Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 18, no. 3, March 2022, pp. 149-157
DOI: 10.23940/ijpe.22.03.p1.149157

Empirical Research for Self-Admitted Technical Debt Detection in Blockchain Software Projects

Yubin Qu[a,b], W. Eric Wong[c,*], and Dongcheng Li[c]

[a]*Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, 541004, China*
[b]*School of Information Engineering, Jiangsu College of Engineering and Technology, Nantong, 226001, China*
[c]*Department of Computer Science, University of Texas at Dallas, Richardson, 75080, USA*

Abstract

Blockchain technology has been used in various fields including digital currencies, distributed storage, and more. The issue of SATD detection for open source blockchain software systems has not been studied. We provide an in-depth analysis of the code comments of open blockchain source software projects. A pre-trained model based on cost-sensitivity is proposed, which is compared with the baseline model on several evaluation metrics. The results were statistically analyzed. The results show that the pre-trained model based on natural language understanding is able to achieve better classification performance. Our method improves 102% over the baseline method in the F1 metric.

Keywords: self-admitted technical debt; blockchain; pretrained model

1. Introduction

In 2008 Bitcoin technology was created as a new type of digital currency [1]. Currently Bitcoin has become the world's most important digital currency. This digital currency uses blockchain technology, which makes transactions between different entities transparent, distributed, and has the important characteristic of being tamper-proof. With this, blockchain technology extends to various fields including the Internet of Things [2], smart contracts [3,4], supply chain management [5], and many other applications. To quickly develop and deploy new blockchain software systems, many application developers directly use software frameworks shared on GitHub, such as Bitcoin[1], go-ethereum[2], etc. Software framework developers focus on how to implement the various complex algorithms in blockchain technology and refactor the code for algorithm upgrades. Application developers are concerned with directly calling open source blockchain technology frameworks to implement their own business.

Developers of blockchain frameworks are expected to deliver high quality bug-free software frameworks. This is very critical as studies have shown that failures caused by software bugs can have very severe consequences including property damage, monetary loss, etc. [6,7|. However, the reality is that due to time constraints, testing resource constraints, or other factors, blockchain frameworks often have pending code or sub-optimal code known as technical debt [8]. Technical debt was first introduced by Cunningham in 1993 and describes this situation [9]. Technical debt that is actively introduced by the software framework development programmer in code comments is known as self- admitted technical debt (SATD) [10-14]. Qu et al. have opened up a dataset of self-admitted technical debt for blockchain software systems on GitHub3, and their research shows that there are various technical debts for blockchain software systems. However, they did not conduct further in-depth research on the dataset to explore how to detect self-admitted technical debt more effectively. Guo et al. show through empirical research that the detection of self-introduced technical debt can be effectively performed using a simple but extremely effective method. This classification method is based on the presence of a large number of indicator tags in the

[1] https://github.com/bitcoin/bitcoin
[2] https://github.com/ethereum/go-ethereum
[3] https://github.com/qyb156/blockchain_SATD_research

* Corresponding author.
E-mail address: ewong@utdallas.edu

code annotations. Whether this conclusion holds true for blockchain software systems is open to investigation [15].

To help application developers detect self-admitted technical debt in code comments more effectively, we have conducted an in-depth exploration of open source datasets for blockchain software systems. The main problem that our research focuses on is how to design more effective systems for detecting self-admitted technical debt.

The remainder of this paper is structured as follows. Section 2 presents the related work of our study, including the research works on blockchain software projects and research works on self-admitted technical debt. Section 3 presents our research approach. Section 4 depicts the experimental setup. We also compare the findings in our work with the findings in previous work. Finally, Section 6 concludes our study and presents future work.

