A Survey of Self-Admitted Technical Debt Detection

Xianzhen Dou[1,2], Long Li[1], Yubin Qu[1,2]*

[1] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technical, China
[2] School of Information Engineering, Jiangsu College of Engineering and Technical, China
douxianzhen@163.com, lilong@guet.edu.cn, quyubin@hotmail.com

Abstract

Self-Admitted Technical Debt is a key research area in the current software engineering field. By detecting Self-Admitted Technical Debt, potential bugs in software code can be detected early, thus improving software quality. We have systematically organized and analyzed SATD detection in recent years and proposed several future research directions.

Keywords: Survey, Self-admitted technical debt detection, Deep learning

1 Introduction

Technical debt in software development is a kind of specific debt that refers to the errors introduced by programmers consciously or unconsciously in the process of software coding or unfinished code [1]. The purpose of software development is to develop high-quality, defect-free, and structurally complete code. However, in the actual software development process, due to limited resources, such as the limitation of development cost, the requirement of developing time, and the shortage of human resources, the original software development plan will be disrupted [2-3]. In order to deliver software products according to the original plan, programmers may adopt suboptimal schemes to complete software code, including hard coding, function simplification, etc. These suboptimal solutions will affect the robustness of software products in the long run, requiring timely code refactoring [4]. In particular, there is a specific type of technical debt known as self-admitted technical debt (SATD). This technical debt is actively introduced by programmers, such as hard coding function parameters, and is recorded in code comments. Many previous studies have shown that code annotation plays a key role in ensuring the quality of software products [5-6]. Therefore, Potdar et al. investigated this kind of technical debt in code comments and called it self-admitted technical debt [7]. Their research shows that SATD is widespread and may have a negative impact on software maintenance. Subsequently, Wehaibi et al. conducted an empirical study [8], indicating a correlation between SATD and software quality. SATD may not only lead to software defects but also to the failure of software system reconstruction in future iterations. Therefore, it is necessary to identify SATD in time in the process of software development and repay the existing technical debt. This problem is called self-admitted technical debt in academia.

To systematically analyze, summarize and compare this issue, important academic search engines at home and abroad (such as Google Scholar, DBLP, CNKI, etc.) are selected to search for papers related to this review topic. Select the English keyword "self-admitted technical debt detection" and search in DBLP. Seven papers directly related to SATD detection were reviewed, including one paper on empirical research for self-adaptive technical debt detection in blockchain software projects, which is the latest achievement of the research group in 2022.

Select the English keyword "self-admitted technical debt" and search in DBLP.

There have been 75 relevant SATD research papers since 2014. In recent years, it has shown an increasing trend year by year. In 2022, 23 related papers were published, and Emad Shihab and others were the main researchers in this field. To understand the research on SATD in China, the keyword "self-admitted technical" is selected for subject retrieval in CNKI. There are seven relevant papers; among them, four papers are related to the detection of self-admitted technical debt, which shows that this problem has been paid attention to by domestic researchers. In 2022, the research team of Nanjing University published a review article [9] in the Journal of Software, which deeply combed the self-admitted technical debt. Through quantitative analysis of published papers, they found that the number of papers published in this field showed an overall upward trend year by year. Especially in 2019-2020, this shows that the research field of SATD is getting more and more attention from researchers.

Then, we analyzed the title, publication source, abstract, and keywords of the paper, filtering out the papers unrelated to the review topic. The relevant cited papers were analyzed to add the missing papers, and the relevant authors were analyzed to add the latest research results. Finally, 27 papers related to the topic were determined. Compared with the review article research on self-admitted technical debt [9], the course group pays more attention to the breadth of research in the subfield of self-admitted technical debt detection, and the research goal focuses on the detection of self-admitted technical debt.

SATD detection techniques could be strengthened by including a discussion on the broader implications and applications of SATD detection in software engineering

*Corresponding Author: Long Li; E-mail: lilong@guet.edu.cn
DOI: https://doi.org/10.70003/160792642025052603010

practices. This includes the impact on software quality, maintenance, and technical debt management. Understanding the broader implications can provide insights into how SATD detection can be integrated into software development processes and what benefits and challenges it may bring. For example, the early detection of SATD can help improve software quality by identifying potential issues early in the development process. It can also facilitate technical debt management by providing a clear understanding of the technical debt landscape within a codebase. Additionally, SATD detection can aid in software maintenance by highlighting areas of the code that may require refactoring or further attention. Overall, the inclusion of a discussion on the broader implications and applications of SATD detection would enhance the paper's contribution to the field of software engineering.

2 Research Framework of Self-Admitted Technical Debt Detection

By analyzing the existing research results on self-admitted technical debt detection, the following two scenarios can be summarized:

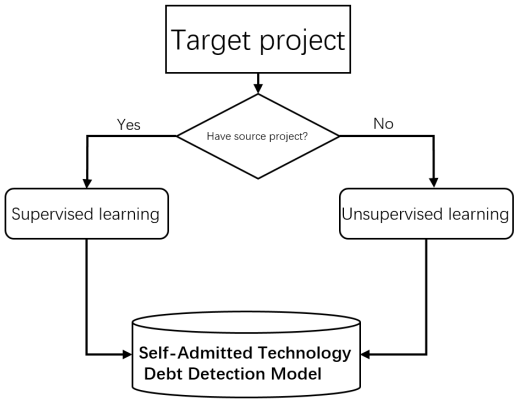


Figure 1. Research framework

(1) Scenario 1: the new item T needs to be identified, labeled SATD data set S exists, $s = \{s_1, s_2, ..., s_n\}$, and the data set corresponding to the new item T is t.

