Multiclass Classification for Self-Admitted Technical Debt Based on XGBoost

Xin Chen, Dongjin Yu, Xulin Fan, Lin Wang, and Jie Chen

***Abstract*—In software development, due to the demands from users or the limitations of time and resources, developers tend to adopt suboptimal solutions to achieve quick software development. In such a way, the released software usually involves not-quite-right code that is called technical debt, which will significantly decrease the quality of software and increase the maintenance cost. Recently, the concept of self-admitted technical debt (SATD) is proposed and refers to technical debt that is self-admitted by developers in code comments. Existing studies mainly focus on detecting technical debt by classifying code comments into either "SATD" or "non-SATD." However, different types of SATD has different impacts on software maintenance and needs to be handled by different developers. Therefore, the detected SATD should be further classified so that developers can understand and remove technical debt better. In this article, we propose a new method based on eXtreme Gradient Boosting (XGBoost) to classify SATD into multiple classes. In our approach, we first preprocess the original code comments and adopt the easy data augmentation strategy to overcome the class unbalance problem. Then, chi-square is leveraged to select representative features from the textual feature set. Finally, we apply XGBoost to train a classifier and use the trained classifier to partition each comment into the corresponding class. We experimentally investigate the effectiveness of our approach on a public dataset, including 62 566 code comments from 10 open-source projects. Experimental results show that our approach achieves 56.66% in terms of macroaveraged precision, 59.07% in terms of macroaveraged recall, and 55.77% in terms of macroaveraged F-measure on average, and outperforms the natural language processing based method by 4.98%, 5.32%, and 3.17%, respectively. In addition, the experimental results also demonstrate that the data augmentation strategy is effective in improving the effectiveness of our approach.**

***Index Terms*—Class unbalanced problem, easy data augmentation (EDA), multiclass classification, technical debt, XGBoost.**

I. Introduction

IN SOFTWARE development, to meet conflicting goals, such as high quality, limited resources, and rapid release, developers have to adopt suboptimal solutions to complete rapid software development [1]. These solutions can help developers achieve short-term goals, but usually produce not-quite-right code that requires an increased cost to maintain in the long-term running of software [2]. Technical debt is a metaphor to reflect that suboptimal solutions are taken into consideration to achieve short-term goals in a software system [3]. For developers, technical debt has its own characteristics. On the one hand, technical debt is inevitable and ubiquitous [4]. It has negative impacts on the quality of software. Specifically, the impacts of technical debt seem to be small in the short-term, but it may accumulate to form great threats to software with the continuous update and evolution and is hard to be quantified. On the other hand, technical debt is unpredictable and invisible [4]. Developers need to spend a lot of time and resources to identify and remove technical debt, thus greatly increasing the maintenance cost. Technical debt is always a research focus of industry and academia. Many technical debt detection and management platforms have sprung up, such as SonarQude,[1] NDepend,[2] and Teamscale.[3]

To help developers reduce the software maintenance cost, automatically detecting technical debt becomes one of the most important tasks in software maintenance. In the literature, researchers have proposed many methods to detect technical debt. Some studies focus on leveraging source code to identify technical debt [5]–[7]. They adopt analysis tools to measure different indicators related to source code and judge whether the code is technical debt based on these indicators [5]–[7]. However, the methods based on source code usually lead to high false positive rate [1]. Another body of studies try to utilize code comments to detect technical debt, which is called self-admitted technical debt (SATD) [4], [8], [9]. A generic paradigm is to extract features from code comments and encode these extracted features to train a classifier for predicting the label of each new code comment [4], [8], i.e., whether it is SATD or not. Compared with methods based on source code, methods based on code comments are more reliable and lightweight since technical debt is self-admitted by developers in code comments [1].