2. Related Work

2.1. Blockchain Software Projects

Blockchain technology is a decentralized, tamper-proof, and transparent chain storage technology [16]. Blockchain technology stores a series of blocks, each of which contains transaction information as well as unique hash information. A hash value is generated by a specific algorithm [17]. Each block's hash value is calculated based on the block's data, the current timestamp, and the hash value of the previous block. When the information in one of the blocks changes, the information in the following block immediately will also change. This prevents the blockchain from being tampered with maliciously [18]. In a large blockchain network, tampering with the blockchain through collusion is essentially impossible [19]. All nodes in a blockchain network system can simultaneously go looking for the next writable block. This search process is known as mining. The computation process needs to follow a specific algorithm and is called verifying completion when specified requirements are met. The node that finds a block is rewarded with a certain amount of tokens. The difficulty of mining is determined by a predefined algorithm [20]. There is no control center in a blockchain network and the underlying principle is that each participant is equal and together they make up the blockchain network system. The completion of transactions between the network nodes is achieved through a mediation mechanism. A prerequisite for a block to be validated is that the majority of participants accept the current block as trustworthy and genuine. Therefore, once a block has been verified, it is almost impossible to tamper with it.

The most successful application of blockchain technology in the field of digital currencies is bitcoin. Bitcoin uses peer-to-peer technology to achieve a decentralised operation. Managing transactions as well as issuing currency is done using a collaborative approach of participants. The bitcoin core project is currently one of the most popular open source projects on GitHub. The project provides users with an interface service that allows them to quickly call the digital currency.

Another innovative application of blockchain technology is the smart contract system. This system fixes the transaction information between buyers and sellers in a self-executing contract via software code. Inside a decentralised environment, trusted anonymous participants who are allowed to run the contract can do so, when the preconditions are met. As the contract is deployed in a distributed environment, it is also transparent, traceable and untamperable. Go-ethereum is one of the most popular open source blockchain software projects in GitHub. Go-ethereum is the official golang implementation of the Ethereum protocol.

2.2. Self-Admitted Technical Debt Detection

Technical debt was first introduced by Cunningham in 1993 and describes this situation [9]. Potdar and Shihab introduce a new way to identify technical debt. This technical debt is derived from code annotations and is referred to as self-admitted technical debt [10]. It has been shown that SATD is prevalent in software projects and can be used to detect different types of technical debt (e.g. design, defects, etc.) [12]. The intuition tells us that the presence of software code with technical debts inevitably affects software quality and even brings about software defects [21]. Sultan et al. show through empirical studies in open source software projects that software defects are not related to SATD. However, the existence of SATD makes the evolution of future software code more complex [14].

Since SATD can make software development more complex and can introduce potential software defects, SATD detection has long been a focus of academic research. To predict the presence of SATD in a target project, it is common practice to first obtain code comments in historical software projects in order to construct a training dataset [22]. Tools for obtaining code annotations include JDeodorant [23] et al. Next, the training dataset is manually annotated by experienced programmers. The model is trained on this labelled data using techniques such as pattern recognition, natural language processing techniques, deep learning techniques, and so on. Finally, during the software development process, the training

model is called on the target code comments to obtain the classification labels (SATD, or non-SATD). The SATD detection problem can therefore be reduced to a typical machine learning binary classification problem.

Since the introduction of SATD, a number of medium automation techniques have been used to detect SATD, and we will now highlight a few common detection methods.

Potdar et al. et al. proposed a pattern matching approach to detect SATD [10]. Specifically, they read 101,762 code annotations and summarized 62 SATD patterns. Each pattern is a word or phrase (e.g., stupid, get rid of this), as these phrases often appear in code comments containing SATD. The process of detection is to see if a particular pattern exists in the target code annotation. If a pattern is present, it is marked as SATD, otherwise it is marked as non-SATD. This method has a low recall and can identify only some SATD.

To address the limitations of pattern matching methods, Maldonado et al. [13] proposed natural language processing techniques to detect SATD. They collected datasets from 10 open source software projects. They filtered and manually labelled the dataset as SATD or non-SATD. The maximum entropy classifier was trained to detect SATD. Their results show that better classification results can be achieved with a small amount of training data.

Huang et al. proposed the use of text mining to detect SATD [24]. They first pre-processed the code comments using natural language processing techniques, including word separation, deactivation, and word stemming. To reduce the problems associated with the curse of dimensionality, they performed feature selection using information gain techniques. Next on the training dataset, Naïve Bayes Multinomial models were trained. Predictions were made for each code comment for a particular software project. Their experiments showed that the method outperformed pattern matching methods.