(2) Scenario 2: the new item to be identified is T, with no labeled SATD dataset. The data set corresponding to the new item is t, and each data set belongs to an item with a specific pattern.

For the above two different scenarios, the current self-admitted technical debt Detection can be divided into two categories:

(1) Based on the supervised learning method. This kind of method is mainly for scenario 1, training the detection model based on the labeled SATD data set and identifying and predicting the new items to be identified.

(2) Method based on unsupervised learning. This kind of method is mainly for scenario 2. There is no labeled SATD data set, and the prediction is based on the potential pattern detection in the new project to be identified.

The above method can be used to classify the self-admitted technical debt Detection of the target project. The research framework is shown in Figure 1.

3 Methods Based on Supervised Learning

In the past decade, various supervised learning methods have been used to identify self-admitted technical debt. Potdar et al. [7] identified 62 common SATDs by manually analyzing the annotation patterns in code annotations, including keywords, phrases, etc. The research of Bavota et al. [10] shows that the 62 SATD modes proposed above can identify SATD with high accuracy in other projects. However, this pattern detection method heavily relies on manual detection, which may bring a low detection recall rate because there may be many other different SATD patterns in other projects.

To solve the problem of the insufficient generalization ability of pattern detection methods and SATD detection, many other machine learning methods have been introduced. Maldonado et al. proposed a natural language processing method [11], Huang et al. proposed a text mining method [12], and Ren et al. proposed a method based on the convolutional neural network [13-15]. These methods show that machine learning techniques can effectively improve the performance index of SATD detection.

Based on the method of supervised learning, we should first solve the problem of data collection. At present, Maldonado et al. put forward the most widely used dataset in this research [11], which comes from 10 open source projects, including Apache antargouml, Columbia, EMF, hibernate, jedit, JfreeChart, JMeter, jruby, and squirre. These ten projects belong to different application fields with different project scales. Most of them are Java projects with obvious class imbalance. The data collection process is shown in Figure 2.

The first step is to use the eclipse plug-in tool jdeodorant to parse the source code and extract the comments. Collect the specific information of each line of code, including the start position and end position of each comment, as well as the different categories of comments. A total of 259229 open source projects were selected from these 10 open source projects, with an average of 25923 code comments per source project. Since only a small part of the code contains SATD, labeling these source codes is time-consuming and laborious.

The second step uses five heuristic filtering strategies to identify SATD and eliminate non-SATD code comments. After cleaning, irrelevant SATD comments generated by the machine are cleared, and the remaining SATD code comments to be marked are 62566 lines. Heuristic strategy can not only effectively reduce the workload of manual annotation but also improve the classification accuracy of machine learning methods. Code comments using heuristic filtering strategies include [9]:

(1) License comments. Such comments are usually added before the class declaration and describe the code's license information. Generally, SATD is not included in

such notes. Note that if such comments contain task tags (such as "todo", "fixme" or "XXX"), these comments will not be filtered because these tags usually appear with SATD.

(2) Automatically generated comments. Such comments are automatically generated by the development environment and do not describe meaningful content, so they cannot indicate SATD. For example, "auto-generated constructor stub".

(3) Javadoc comments. This annotation explains the definition and function description of entities in Java code. SATD will not normally be included unless it contains a task tag.

(4) Commented source code. This type of comment refers to some temporarily unused and commented-out content in the code. Generally, the text description related to SATD cannot be obtained intuitively, so it needs to be filtered out.

(5) Long comments. Some long notes in a project are composed of multiple single-line notes adjacent to each other. These single-line notes are separated in form but describe an overall event in content. For this type of annotation, these adjacent single-line annotations need to be combined into a complete annotation instance instead of filtering.

In step 3, after cleaning up the original data set, Maldonado and others instructed programmers to mark the data manually. Manually annotated datasets may contain personal biases, which may affect the performance of the detection model. Therefore, hierarchical sampling of data is used to reduce the impact of different programmers' annotations. To further verify the validity of the data, independent individuals of a third party are used to mark the data separately. The statistical results show that the data marked by different users have high statistical consistency. Cohen's kappa coefficient can reach 0.81, with a coefficient greater than 0.75 indicating that the error in the two people's marking is within the acceptable range.

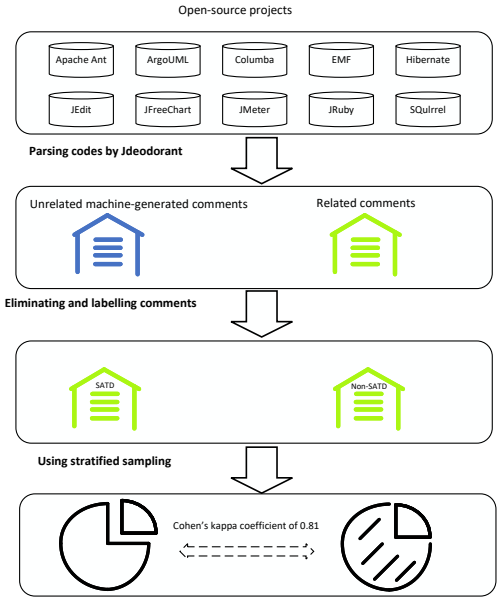


Figure 2. Data collection

Class imbalance exists in various fields. Though the ratio of the minority class is low, the minority class has greater influence. For example, when checking for lung cancer, most people tend to be healthy, but few people with potential lung cancer should be considered more. There is also the class imbalance for the SATD data set in software engineering.