Currently, identifying technical debt by code comments is a research focus in technical debt detection. The aforementioned studies have achieved promising results in SATD detection, but they mainly focus on resolving the binary classification problem (i.e., classifying code comments into "SATD" or "non-SATD") and do not consider the impacts of different types of SATD on

Manuscript received 26 August 2020; revised 2 February 2021 and 13 May 2021; accepted 6 June 2021. Date of publication 29 June 2021; date of current version 1 September 2022. This work was supported in part by the Natural Science Foundation of Zhejiang Province under Grant LY21F020020, in part by the National Natural Science Foundation of China under Grant 61902096 and Grant 61702144, and in part by Key Project of Science and Technology of Zhejiang Province under Grant 2017C01010. Associate Editor: D. Lo. *(Corresponding author: Dongjin Yu.)*

The authors are with the School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China (e-mail: chenxin4391@hdu.edu.cn; yudj@hdu.edu.cn; fanxulin@hdu.edu.cn; 171050046@hdu.edu.cn; cjie@hdu.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2021.3087864.

Digital Object Identifier 10.1109/TR.2021.3087864

[1][Online]. Available: https://www.sonarqube.org/

[2][Online]. Available: https://www.ndepend.com/

[3][Online]. Available: https://www.cqse.eu/en/products/teamscale/landing/

maintenance efficiency. Actually, identifying different types of technical debt is an important task, which can help developers understand technical debt better. In the literature, Maldonado *et al.* manually classified SATD into different types, such as design debt, requirement debt, defect debt, documentation debt, and test debt [1]. They analyzed that different types of SATD may result in different unexpected behaviors and need to be handled by different developers [1]. In another empirical study [9], Maldonado and Shihab pointed out that identifying different technical debt can help developers understand technical debt better and is an complement to existing studies related to technical debt detection.

In this study, to help developers understand technical debt better, we attempt to resolve the problem of multiclass SATD classification. We mainly focus on identifying three types of technical debt (namely defect debt, design debt, and implementation debt) from raw code comments. We propose a new method based on chi-square (CHI) and eXtreme Gradient Boosting (XGBoost) [10]. In our approach, we first filter out the identical code comments by simple string matching or the cosine similarity method. Then, we adopt a data augmentation strategy to balance the number of comments between small classes and large classes to break through the class unbalanced problem. Next, CHI is applied to select representative features from the feature set, and the CountVectorizer tool [11] is leveraged to represent each comment. Finally, we build a classifier based on XGBoost and use the trained classifier to classify new code comments into the corresponding classes.

To validate the effectiveness of the proposed approach, we conduct extensive experiments on a dataset, including 62 566 code comments from 10 open-source projects provided by Maldonado *et al.* [1]. We investigate four research questions (RQs) and employ widely used macroaveraged precision (MacroP), macroaveraged recall (MacroR), and macroaveraged F-measure (MacroF) to evaluate the performance of our approach in classifying SATD comments. We select four state-of-the-art methods, namely the natural language processing (NLP) based method [1], the naive Bayes multinomial (NBM) based method [4], the convolutional neural network (CNN) based method [8], and the bidirectional long short-term memory (BiLSTM) based method [12], as baselines for comparison. The experimental results show that our approach achieves 56.66% in terms of MacroP, 59.07% in terms of MacroR, and 55.77% in terms of MacroF on average, and outperforms the state-of-the-art NLP-based method by 4.98%, 5.32%, and 3.17%, respectively. Meanwhile, the results also demonstrate that the data augmentation strategy is effective to address the class unbalance problem. In addition, experimental results also reveal that different types of technical debt usually contains some specific features. With these features, the SATD comments may be easily identified and their types can be determined.

In this article, we make the following contributions.

1) To the best of our knowledge, this is the first work aiming at identifying three different types of SATD, including defect debt, design debt, and implementation debt, to help developers understand technical debt better.
2) We propose a new method by leveraging the CHI strategy to select representative features from the feature set and XGBoost to build the classifier for identifying these three types of SATD, respectively.
3) We perform extensive experiments to evaluate the performance of the proposed approach and compare our approach with four baseline methods on a public dataset. The results show that our approach outperforms these baseline methods.

The rest of this article is structured as follows. In Section II, we present the background of technical debt and the motivation for conducting this work. The approach for detecting these three types of SATD is detailed in Section III. Sections IV and V present the experimental setup and the experimental results, respectively. In Section VI, discussion is given. The threats to validity are discussed in Section VI I, and the related work is reviewed in Section VIII. Finally, Section IX concludes this article.