As the text mining method proposed by Huang et al. is not interpretable, Ren et al. proposed a *TextCNN-based* SATD detection method [22,25]. The method learns domain-related semantic vectors from the training dataset, and the deep semantics of the code annotations can be obtained by stitching the different semantic vectors through multiple convolutional kernels. Finally, the semantic vectors are classified by a fully-connected model. The model can also learn interpretable patterns from semantic vectors, thus reducing the workload of SATD pattern annotation. This method is the state-of-the-art approach that works well in performance, generalizability, and explainability.

Guo et al. conducted an observational study of open datasets and found that four representative task marker phrases (e.g., todo, fixme, XXX, hack) were prevalent in the code comments of JAVA software projects, and that these phrases could be used to classify SATD. They therefore propose a simple heuristic taxonomy method which classifies the target code comments by matching the flag phrases. This is done by first pre-processing the code comments in the target items. All code comments are converted to lower case and English characters are retained. The next step is to perform fuzzy matching, and as long as any of the four flag phrases are present in the code annotation, the code comment is considered with SATD. Their experimental results show that the method is effective in classifying SATD and can be used as a baseline method for SATD classification.

2.3. Bidirectional Encoder Representations from Transformers

Bidirectional Encoder Representations from Transformers (BERT) [26] is a pre-training technique proposed by Google for natural language processing. BERT has shown excellent performance on a variety of tasks in natural language understanding. Unlike context-independent models such as word2vec, BERT is able to use different vector representations for different words in different contexts. For example, water is represented as the same word vector for "plants need water" and "there is water in the report". In BERT, water is represented differently in the two sentences.

3. Our Research Approach

3.1. The Motivation of using Pre-Trained Models

Developers of blockchain system frameworks will often introduce SATD in code comments to remind themselves to repay that debt in subsequent software development. For example, in the Bitcoin project, the code comment "wouldn't support bip 68" implies that there is a technical debt of compatibility. In the project, the code comment "todo: why do we flip around the recovery id?" implies that there is a technical debt of defect. To correctly predict code comments, the classifier needs to decompose the internal features of the code comments in order to provide a decision basis for prediction.

Based on the simple code annotation example we have provided, we need to address several key issues in order to make a correct prediction about whether the code comment contains SATD or not. There are a number of indicative words that can

be used to mark the presence of SATD in code comments, for example, todo, xxx, hack, fixme, etc. Other words or phrases, such as error, perhaps, maybe, error, etc., are also likely to indicate the presence of SATD in a code comment, so pattern matching and text mining [24] cannot be used to effectively classify code comments. Second, code comments have an indeterminate sentence length. Text mining techniques cannot effectively handle code comments with inconsistent sentence lengths. Variable-length code comments are also a challenge for other classifiers. Third, neither text mining nor the classifiers based on convolutional neural networks can effectively address the problem of contextual semantics in code comments. Ren et al. use convolutional neural networks to learn deep semantics in code comments using word embeddings, which are able to obtain word vector representations from pre-trained datasets. However, it was unable to address the potential contextual semantic issues in the code annotations. Finally, we also need to consider the class imbalance that naturally exists in the training dataset as well as in the target dataset. The code comments containing SATD represent a very small proportion. Most of the code comments are without SATD. This is something we should be particularly aware of when selecting classifiers.

To address the above challenges, we use a cost-sensitive pre-trained model for SATD detection. BERT was used as our classifier. Large-scale pre-trained models, such as BERT, have recently achieved great success and have become a milestone in the field of artificial intelligence. With sophisticated and huge model parameters, pre-trained models are able to capture knowledge from both large scale labelled and unlabelled datasets. By storing this knowledge in a complex network and fine-tuning on a specific task, downstream classification tasks that have acquired rich semantic knowledge can achieve better classification results. BERT is able to better learn the contextual semantics of the code annotations and thus classify SATD more accurately. We use a cost-sensitive weighted loss function to the class imbalance problem in SATD detection, giving more weight to code comments containing SATD.