There are several traditional approaches for class imbalance. (1) Oversampling methods use random sampling repeatedly for the minority class. For DL, this oversampling method may slow down the training speed and cause an overfitting problem. (2) Undersampling methods may exacerbate the problem of insufficient data and drop valuable instances. (3) Cost-sensitive methods involve reforming classification models, such as cost-sensitive learning framework, cost-sensitive dataspace weighting with Adaptive Boosting, and cost-sensitive neural network. The idea that the loss function can be adapted to account for expected costs was introduced into our proposed loss function.

Cunningham first introduced technical debt to describe where long-term code quality was traded for short-term goals [1, 8]. Previous studies showed that technical debt was inevitable, and if it could not be dealt with in time, it would reduce product quality and increase system risk [3, 6]. However, technical debt is only sometimes visible. Parts of the previous research focused on detecting technical debt by analyzing static code. Recently, Self-Admitted Technical Debt was proposed [5]. This type of technical debt is intentionally introduced by developers. Its purpose is to provide sub-optimal technical solutions for the current software code, etc., and these codes may be optimized through software refactoring. Although the ratio of SATD in the entire software project is not very high, its impact should not be underestimated. Because rule-based detection could not be used for code comments due to its natural non-structural characteristics, source Code comments with SATD cannot be automatically detected by computers.

Based on the recent survey, six different approaches at the file level were introduced to identify SATD. These approaches can be divided into two groups: (1) pattern-based approaches for textual patterns in comments; and (2) machine learning-based approaches. In previous studies, SATD detection mostly used text-mining methods [2, 8]. Deep learning has been applied to many fields in software engineering.

3.1 Model Based on Natural Language Processing Technical

In 2017, Maldonado and others first proposed using natural language processing techniques to identify SATD. The types of SATD detection include design SATD and demand SATD. The maximum entropy Stanford classifier automatically learns features from the training data set, calculates the corresponding weights, and finally judges the probabilities of different types of SATDs according to the maximum likelihood probability. In 2018, Wattanakriengkrai et al. [16] proposed combining n-gram IDF and the automatic learning algorithm auto sklearn to

find the optimal classifier. The model focuses on designing SATD and the demand for SATD. To generate a three-classification model, a random forest is introduced to transform the classification results into designing SATD, demand SATD, and no SATD. The model can reduce the cost of model training and maintenance. In 2019, a special type of SATD was discovered and defined by Maipradit et al. [17]. For this SATD, programmers must wait for additional trigger conditions to activate the code. When the external conditions are met, this SATD should be actively prompted to delete. Maipradit et al. designed a classifier based on natural language processing to recognize this kind of SATD.

3.2 Text Mining-based Detection Model

In 2018, Huang et al. [12] proposed a detection model for text mining. In this model, all code comments are preprocessed as text, all code comments are converted to the stem form of words, and irrelevant words are eliminated. The preprocessed text is used as input, the effective features of classification are obtained by feature selection, and the final SATD type is obtained by voting on the sub-classifiers.

3.3 Deep Learning Based Detection Model

Unlike the 62 intuitive patterns summarized by Potdar et al., Huang et al.'s text mining method can't interpret the detection results. In addition, the experiments of Huang et al. show that the text mining method has limited versatility and adaptability for detection in cross-project environments. Ren et al. proposed to adopt the detection model based on the convolutional neural network [13] and first determined the characteristics of five SATD reviews that affect the performance, universality, and adaptability of the SATD detection model. Their aim was to improve the accuracy of SATD prediction and the interpretability of prediction results based on deep learning. A method based on a convolutional neural network (CNN) is proposed to identify SATD in source code annotation. This method is to learn to extract the information text feature of the SATD detection task from the code annotation. This learning ability is not only important for SATD detection but also can improve the universality and adaptability of the model. A backtracking method is developed to visualize the text features learned by CNN, focusing on the influential key phrases in the input comment. These keywords contribute the most to determining whether the comment is SATD. These key phrases provide an intuitive explanation for CNN's prediction and also reveal many less obvious and less frequent SATD patterns, which are difficult to identify only through human observation. The framework is shown in Figure 3.