II. Background and Motivation

In this section, we detail the background of technical debt as the motivation for resolving the problem of multiclass SATD classification.

The concept of technical debt is first proposed by W. Cunningham in 1992 [2] and expanded by other researchers [3], [4]. Technical debt is not always bad, but most of the time has a negative impact on the quality of software [13]. Therefore, many studies have been conducted to help developers detect technical debt [2], [7], [8]. Recently, the concept of SATD is proposed and refers to that technical debt that is intentionally introduced by developers [4]. Many effective methods have been developed to distinguish "SATD" from "non-SATD." However, few studies take different types of SATD into consideration. In the literature, Alves *et al.* conducted an empirical study to investigate SATD and summarized an ontology about possible types of SATD, including architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test, and test automation debt [14]. Based on this, Marinescu also tried to quantify different of SATD by an empirical investigation [13]. They emphasized that different types of technical debt have different impacts on software and design debt has the highest impact.

Identifying different types of technical debt is an important task. It can help us understand technical debt better from the point of view of developers [9]. Meanwhile, many studies focus on distinguishing SATD comments and non-SATD comments. Comparatively, only a few studies attempt to detect different types of SATD comments. Therefore, identifying different types of technical debt is an important complement to existing studies. In this study, we mainly focus on identifying three types of SATD, namely defect, design, and implementation debt based on the assumption that code comments are documented well. However, there exist challenges in identifying these three types of technical debt. First, by an investigation on code comments, we find that the number of SATD comments is less than that of

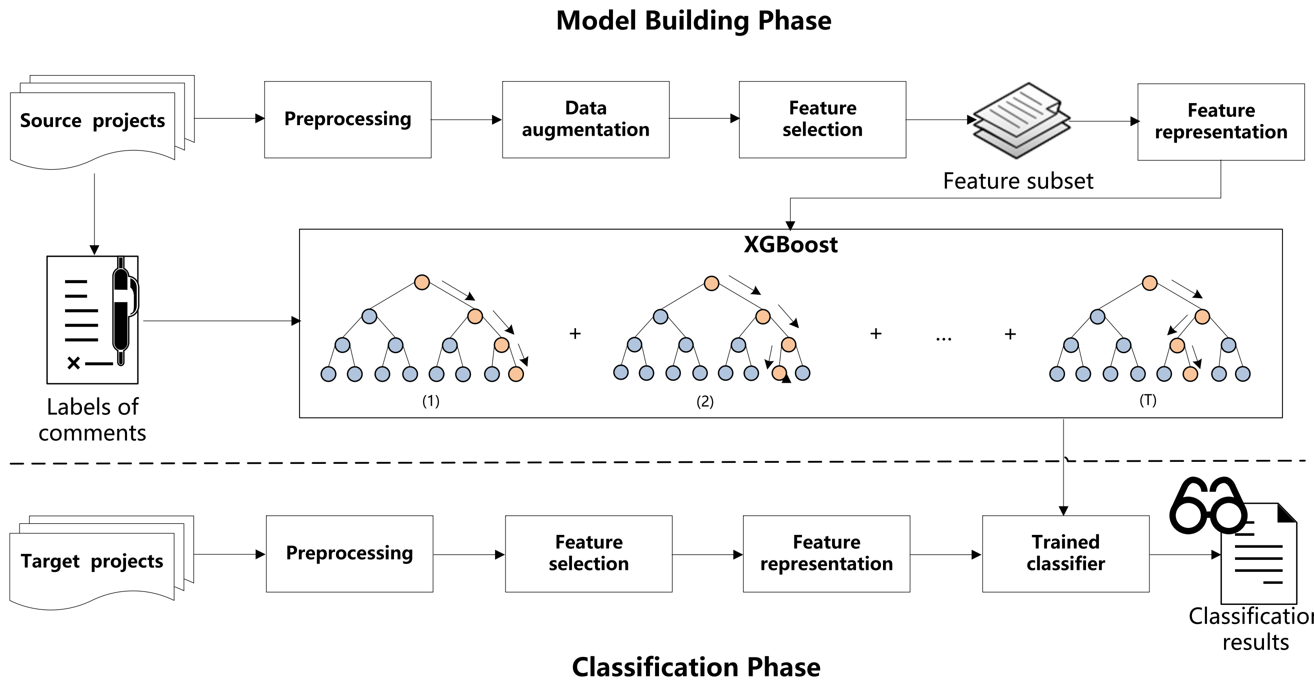

Fig. 1. Overall framework of our approach.