3.2. Approach

Based on the open dataset shared by ourselves for blockchain software projects, a new type of SATD, resource debt, was found. Moreover, code comments with task tags do not completely determine whether the code comment contains SATD. Thus we argue that, due to the presence of complex natural language features in the code comments, better classification results should be achieved using the currently widely used pre-trained models. To address the class imbalance problem, we use a weighted cross-entropy loss function in our pre- trained model.

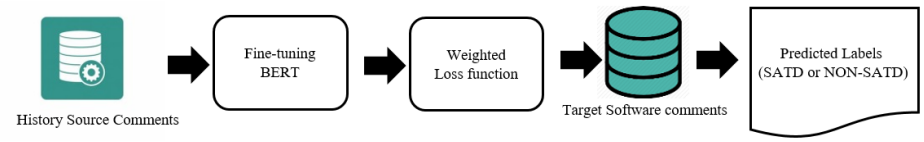

Figure 1. The framework of our approach

As shown in Figure 1, our approach includes three main steps: fine-tune BERT on history source comments, train the model based on weighted loss function, and predict labels on target software comments. Predicting whether a code annotation contains SATD can be reduced to a binary classification problem. The core of our approach is to fine-tune the BERT model using historical code comment data. In the training phase of the model, we select n of the projects as the training dataset and the remaining ones as the test dataset. The BERT is fine-tuned on the training dataset according to the SATD distribution, and finally a classification model is obtained based on a weighted loss function. Finally, the performance of our classifier is tested on the test dataset.

3.2.1. The Architecture of BERT

The main structure of the BERT is the transformer. Each layer of the BERT is made up of multiple standard encoders. Each encoder is generated by a multi-Head-Attention, Layer Normalization, feedforward, and Layer Normalization. This results in the ability to predict the token of the mask based on the context of the input text, thus capturing the bidirectional relationship. By using the self-attention mechanism, problems such as the inability to accelerate serialized data in the GPU can be solved; in addition, since the correlation between each word and all words in the sentence is calculated, the length dependency problem in RNN networks can be effectively solved. This enables the capture of the semantics of each word in the code comment. Through the multi-head attention mechanism, the semantic representation of the input sentence is obtained from multiple perspectives. By concatenating multiple outputs into a single semantic vector, a deep semantic representation of the sentence is obtained. Inside the BERT model, the multilayer encoder obtains a deep representation of the surface, phrase, syntactic and semantic levels of the sentence.

The next step is to fine-tune the model on the labelled code comment data. A [cls] symbol is added in front of each code comment input to indicate whether the code comment contains SATD. This symbol is used to indicate whether the code annotation contains SATD, and the output of the vector corresponding to this symbol is used as the semantic representation of the entire code comment. We can assume that this semantic representation learns the relevant semantic information from the code comments.

3.2.2. Weighted Loss Function

Assuming that the input code comment belongs to class *i* and its corresponding true label is *t*, the corresponding output probability is *y*. The cross-entropy function loss = $-\sum t_i \log(y_i)$. This loss function calculates the degree of deviation between the predicted probability and the true value of the code comments in the training set. However, this cross-entropy loss function cannot be used directly for SATD detection. In the dataset we studied, the code comments containing SATD were much smaller than the code comments without SATD.

In our study, we propose to use the weighted loss function in BERT. We set a weight for each class, assuming that the number of code comments containing SATD in the training dataset is *m* and the number of code comments without SATD is *n*. Then, we define the weight of code comments containing SATD as n/(m+n) and the weight of code comments without SATD is m/(m+n). Then the loss function is denoted as:

$$\text{loss} = -(\sum(\frac{m}{m+n} t_{non-SATD} \log(y_{non-SATD}) + \frac{n}{m+n} t_{SATD} \log(y_{SATD}))).$$

In this improved loss function, we give more weight to code comments labelled to be SATD, penalizing those labelled to be non-SATD.