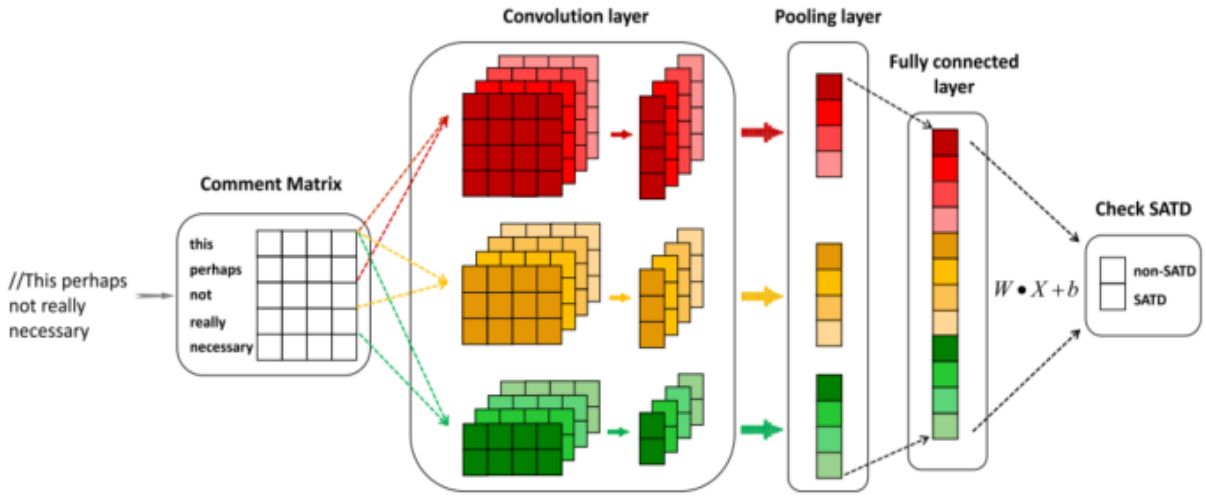


Figure 3. Deep learning-based detection model

In this model, by using multiple convolution kernels in the convolution layer, it can effectively address issues prevalent in code annotation, such as variable phrase frequency, item uniqueness, variable code annotation length, semantic diversity, class imbalance, and more. The implementation method employs convolution kernels of varying sizes to learn the semantic representation of different texts. The output results of different convolution kernels are concatenated into a vector, and a linear classifier is used for vector classification. The classification results are SATD and non-SATD, representing a binary classification problem.

In 2022, in view of the effectiveness of the deep learning detection model, Qu et al. conducted empirical research on common deep learning models on the open-source blockchain project. Its basic framework is shown in Figure 4.

By fine-tuning the training process on the pre-training model Bert, the weighted loss function is introduced to solve the class imbalance problem. The research results on the open-source blockchain project show that the performance of this method is better than the convolutional neural network model proposed by Ren et al.

In 2022, Qu et al. [18] further verified the effectiveness

of the deep learning method using the interpretability technical based on gradients to carry out empirical research on convolutional neural networks proposed by Ren et al. The used technologies include saliency maps, integrated gradients, etc. The research results showed that the classification method based on deep learning could cover the manual annotation mode. The interpretability analysis process is shown in Figure 5.

The interpretability of deep learning models is of paramount importance as it enables researchers and practitioners to understand and trust the model's predictions. It provides insights into how the model arrives at its decisions, which is crucial for debugging, improving model performance, and ensuring transparency and accountability in applications. Techniques such as saliency maps, integrated gradients, and others help to visualize the influence of input features on the model's predictions, thereby enhancing interpretability. By gaining a deeper understanding of the model's behavior, developers can make informed decisions regarding model selection, feature engineering, and other aspects of model development and deployment.

In the weighted convolutional neural network, the gradient of each input neuron is inversely calculated for the classification result. The gradient with large change is the gradient with great influence on the classification result. Comparison of different SATD detection methods is shown in Table 1.

