirement self-admitted technical debt through source code comments. The purpose of our technique is to accurately detect design and requirement self-admitted technical debt although the patterns of design and requirement source code comments are dissimilar. We conducted an experiment on the dataset that we gathered from the previous study. The dataset contains source code comments of 10 open source projects namely, Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and SQuirrel SQL. To assess classification performance of our model, we performed leave-one-out cross-project validation or training on nine projects and testing on one project. Based on the evaluation results of performing leave-one-out cross-project validation 10 times, we conclude that

- N-gram IDF-based model can outperform the work of Maldonado et al. with an average precision of 81%, recall of 56%, and F1-score of 64% when detecting design self-admitted technical debt. For identifying requirement self-admitted technical debt, our method can produce outstanding outputs with an average precision of 80%, recall of 56%, and F1-score of 63% that exceeds the baseline by 23%.
- Our classification model also works well for classifying defect self-admitted technical debt, that has a relatively small training set with an average precision of 62%.
- Applying N-gram IDF is able to separate source code comments, which have self-admitted technical debt, from no self-admitted technical debt code comments since it can extract dominant n-grams indicated the presence of self-admitted technical debt.

For future work, we plan to enhance the performance of our classification model by using other n-gram extraction tools, which can cover more n-gram key terms, and find an approach to improve the compression of our features. We also plan to extend our work to be able to classify multiclass label corpus and other programming languages. Lastly, we aim to establish an experiment on other areas not only source code comments classification.

Acknowledgments

The authors would like to thank Napat Srisermphoak from Mahidol University for valuable discussions in this work. This work has been supported by JSPS KAKENHI (Grant Number 16H05857 and 17H00731) and Faculty of Information and Communication Technology, Mahidol University.

References

[1] W. Cunningham, “The wycash portfolio management system,” *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, Dec. 1992.

[2] A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 91–100.

[3] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In search of a metric for managing architectural technical debt,” in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, Aug 2012, pp. 91–100.

[4] N. Zazworka, R. O. Spínola, A. Vetro’, F. Shull, and C. Seaman, “A case study on effectively identifying technical debt,” in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE ’13, 2013, pp. 42–47.

[5] E. d. S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical debt,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct 2015, pp. 9–15.

[6] S. Wehaibi, E. Shihab, and L. Guerrouj, “Examining the impact of self-admitted technical debt on software quality,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 179–188.

[7] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 315–326.

[8] E. d. S. Maldonado, E. Shihab, and N. Tsantalis, “Using natural language processing to automatically detect self-admitted technical debt,” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, Nov 2017.

[9] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spnola, “Towards an ontology of terms on technical debt,” in *2014 Sixth International Workshop on Managing Technical Debt*, Sept 2014, pp. 1–7.

[10] M. A. d. F. Farias, M. G. d. M. Neto, A. B. d. Silva, and R. O. Spnola, “A contextualized vocabulary model for identifying technical debt on code comments,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct 2015, pp. 25–32.

[11] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” *Empirical Softw. Engg.*, vol. 23, no. 1, pp. 418–451, Feb. 2018.

[12] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Satd detector: A text-mining-based self-admitted technical debt detection tool,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE ’18, 2018, pp. 9–12.

[13] M. Shirakawa, T. Hara, and S. Nishio, “Idf for word n-grams,” *ACM Trans. Inf. Syst.*, vol. 36, no. 1, pp. 5:1–5:38, Jun. 2017.

[14] ——, “N-gram idf: A global term weighting scheme based on information distance,” in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW ’15, 2015, pp. 960–970.

[15] P. Terdchanakul, H. Hata, P. Phannachitta, and K. Matsumoto, “Bug or not? bug report classification using n-gram idf,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 534–538.

[16] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms,” *CoRR*, vol. abs/1208.3719, 2012.

[17] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, “Efficient and robust automated machine learning,” in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2962–2970.

[18] N. Trendafilov, M. Kleinsteuber, and H. Zou, “Sparse matrices in data analysis,” *Computational Statistics*, vol. 29, no. 3, pp. 403–405, Jun 2014.