3.2.3. Model Training

The training process for the code comments is a multi-round optimisation process in which the loss function of the predicted versus true values of the code comments is calculated at each epoch of optimisation, and the network parameters for each encoder are updated according to the back propagation algorithm. The gradient optimisation algorithm uses the Adam algorithm.

3.2.4. SATD Prediction

Once we have fine-tuned the BERT model, we can classify the code comments in the test dataset. The pre-processed code comments are fed into our trained model and the categories with higher prediction probability are used as prediction categories for the code comments.

4. Experimental Design

4.1. Data Collection

To investigate in depth the performance of classifiers for SATD detection in blockchain software projects, we conducted an empirical study on an open dataset. We have collected the open source blockchain framework projects on GitHub. These projects cover multiple fields, from cryptocurrency to distributed storage and medical health. The selection of these projects follows the development sequence of blockchain technology, including both early cryptocurrency projects and the application framework of blockchain technology that has been more active in recent years. We first chose the two most popular cryptocurrency projects, including Bitcoin[4]and Ethereum[5] projects. Then we select 4 other projects on GitHub, based on the fact that these 4 projects have won a large number of "stars" from other developers. These "stars" means that these 4 projects are widely used by developers. These 4 projects include chia-blockchain[6], diem-libra-core[7], fabric[8], and solidity[9]. Table 1 describes the blockchain projects used in our research, including the name of the project, the code version used, the total number of lines in the project, the programming language used in the project and stars from GitHub. The bitcoin project is an

[4] https://github.com/bitcoin/bitcoin
[5] https://github.com/ethereum/go-ethereum
[6] https://github.com/Chia-Network/chia-blockchain
[7] https://github.com/diem/diem
[8] https://github.com/hyperledger/fabric
[9] https://github.com/ethereum/solidity

experimental digital currency. The Ethereum project is official Golang implementation of the Ethereum protocol. The diem project is a decentralized, programmable database. The solidity project is a statically typed, contract-oriented, and high-level language. The fabric project is a platform for distributed ledger solutions. The chia project is a modern cryptocurrency. For the Bitcoin, solidity, and chia projects, we utilize the sloccount[10] to calculate the total source lines of code, following previous study by Maldonado. For the remaining three projects, we developed a Python-based program to count the total number of lines of code.

Table 1. Summary of the Studied Projects. For each project, we present the used version, the total number of lines of code, the main programming languages, stars, and application area.

Project	Release	#Line of Code	Languages	Stars	Application Area
bitcoin	22	221466	C++,Python	58.9K	an experimental digital currency .
Ethereum	1.10.12	394246	go	33.3K	Official Golang implementation of the Ethereum protocol.
diem	1.0.2	227505	Rust	16.2K	a decentralized, programmable database.
solidity	0.8.10	224404	C++,Solidity, Python	13K	a statically typed, contract-oriented, high-level language.
fabric	2.3.3	1109729	go	12.8K	a platform for distributed ledger solutions.
chia	1.2.7	62232	Python	9.5K	a modern cryptocurrency .

Next we downloaded the source code of six software projects and used the code comment extraction tool to obtain the code comments for each project. After filtering unrelated source comments, we manually labelled all the instances. Three experienced programmers were invited to label the data and perform statistical analysis, and the final statistics proved that there was strong consistency of the three individuals.

4.2. *Baseline Method*

Due to the simplicity of MAT, proposed by Guo et al. [15], it can be as a baseline approach in SATD detection. The MAT approach is simple to describe, implement, and interpret. The MAT approach can offer a comparable performance to standard approaches. Other SATD detection approaches, including Pattern Match [12], NLP [13], TM [24], and CNN [22], may be hard to reproduce due to its complex structure and parameters. For blockchain software projects, as shown in Table 2, there are many task tags in files. The MAT is a simple-yet-effective baseline approach. There are two phrases in the MAT approach, "preprocessing" and "fuzz matching". After preprocessing the source comments, a list of tokenized and stemmed words are generated. Then, the preprocessed token list is matched with task tags. A comment is considered indicating SATD if and only if there is at least one task tag ("todo", "fixme", "xxx" or "hack") that occurs in the corresponding token list.