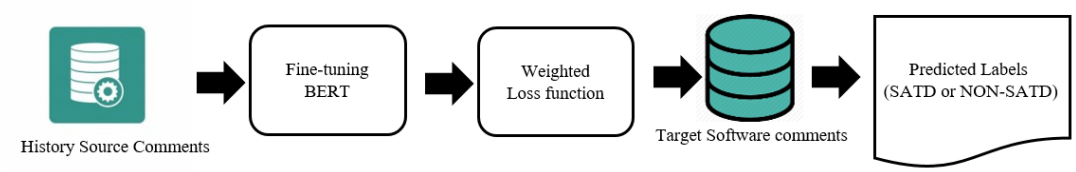


Figure 4. The framework of SATD detection in blockchain projects

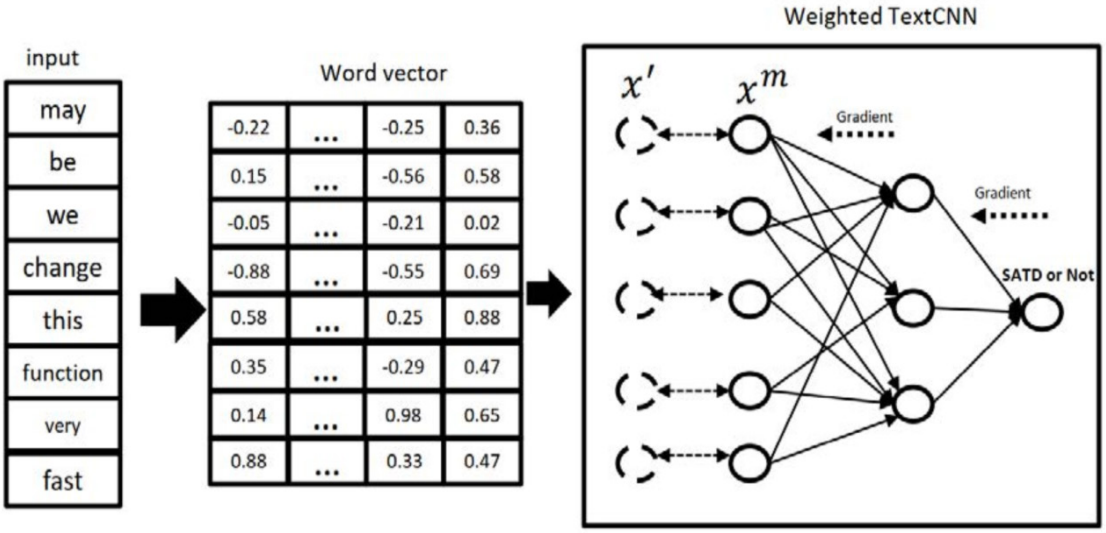


Figure 5. The interpretability analysis process

Table 1. Comparison of different SATD detection methods

Method	Advantages	Disadvantages
Pattern Detection	Simple and intuitive	Low recall rate, heavily dependent on manual analysis
Natural Language Processing (NLP)	High accuracy, automated feature extraction	Requires large amount of labeled data, complex model training
Text Mining	Efficient feature extraction, good generalization ability	Limited by feature engineering, low interpretability
Deep Learning	High accuracy, strong generalization ability	Requires large amount of data, black box problem

4 Method Based on Unsupervised Learning

The unsupervised model is very convenient to build and use compared with the supervised model. The unsupervised model mainly identifies the existence of SATD by summarizing different technical debt patterns. In 2014, Potdar et al. proposed 62 modes by manually reading code comments, laying the foundation for unsupervised learning. They determined whether the code comment has SATD according to whether the mode in the target data set appears. However, this model needs to be more specific, resulting in a low recall rate in the actual project.

The unsupervised learning approaches can leverage various patterns and heuristics to identify SATD. For instance, patterns might include specific comment tags like "todo: needs documentation" and "todo: not complete," or particular keywords present in the code. Heuristics may involve predefined rules for common technical debt types, such as documentation debt and requirement debt.

In 2018, Passos et al. [19] proposed that the notes containing the modes "todo: needs documentation," "todo: not documented," and "please document" usually include document debt, while the notes containing the modes "not implemented", "todo: not complete" and "not yet supported" usually include demand debt.

In 2019, Guo et al. [20] found that there is a strong connection between task tags and SATD according to the situation of task tags in Java projects. Typical task tags include "todo", "fixme", "hack" and "XXX". These task tags are closely related to the Java project and the corresponding integrated development environment. Guo et al. conducted empirical research in common open-source projects and collected other Java projects for research. The research results show that the method mat based on fuzzy matching can achieve, or even very close to, the detection performance of the convolutional neural network. However, this method has obvious defects. For example, these task tags may exist in Java projects, while typical task tags may not exist in some projects. The programming languages of related projects are shown in Table 2.

Table 2. The programming languages of related projects

Dataset	Project	Language
Maldonado et al.	Ant	JAVA
	ArgoUml	JAVA
	Columba	JAVA
	EMF	JAVA
	Hibernate	JAVA
	Jedit	JAVA
	JFreeChart	JAVA
	Jmeter	JAVA
	JRuby	JAVA
	Squirrel	JAVA
Guo et al.	Dubbo	JAVA
	Gradle	JAVA
	Groovy	JAVA
	Hive	JAVA
	Maven	JAVA
	Poi	JAVA
	SpringFramework	JAVA
	Storm	JAVA
	Tomcat	JAVA
	Zookeeper	JAVA

In 2022, Qu et al. [21] conducted empirical research on open-source blockchain projects, including Bitcoin, Ethereum, solidity, fabric, and Chia. Through manual annotation of code comments, it is finally found that in these projects, typical task tags do not always exist in each comment, and the fuzzy matching algorithm cannot effectively identify SATD in these projects. The specific empirical research results are shown in Table 3.

Table 3. Empirical research results

Project	SATD Types	#Count	todo	fixme	xxx	hack	#All Tags	#Percent
bitcoin	SATD	1201	143	0	0	5	148	12.32%
	WITHOUT_SATD	28205	1	2	7	7	17	0.06%
Ethereum	SATD	708	120	2	6	4	132	18.64%
	WITHOUT_SATD	40780	72	0	0	18	90	0.22%
Diem	SATD	446	212	1	8	6	227	50.9%
	WITHOUT_SATD	12411	112	2	9	10	133	1.07%
solidity	SATD	503	3	0	0	0	3	0.6%
	WITHOUT_SATD	1267	1	0	0	0	1	0.08%
fabric	SATD	806	386	7	12	4	409	50.74%
	WITHOUT_SATD	36392	170	1	72	16	259	0.71%
chia	SATD	207	71	0	0	3	74	35.75%
	WITHOUT_SATD	2504	7	0	0	2	9	0.36%

5 Performance Evaluation Index Analysis

The problem of self-admitted technical debt detection is often transformed into a binary classification problem. The classification results of code annotation include SATD and non-SATD. From the perspective of binary classification, the classification results of annotation detection can be divided into the following four cases.