non-SATD comments in a project. Meanwhile, the number of different types of SATD comments varies sharply in the same project. For example, the number of design debt comments is about 20 times that of defect debt comments and about 15 times that of implementation debt comments in the JFreeChart project (see Section IV-B). Second, different types of technical debt may contain some same features. It is hard to learn useful semantic information for different types of technical debt comments.

To overcome these challenges, we propose a new method based on CHI and XGBoost. CHI is effective feature selection algorithm that is often used to detect the independence of two events in statistics. Compared with mutual information (MI) [15], CHI is more powerful since it can take both the occurrence and absence of a feature into consideration [16]. XGBoost is a scalable machine learning system for tree boosting [10]. It has been widely applied to a variety of machine learning and data mining challenges and presents good performance. For example, in the machine learning competition Kaggle 2015, 17 out of 29 winning solutions adopted XGBoost to train models [10]. Also, XGBoost was employed by every winning team in the top-10 in KDDCup 2015 [10]. In addition, we apply the easy data augmentation (EDA) strategy to add the number of samples belonging to small classes to overcome the class unbalanced problem.

III. Methodology

In this section, we describe the overall framework of our approach, as shown in Fig. 1. Our approach takes in the word vectors of code comments as the input and then outputs the corresponding class label of a comment, namely design debt, defect debt, and implementation debt. More specifically, we first filter out the identical code comments and preprocess the retained comments. Then, we adopt data augmentation to increase the number of code comments in small classes. After that, the feature selection strategy CHI is applied to select representative features with the highest CHI scores. Finally, we adopt XGBoost to train a classifier and leverage the trained classifier to predict the labels of new code comments.

A. *Preprocessing*

Generally, developers are used to write code comments freely in their own styles and have their own preferences to terms and phrases, thus code comments are presented in distinct free-text forms and their quality varies sharply. However, classification models can only process standardized and structure text. Therefore, we need to preprocess the original code comments and filter out the noise data. The preprocessing procedure is composed of data filtering, change record deletion, tokenization, stop word removal, and lemmatization.

1) Data Filtering: By investigating code comments, we observe that there are some identical ones, i.e., they contains the same textual contents and the same semantic structure. Actually, these identical code comments do not increase the diversity of the sample space (a comment is a sample in this study) and have no positive effect on identifying SATD comments. Instead, they will spend additional computing cost. Therefore, we need to filter out these identical comments according to the simple string matching method or the cosine similarity method [17].

2) Change Record Deletion: Due to the continuous update and evolution of software, developers may record the revision change history in comments. In practice, these comments are not related to SATD and would increase the number of features. They are unhelpful for detecting SATD. Therefore, we should delete historical revision records that are usually presented in the form of "xx-xx-xx: text," where "xx-xx-xx" represents a date and "text" describes the historical revision change information. To distinguish historical revision records, we search each comment whether it contains a date presented in the form of "xx-xx-xx." If the comment contains a date, we directly delete the date and the following textual content.

3) Tokenization: The process of tokenization is to divide the continuous text string into a set of words, phrases, punctuation, or numbers. In this study, we only keep the tokens that are composed of English letters. In addition, we convert all words into lowercase to eliminate differences.

4) Stop Word Removal: In identifying SATD comments, some frequently occurred words are considered as stop words that carry a litter semantic information and have a negative impact on the prediction model. Huang *et al.* [4] considered some stop words to be helpful for detecting SATD comments, but most of the time, they do not contain useful information. Therefore, in this study, we remove all the stop words according to the common stop word list.[4]

5) Lemmatization: Code comments may contain different forms of words, such as "use," "using," and "used." These words convey the same semantic and should be reduced to the root form "us." Therefore, we implement the lemmatization operation that aims to reduce inflectional forms and sometimes derivatively related forms of a word to a common base form. In this study, we leverage the widely used Porter stemmer[5] to reduce a word to its representative root form.