Table 2. The Detailed Number of Sampled Instances That Contain Each Task Tag on Each Project

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	2	132	18.64%
	WITHOUT_SATD	40780	72	0	0	18	58	0.22%
diem	SATD	446	212	1	8	6	227	50.90%
	WITHOUT_SATD	12411	112	2	9	10	133	1.07%
solidity	SATD	503	3	0	0	0	3	0.60%
	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%

[10] https://dwheeler.com/sloccount/

4.3. Evaluation Metrics

To investigate the impact of different approaches, we consider the following performance measures: precision, recall, and F1-measure. These performance evaluation measures have been used in previous studies for evaluating the results of experiments on the imbalanced datasets [12,13,22,24]. For a binary classification problem, the confusion matrix is shown in Table 3 for SATD detection.

Table 3. Confusion matrix for SATD detection

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

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

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

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

The precision is used to indicate the proportion of SATD that are correctly classified as SATD among those classified as SATD comments. The recall is used to indicate the proportion of SATD that are correctly classified as SATD among those true SATD comments. The F1-measure is a summary measure that combines both precision and recall.

4.4. Experimental Environment

The experimental environment is a workstation equipped with Nvidia RTX 2700 GPU, Intel(R) Core(TM) i7-10700K CPU and 64GB RAM, running Windows 10 Professional.

5. Experimental Results

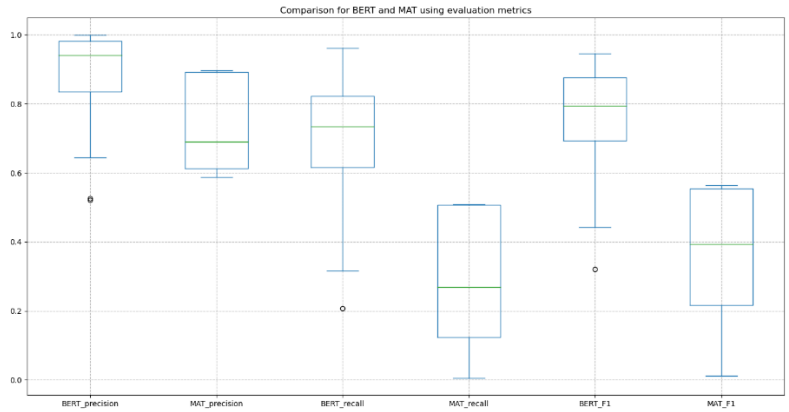

Figure 2. Comparison for BERT and MAT using evaluation metrics"

The comparison for BERT and MAT using evaluation metrics is shown in Figure 2. We show that our approach can get better classification performance using cross-project classification when a new software project is started. The code comments of 5 projects are selected as training data and validation data, and the remaining one project is used as test data. Therefore, there are 6 experiments for different classification models. 90% of the project comments of the 5 projects are randomly selected as training data, and the other 10% are used as validation data. For every experiment, precision, recall, and F1-measure are computed on test data. In addition, the Wilcoxon Signed-rank test is used to test whether the differences of precision, recall, and F1-measure in the 6 experiments are statistically significant at the *p-value* <0.05. Finally, Cohen's *d* effect size is used to quantify the amount of difference. For different effect sizes, it represents the diversity between related methods. The difference is considered Small (d=0.2), medium (d =0.5), or large (d =0.8).

From Figure 2, our proposed cost-sensitive pre-training model outperforms the baseline classification method. From the median calculation, the precision metric improved by 36.25%; the recall metric improved by 172.4%; and the F1 metric improved by 102% over the baseline method. The Wilcoxon Signed-rank test shows that the differences of F1-measure in the 6 projects are statistically significant at the *p-value* <0.05. Cohen's *d* effect size in F1-measure is 0.89, which shows that there is a bigger difference between the two models.