A. For an annotation identified as containing SATD, it does contain SATD. This result belongs to TP (true positive);

B. For an annotation identified as containing SATD, it does not contain SATD. This result belongs to FP (false positive);

C. For an annotation identified as not having SATD, it does contain SATD, and this result belongs to FN (false negative);

D. For an annotation identified as not having SATD, it does not contain SATD. This result belongs to TN (true negative)

Based on the above four situations, the relevant confusion matrix is designed, as shown in Table 4 below.

Table 4. Confusion matrix

Actual value / Predicted value	SATD	NON-SATD
SATD	True SATD	False NON-SATD
NON-SATD	False SATD	True NON-SATD

For the calculation results of all data samples on the test data set, calculate the index values according to the following formula.

(1) precision, the proportion of all instances with SATD prediction results, including real SATD samples

$$precision = \frac{True\ SATD}{True\ SATD + False\ SATD}$$

(2) recall, the proportion of all real SATD samples correctly predicted as SATD samples

$$recall = \frac{True\ SATD}{True\ SATD + False\ NON - SATD}$$

(3) F1 − measure, this index is the harmonic average of precision and recall, which can objectively reflect the comprehensive performance of the Detection model

$$F1 - measure = \frac{2 * precision * recall}{(precision + recall)}$$

The above three indicators are most used in the performance comparison of detection methods. Values of these indicators range from 0 to 1; the larger the value range of these indicators, the better the performance of detection methods. Due to the class imbalance problem in SATD dataset, F1 − measure is widely used to evaluate the performance of Detection methods.

(4) Accuracy: the proportion of samples correctly classified as SATD and non-SATD in all classification results.

6 Evaluation Data Set Analysis

To accurately evaluate the effect of SATD detection model, performance evaluation needs to be carried out on the benchmark data set. However, it is essential to acknowledge the challenges and limitations of the existing benchmark datasets, such as the potential biases introduced by manual labeling and the lack of diversity in project domains and programming languages. Currently, the most commonly used dataset is extracted from the open-source Java project by Maldonado and others using the tool jdeodorant. The basic information of the dataset-m is shown in Table 5 [4].

The first table in the figure shows the data set name, and the second column represents the project name; the third column represents the number of lines of extracted code comments; the fourth column represents the number of remaining effective code comment lines after filtering; the fifth column represents the number of SATD code comments; the sixth column represents the proportion of SATD in all code comments; the seventh column represents the number of contributors; the eighth column represents the number of class entities. The ninth column represents the number of code lines.

In 2019, Guo et al. [4, 20], in order to more comprehensively evaluate their unsupervised fuzzy matching algorithm mat, collected and sorted another dataset dataset-g. The data collection method of this dataset is consistent with that of dataset-m, which comes from another 10 open-source Java projects. The generalization ability of the mat method in the actual project is analyzed from the new data set, and the larger data set can better reflect the statistical results. The ten datasets are dubbo-2.7.4, gradle-5.6.3, groovy-2.5.8, hive-3.1.2, maven-3.6.2, poi-4.1.1, springframework-5.2.0, storm-2.1.0, tomcat-9.0.27, and zookeeper-3.5.6. There are 266,980 samples in the original data set. After filtering, 81,260 samples are obtained. The proportion of SATD is 1.12%.

Qu et al. [21] conducted an empirical study on SATD in blockchain projects in 2022 and collected data sets of common open-source blockchain projects. The data set collection method is consistent with the dataset-m collection method. The brief information on each project is shown in Table 6. In the figure, the first column represents the project name, the second column represents the version number adopted, the third column represents the number of code lines, and the fourth column represents the programming language adopted. The fifth column represents the number of GitHub projects plus stars. These blockchain projects do not use Java programming language, and this data set can provide research objects that are more suitable for the needs of practical blockchain projects.

Table 5. The basic information of the dataset-m

Dataset	Project	#Comments	After filtering	#SATD	% of SATD	Cont.	# of classes	KLOC
Maldonado et al. collected (Dataset-M)	Ant	21,587	3,052	102	0.47%	74	1,475	115
	ArgoUML	67,716	5,426	969	1.43%	87	2,609	926
	Columba	33,895	4,090	128	0.38%	10	1,711	155
	EMF	25,229	2,585	74	0.29%	30	1,458	228
	Hibernate	11,630	2,492	377	3.24%	314	1,356	703
	JEdit	16,991	4,644	195	1.15%	57	800	310
	JFreeChart	23,474	2,494	101	0.43%	19	1,065	317
	JMeter	20,084	4,148	282	1.40%	41	1,181	354
	JRuby	11,149	3,652	383	3.44%	374	1,486	841
	SQuirrel	27,474	4,473	201	0.73%	40	3,108	708
	Total	**259,229**	**37,056**	**2,812**	**1.08%**	**1,046**	**16,249**	**4,657**
We collected (Dataset-G)	Dubbo	5,875	1,649	85	1.45%	255	2,532	141
	Gradle	15,901	3,324	321	2.02%	409	13,541	406
	Groovy	14,199	4,435	249	1.75%	284	2,729	181
	Hive	81,127	29,340	1,046	1.29%	192	15,463	1,257
	Maven	5,448	1,219	136	2.50%	87	1,158	84
	Poi	45,666	15,033	618	1.35%	12	4,793	406
	SpringFramework	42,574	7,712	98	0.23%	401	14,686	654
	Storm	12,258	3,639	92	0.75%	304	4,787	282
	Tomcat	37,038	12,218	287	0.77%	31	4,120	335
	Zookeeper	6,894	2,691	63	0.91%	93	1,322	87
	Total	**266,980**	**81,260**	**2,995**	**1.12%**	**2,068**	**65,131**	**3,833**
Total	**-**	**526,209**	**118,316**	**5,807**	**1.10%**	**3,114**	**81,380**	**8,490**
Average	**-**	**26,310**	**5,915.8**	**290**	**1.10%**	**156**	**4,096**	**425**