B. Data Augmentation

By an investigation on the dataset, we obverse that the number of different types of SATD comments is extremely unbalanced. For example, the ten projects (the details about the ten projects are presented in Section IV-B) includes 58 495 non-SATD comments, which is about 14.37 times the number of SATD comments. In addition, the number of design debt comments is about 20 times that of defect debt comments and about 15 times that of implementation debt comments on the JFreeChart project. The class unbalance problem may seriously impact the classification results of SATD comments. Moreover, learning semantic knowledge from a small number of labeled code comments is difficult for machine learning techniques to predict the correct labels of code comments. Therefore, we need to address the class unbalanced problem and increase the number of code comments in small classes.

The generally used methods for the unbalanced problem are undersampling [18] and oversampling [19]. Undersampling balances the number of samples between large classes and small classes by selecting some samples from large classes, but it may miss some important information and decrease the diversity of sample space. Meanwhile, the method will reduce the number of samples for training classifier. In contrast, oversampling achieves the data balancing by repeatedly selecting samples from small classes, but it does not increase the diversity of sample space and easily leads to overfitting. Another important method for addressing the unbalanced problem is data augmentation by generating new samples based on existing samples. Comparatively, data augmentation increases not only the number of samples in a small class, but also the diversity of sample space.

In this study, we attempt to adopt EDA [20], which is an effective data augmentation technique. Typically, EDA contains four basic operations, namely synonym replacement, random insertion, random swap, and random deletion. Synonym replacement and random insertion will introduce new features that may influence the prediction results of SATD comments. Meanwhile, random deletion may lose some features that have important roles on identifying SATD comments. Therefore, we only adopt the random swap operation.

Random swap: This operator randomly selects two samples belonging to the same small class and breaks down each selected sample into two fragments. Then, the fragments of the two samples are swapped to form two new samples.

C. Feature Selection

In this study, each comment can be viewed as an independent sample and each word is a feature. There are a large number of features in each project, thus each comment will be represented by a high-dimensional vector. However, an overly high number of dimensions may incur the curse-of-dimensionality problem [4]. Feature selection is an important step to reduce the dimensions of feature space and improve the performance of classification models by removing irrelevant and redundant features. Traditionally, there are many commonly used feature selection methods, e.g., term frequency, document frequency, MI, and CHI. They have been applied to resolve various software engineering problems [4], [16]. In this study, we adopt CHI statistics to select the most relevant features. The main reason is that it takes into account both the occurrence and absence of a term [16]. CHI mainly measures the independency between the class G_i and the feature w_j. The formula for computing the score of each feature corresponding to G_i is defined as follows:

$$\mathrm{CHI}(w_j, G_i) = \frac{N(AD - CB)^2}{(A + C)(B + D)(A + B)(C + D)} \tag{1}$$

where N is the total number of code comments in the whole dataset, A is the number of code comments that not only contain the word w_j but also belong to the class G_i, B is the number of code comments that contain the word w_j but do not belong the class G_i, C is the number of code comments that belong to the class G_i but do not contain the word w_j, and D is the number of code comments that neither contain the word w_j nor belong the class G_i.

Then, we calculate the probability $P(G_i)$ of comments (belonging to G_i) occurring in the dataset. For the feature w_j, the eventual CHI score is determined according to the following

[4][Online]. Available: https://code.google.com/archive/p/stop-words/
[5][Online]. Available: http://tartarus.org/martin/PorterStemmer/