A comparison of the experimental results shows that our classification method far exceeds the baseline classification method. The method proposed by Guo et al. is based on JAVA open source projects. Developers developing these open source projects can get more convenient development aids due to development tools such as Eclipse. Programs can more easily introduce code comments with task tags. For the blockchain software projects, however, there are no similar development aids. Therefore, techniques based on deep understanding of natural language processing are more likely to achieve better classification performance. This also implies to us that we should choose a more suitable classification model for different types of software projects when detecting SATD.

6. Conclusion

Blockchain technology has been used in various fields, including digital currencies, distributed storage, and more. We provide an in-depth analysis of the code comments of open blockchain source software projects. A pre-trained model based on cost-sensitivity is proposed, which is compared with the baseline model on several evaluation metrics. The results were statistically analyzed. The results show that the pre-trained model based on natural language understanding is able to achieve better classification performance. Our method improves 102% over the baseline method in the F1 metric. This also foreshadows that we should consider using pre-trained models more when detecting SATD for blockchain software projects.

In the future we will apply our proposed method to other areas to validate the generalization capability of the method. In addition, we will also focus more on deep learning classification models for SATD detection to improve the performance of SATD classifiers in blockchain software systems.

Acknowledgements

This work was supported by Philosophy and Social Science Research Projects in Jiangsu (2020SJB0836), Nantong Science and Technology Project (JC2021124), Guangxi Key Laboratory of Trusted Software (kx202046, kx202013), Scientific Research Projects of Jiangsu College of Engineering and Technology (GYKY/2020/4), Research Project of Modern Educational Technology in Jiangsu Province (2021-R-94735), Special Project of China Higher Education Association(21SZYB23) . Sponsored by Special Foundation for Excellent Young Teachers and Principals Program of Jiangsu Province and Qing Lan Project of Jiangsu province.

References

1. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review, pp. 21260, 2008.
2. Huh, S., Cho, S., and Kim, S. Managing IoT Devices using Blockchain Platform. In 2017 19th international conference on advanced communication technology (ICACT), IEEE, pp. 464-467, 2017.
3. Delmolino, K., Arnett, M., Kosba, A., Miller, A., and Shi, E. Step by Step towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. In International conference on financial cryptography and data security, Springer, Berlin, Heidelberg, pp. 79-94, 2016.
4. Li, D., Wong, W.E., Zhao, M. and Hou, Q. Secure Storage and Access for Task-Scheduling Schemes on Consortium Blockchain and Interplanetary File System. In 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 153-159, IEEE, December 2020.
5. Korpela, K., Hallikas, J., and Dahlberg, T. Digital Supply Chain Transformation toward Blockchain Integration. In proceedings of the 50th Hawaii international conference on system sciences, 2017.
6. Wong, W.E., Li, X. and Laplante, P.A. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. Journal of Systems and Software, vol. 133, pp. 68-94, November 2017.
7. Wong, W.E., Debroy, V., Surampudi, A., Kim, H. and Siok, M.F. Recent catastrophic accidents: Investigating how software was responsible. In 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. pp. 14-22, IEEE, June 2010.
8. Liu, J., Huang, Q., Xia, X., Shihab, E., Lo, D., and Li, S. Is using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society, pp. 1-10, 2020.
9. Cunningham, W. The WyCash Portfolio Management System. ACM SIGPLAN OOPS Messenger, vol. 4, no. 2, pp. 29-30, 1992.
10. Potdar, A. and Shihab, E. An Exploratory Study on Self-admitted Technical Debt. In 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, pp. 91-100, 2014.