Table 6. The brief information on each project

Project	Release	#Line of code	Languages	Stars
bitcoin	22	221,466	C++, Python	58.9K
Ethereum	1.10.12	394,246	go	33.3K
diem	1.0.2	227,505	Rust	16.2K
solidity	0.8.10	224,404	C++, Solidity, Python	13K
fabric	2.3.3	1,109,729	go	12.8K
chia	1.2.7	62,232	Python	9.5K

7 Future Research Directions

There are many challenges in the current SATD detection research [9]. When building SATD detection, some models treat it as a binary classification problem. However, in actual projects, different types of SATD may have different concerns. Some models did not consider class imbalance when designing the SATD detection model. Although some detection models can achieve relatively high F1 values on the experimental data set, considering the complexity of the industry, it is still unable to be effectively migrated to the project. The model based on machine learning also has the problem of super parameter optimization, especially the detection model based on deep learning. In the actual project, the model must be optimized according to the characteristics of different projects. In addition, the reliability of the data set also has some problems. Yu et al. [22] found the real category of about 426 annotation instances in the data set provided by Maldonado et al. It was wrongly marked. According to the review of SATD detection and the existing challenges, the research group believes that the following aspects need more research attention.

(1) Provide a unified definition of SATD and build a standardized SATD data set. The division of SATD is based on the programmer's understanding of code

comments, and it is manually classified. This classification has strong subjectivity and bias. Different programmers may have different perceptions of different SATD types. For example, there may be some fuzziness in the manual classification of design debt and demand debt. Therefore, it is necessary to establish a unified classification standard, which can effectively reduce the difficulty of manual annotation and improve the quality of data sets. At the same time, in the process of data set construction, it should contain as much information as possible, such as the introducer, survival time, remover, SATD association code, etc., which will help to provide important support for subsequent SATD research.

(2) Study the recommended model of SATD removal. At present, the Detection model only provides the category prediction of SATD. However, more SATD removal suggestions need to be provided in the project. At present, there are few studies in this field.

(3) Development of SATD management tools. Although SATD widely exists in various projects, there is currently a lack of complete and systematic management SATD tools, including Detection, classification, importance suggestions, etc. It can only scan whether there is SATD of outstanding debts before software development.

(4) Research on SATD detection method. From pattern detection to deep learning, the accuracy of SATD detection is constantly improving, and the current pre-training model has also made greater progress in text processing. How to integrate domain knowledge and pre-training model to improve the accuracy of SATD detection is also a new application research direction.

8 Acknowledgments

This work was supported by Guangxi Key Laboratory of Trusted Software (No. kx202046), 2023 Higher Education Scientific Research Planning Project of China Society of Higher Education (23PG0408), 2023 Philosophy and Social Science Research Programs in Jiangsu Province (2023SJSZ0993), Nantong Science and Technology Project (No. JC2023070). This work is sponsored by the Cultivation of Young and Middle-aged Academic Leaders in "Qing Lan Project" of Jiangsu Province.

References

[1] W. Cunningham, The WyCash portfolio management system, *Acm Sigplan Oops Messenger*, Vol. 4, No. 2, pp. 29-30, April, 1993. https://doi.org/10.1145/157710.157715

[2] Y. Qu, S. Huang, Y. Yao, A survey on robustness attacks for deep code models, *Automated Software Engineering*, Vol. 31, No. 2, pp. 1-40, November, 2024. https://doi.org/10.1007/s10515-024-00464-7

[3] Y. Qu, S. Huang, P. Nie, A review of backdoor attacks and defenses in code large language models: Implications for security measures, *Information and Software Technology*, Vol. 182, Article No. 107707, June, 2025. https://doi.org/10.1016/j.infsof.2025.107707

[4] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study, *ACM Transactions on Software Engineering and Methodology*, Vol. 30, No. 4, pp. 1-56, October, 2021. https://doi.org/10.1145/3447247

[5] B. Fluri, M. Wursch, H. Gall, Do code and comments co-evolve? On the relation between source code and comment changes, *14th Working Conference on Reverse Engineering (WCRE 2007)*, Vancouver, BC, Canada, 2007, pp. 70-79. https://doi.org/10.1109/WCRE.2007.21

[6] H. Malik, I. Chowdhury, H. Tsou, Z. Jiang, A. Hassan, Understanding the rationale for updating a function's comment, *2008 IEEE International Conference on Software Maintenance (ICSM'08)*, Beijing, China, 2008, pp. 167-176. https://doi.org/10.1109/ICSM.2008.4658065

[7] A. Potdar, E. Shihab, An exploratory study on self-admitted technical debt, *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, Victoria, BC, Canada, 2014, pp. 91-100. https://doi.org/10.1109/ICSME.2014.31

[8] S. Wehaibi, E. Shihab, L. Guerrouj, Examining the impact of self-admitted technical debt on software quality, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, 2016, pp. 179-188. https://doi.org/10.1109/SANER.2016.72

[9] Z. Guo, S. Liu, T. Tan, Y. Li, L. Chen, Y. Zhou, B. Xu, Self-admitted Technical Debt Research: Problem, Progress, and Challenges, *Journal of Software*, Vol. 33, No. 1, pp. 26-54, January, 2022. https://doi.org/10.13328/j.cnki.jos.006292

[10] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, Austin, TX, USA, 2016, pp. 315-326.