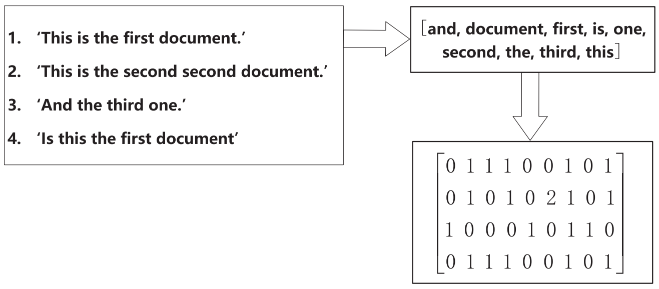
1. 'This is the first document.'
2. 'This is the second second document.'
3. 'And the third one.'
4. 'Is this the first document'
[and, document, first, is, one, second, the, third, this]
0 1 1 1 0 0 1 0 1
0 1 0 1 0 2 1 0 1
1 0 0 0 1 0 1 1 0
0 1 1 1 0 0 1 0 1

Fig. 2. Example for generating the vectors for documents by CountVectorizer.

formula:

$$\mathrm{CHI}(w_j) = \sum_{i=1}^{k} P(G_i)\mathrm{CHI}(w_j, G_i). \tag{2}$$

Finally, we rank these features according to their CHI scores in descending order. We select the top $k(k = 10\%)$ of features with the highest scores and remove other features. In such a way, the dimensions of feature space are greatly reduced.

D. Feature Representation

After feature selection, the main task is to transform each comment into a vector. In this study, we employ vector space model [21] to transform each comment into a vector of which each dimension corresponds to a single feature and its value is the weight of the feature. Meanwhile, we adopt CountVectorizer,[6] an effective feature representation tool provided by the scikit-learn library, to determine the value of each dimension. For a given document (namely a comment in this study), CountVectorizer transforms it into a vector based on the frequency of each word occurring in the text. Given a corpus consisting of a set of documents, CountVectorizer extracts each single word from all documents and ranks all words according to the alphabetical order. Then, for each document, an n-dimensional vector is generated according to its textual content, where n is the number of different words and the value of each dimension is the frequency of the corresponding word occurring in this document. If the document contains no word, the corresponding value is 0. Fig. 2 shows an example about how CountVectorizer works. Notably, we do not remove the stop words within the documents since there are only four different words that are not stop words.

E. XGBoost

In this study, we adopt XGBoost proposed by Chen and Guestrin [10] in 2016 to train a classifier for identifying defect debt, design debt, and implementation debt. XGBoost is an ensemble machine learning system combining a series of decision trees of which each learns from the prior one and influences the next one. It improves the traditional gradient boosting decision tree (GBDT) [22] algorithm with respect to computing speed, generalization performance, and scalability.

[6][Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Compared with gradient boosting, XGBoost introduces regularization to the loss function to establish the objective function

$$J(\theta) = L(\theta) + \Omega(\theta) \tag{3}$$

where

$$L(\theta) = l(\hat{y}_j, y_j) \tag{4}$$

$$\Omega(\theta) = \gamma T + \frac{1}{2}\lambda||\omega||^2. \tag{5}$$

As shown in (3), the objective function is composed of two parts $L(\theta)$ and $\Omega(\theta)$, where θ represents various parameters learned by the given data. $L(\theta)$ is a differentiable convex loss function that measures the difference between the prediction result $\hat{y}_j$ and the target result y_j. There are two kinds of widely used loss functions, namely the mean square loss function $l(\hat{y}_j, y_j) = (\hat{y}_j - y_j)^2$ and the logistic loss function $l(\hat{y}_j, y_j) = y_j ln(1 + e^{-\hat{y}_j}) + (1 - y_j)ln(1 + e^{\hat{y}_j})$. $\Omega(\theta)$ is a regularized term (such as $L1$ norm or $L2$ norm), which is used to penalize the complexity of the model (i.e., the regression tree functions). Among them, T represents the number of leaves in the tree, γ is the learning rate and its value is between 0 and 1. γT is adopted to prevent overfitting. λ is a regularized parameter, $\boldsymbol{\omega}$ is the leaf scores, and ω_i is the score of the ith leaf. Compared against the traditional GBDT algorithm, XGBoost adds $\frac{1}{2}\lambda||\boldsymbol{\omega}||^2$ that can further avoid the overfitting and strengthen the generalization ability of the model.