11. Bavota, G. and Russo, B. A Large-scale Empirical Study on Self-admitted Technical Debt. In Proceedings of the 13th international conference on mining software repositories, pp. 315-326, 2016.
12. Maldonado, E.D.S. and Shihab, E. Detecting and Quantifying Different Types of Self-admitted Technical Debt. In 2015 IEEE 7Th international workshop on managing technical debt (MTD), IEEE, pp. 9-15, 2015.
13. da Silva Maldonado, E., Shihab, E., and Tsantalis, N. Using Natural Language Processing to Automatically Detect Self-admitted Technical Debt. IEEE Transactions on Software Engineering, vol. 43, no. 11, pp. 1044-1062, 2017.
14. Wehaibi, S., Shihab, E., and Guerrouj, L. Examining the Impact of Self-admitted Technical Debt on Software Quality. In 2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER), IEEE, vol. 1, pp. 179-188, 2016.
15. Guo, Z., Liu, S., Liu, J., Li, Y., Chen, L., Lu, H., and Zhou, Y. How Far have we Progressed in Identifying Self-admitted Technical Debts? A Comprehensive Empirical Study. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 4, pp. 1-56, 2021
16. Bosu, A., Iqbal, A., Shahriyar, R., and Chakraborty, P. Understanding the Motivations, Challenges and Needs of Blockchain Software Developers: A Survey. Empirical Software Engineering, vol. 24, no. 4, pp. 2636-2673.
17. Jakobsson, M. and Juels, A. Proofs of Work and Bread Pudding Protocols. In Secure information networks, Springer, Boston, MA, pp. 258-272, 1999
18. Li, D., Wong, W.E. and Guo, J. A survey on blockchain for enterprise using hyperledger fabric and composer. In 2019 6th International Conference on Dependable Systems and Their Applications (DSA), pp. 71-80, IEEE, January 2020.
19. Narayanan, A., Bonneau, J., Felten, E., Miller, A., and Goldfeder, S. Bitcoin and cryptocurrency technologies: a comprehensive introduction. Princeton University Press, 2016.
20. Chuen, D.L.K. ed. Handbook of digital currency: Bitcoin, innovation, financial instruments, and big data. Academic Press, 2015.
21. Kruchten, P., Nord, R.L., Ozkaya, I., and Falessi, D. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes, vol. 38, no. 5, pp. 51-54, 2013.
22. Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X., and Grundy, J. Neural Network-based Detection of Self-admitted Technical debt: From Performance to Explainability. ACM transactions on software engineering and methodology (TOSEM), vol. 28, no. 3, pp. 1-45, 2019.
23. Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. JDeodorant: Identification and Removal of Type-checking Bad Smells. In 2008 12th European conference on software maintenance and reengineering, IEEE, pp. 329-331, 2008.
24. Huang, Q., Shihab, E., Xia, X., Lo, D., and Li, S. Identifying Self-admitted Technical Debt in Open Source Projects using Text Mining. Empirical Software Engineering, vol. 23, no. 1, pp. 418-451, 2018.
25. Chen, Y. Convolutional neural network for sentence classification, Master's thesis, University of Waterloo, 2015.
26. Devlin, J., Chang, M.W., Lee, K., and Toutanova, K. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018, arXiv preprint arXiv:1810.04805.

Yubin Qu received the B.S. and M.S. degrees in Computer Science and Technology from Henan Polytechnic University in China in 2004 and 2008. Since 2009, he has been a lecture with Information Engineering Institute, Jiangsu College of Engineering and Technology. He is the author of more than 10 articles. His research interests include software maintenance, software testing and machine learning.

W. Eric Wong received the M.S. and Ph.D. degrees in computer science from Purdue University. He is currently a Full Professor and the Founding Director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science, The University of Texas at Dallas (UTD). He also has an appointment as a Guest Researcher with the National Institute of Standards and Technology (NIST), Agency of the US Department of Commerce. Prior to joining UTD, he was with Telcordia Technologies (formerly Bellcore) as a Senior Research Scientist and the Project Manager, where he was in charge of dependable telecom software development. In 2014, he was named the IEEE Reliability Society Engineer of the Year. His research interest includes helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability. He has very strong experience developing real-life industry applications of his research results. He is the Editor-in-Chief of IEEE TRANSACTIONS ON RELIABILITY. He is also the Found-ing Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security (QRS).

Dongcheng Li received the BS degree in computer science from University of Illinois at Springfield and the MS degree in software engineering from the University of Texas at Dallas. He is currently working toward the PhD degree at the University of Texas at Dallas. His research focus is on search-based software engineering and intelligent optimization algorithms.