[11] E. Maldonado, E. Shihab, 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, November, 2017. https://doi.org/10.1109/TSE.2017.2654244

[12] Q. Huang, E. Shihab, X. Xia, D. Lo, S. Li, Identifying self-admitted technical debt in open source projects using text mining, *Empirical Software Engineering*, Vol. 23, No. 1, pp. 418-451, February, 2018. https://doi.org/10.1007/s10664-017-9522-4

[13] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, J. Grundy, Neural network-based detection of self-admitted technical debt: From performance to explainability, *ACM Transactions on Software Engineering and Methodology*, Vol. 28, No. 3, pp. 1-45, July, 2019. https://doi.org/10.1145/3324916

[14] A. Rajput, M. Yadav, S. Yadav, M. Chhabra, A. P. Agarwal, Patch-Based Breast Cancer Histopathological Image Classification using Deep Learning, *International Journal of Performability Engineering*, Vol. 19, No. 9, pp. 607-623, September, 2023. https://doi.org/10.23940/ijpe.23.09.p6.607623

[15] S. Singh, A. Sharma, State of the Art Convolutional Neural Networks, *International Journal of Performability Engineering*, Vol. 19, No. 5, pp. 342-349, May, 2023. https://doi.org/10.23940/ijpe.23.05.p6.342349

[16] S. Wattanakriengkrai, N. Srisermphoak, S. Sintoplertchaikul, M. Choetkiertikul, C. Ragkhitwetsagul, T. Sunetnanta, H. Hata, K. Matsumoto, Automatic classifying self-admitted technical debt using n-gram idf,

2019 26th Asia-Pacific Software Engineering Conference (APSEC), Putrajaya, Malaysia, 2019, pp. 316-322. https://doi.org/10.1109/APSEC48747.2019.00050

[17] R. Maipradit, C. Treude, H. Hata, K. Matsumoto, Wait for it: Identifying "On-Hold" self-admitted technical debt, *Empirical Software Engineering*, Vol. 25, No. 5, pp. 3770-3798, September, 2020. https://doi.org/10.1007/s10664-020-09854-3

[18] G. Zhuang, Y. Qu, L. Li, X. Dou, M. Li, An Empirical Study of Gradient-based Explainability Techniques for Self-admitted Technical Debt Detection, *Journal of Internet Technology*, Vol. 23, No. 3, pp. 631-641, May, 2022. https://doi.org/10.53106/160792642022052303021

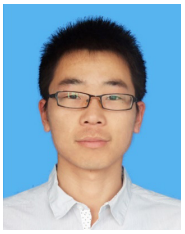
[19] A. Passos, M. Farias, M. Neto, R. Spínola, A study on Identification of documentation and requirement technical debt through code comment analysis, *SBQS '18: Proceedings of the XVII Brazilian Symposium on Software Quality*, Curitiba, Brazil, 2018, pp. 21-30. https://doi.org/10.1145/3275245.3275248

[20] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, B. Xu, MAT: A simple yet strong baseline for identifying self-admitted technical debt, *arXiv preprint arXiv*: 1910. 13238, October, 2019. https://arxiv.org/abs/1910.13238

[21] Y. Qu, T. Bao, X. Chen, L. Li, X. Dou, M. Yuan, H. Wang, Do we need to pay technical debt in blockchain software systems? *Connection Science*, Vol. 34, No. 1, pp. 2026-2047, June, 2022. https://doi.org/10.1080/09540091.2022.2067125

[22] Z. Yu, F. M. Fahid, H. Tu, T. Menzies, Identifying self-admitted technical debts with Jitterbug: A two-step approach, *IEEE Transactions on Software Engineering*, Vol. 48, No. 5, pp. 1676-1691, May, 2022. https://doi.org/10.1109/TSE.2020.3031401

Biographies



Xianzhen Dou received the M.S. degree in School of Electronics and Information from Nantong University in China in 2013. Since 2019, he has been a lecture with Information Engineering Institute, Jiangsu College of Engineering and Technology. His research interests include software engineering, and machine learning.

Long Li received his Ph.D. degree from Guilin University of Electronic Technology, Guilin, China in 2018. He is now a lecturer at the School of Computer Science and Information Security, Guilin University of Electronic Technology, Guilin, China. His research interests include cryptographic protocols, privacy-preserving technologies in big data and IoT.

Yubin Qu received the M.S. degree in Computer Science and Technology from Henan Polytechnic University in China in 2008. Since 2009, he has been a lecture with Information Engineering Institute, Jiangsu College of Engineering and Technology. His research interests include software maintenance, software testing, and machine learning.