Given a dataset with n samples and m features $D = \{(\boldsymbol{x}_j, y_j)\}$, where $\boldsymbol{x}_j (j = 1, 2, \ldots, n)$ represents a sample and y_j is the corresponding label, the output $\hat{y}_j$ of the model is calculated by K additive trees

$$\hat{y}_j = \phi(\boldsymbol{x}_j) = \sum_{k=1}^{K} f_k(\boldsymbol{x}_j), f_k \in F \tag{6}$$

where $F = \{f(\boldsymbol{x}) = \omega_{q(\boldsymbol{x})}\}(q : \mathbb{R}^m \to T, \omega \in \mathbb{R}^T)$ is the space of regression trees, q indicates a tree structure that can map a sample $\boldsymbol{x}_i$ to the corresponding leaf, and f_k corresponds to an independent tree structure q and leaf weights $\boldsymbol{\omega}$.

In addition, considering that (3) uses functions as parameters and cannot be optimized by traditional methods in Euclidean space, XGBoost accumulates regression trees and appends a new optimized object in each iteration. Therefore, at the tth iteration, the objective function is defined by

$$J^{(t)} = \sum_{j}^{N} l(y_j, \hat{y_j}^{(t-1)} + f_t(\boldsymbol{x}_j)) + \Omega(f_t). \tag{7}$$

F. Training Classifier

Classifier training follows a supervised learning paradigm. Therefore, we need a large number of samples containing defect debt comments, design debt comments, and implementation debt comments. In this study, we collect a public dataset involving ten projects, the detailed information is shown in Section IV-B. We employ the leave-one-out cross-validation [23] to train the classifier. The reason is because leave-one-out cross-validation is more reliable and reproducible [24] and it can simulate the real scenario that different projects may have feature differences. Based on the ideal of leave-one-out cross-validation, for each project, we regard the samples from the other nine projects as the training set and the samples from this project as the testing set.

In each training procedure, we leverage the EDA strategy to generate new comments to increase the number of samples in small classes. We first determine the training set and the testing set. Then, for each small class (defect debt or implementation debt) in the training set, we generate n $(n = 1000)$ new samples. More specially, in each run, we first produce m $(m = 100)$ samples by performing the random swap operation 50 times since this operation will generate two new samples each time. Then, we adopt the interclass distance to evaluate the generated samples and select the one with the largest interclass distance. The interclass distance can be calculated by the following formula:

$$d = \frac{1}{\sum_{i=1}^{c} n_i} \sum_{i=1}^{c} \sum_{j=1}^{n_i} |\boldsymbol{y} - \boldsymbol{x}_{ij}| \tag{8}$$

where c is the number of classes and n_i is the number of the samples in the ith class, $\boldsymbol{y}$ denotes a generated sample and $\boldsymbol{x}_{ij}$ represents the jth sample in the ith class, and d represents the average distance between the generated sample and all the samples in all the classes.

We repeatedly perform this procedure n times. By this way, we can enrich the samples of small classes, thus alleviating the impacts of class imbalance problem. Notably, we do not perform the data augmentation strategy for the testing set. The generated samples are put into the training set and fed to the XGBoost to train the classifier.

IV. Experimental Setup

In this section, we detail the experimental setup. First, we present the RQs. Then, we describe the details of data collection. Next, the evaluation metrics and baseline methods are introduced. Finally, the experimental platform and parameter settings are shown.

A. Research Questions

In this article, we mainly investigate the following four RQs to validate the effectiveness of our method from different aspects in identifying these three different types of technical debt, namely defect debt, design debt, and implementation debt.

1) *RQ1*: Can our approach outperform the baseline methods?
2) *RQ2*: How effective is the feature selection on our approach?
3) *RQ3*: Can the data augmentation improve the performance of our approach?
4) *RQ4*: What are the important features for each type of SATD?

$RQ1$ is designed to evaluate the effectiveness of our approach in detecting different types of SATD comments by comparison with baseline methods, which could be helpful to developers in choosing suitable techniques. $RQ2$ is designed to investigate whether feature selection is effective for SATD detection. $RQ3$ is designed to investigate whether the data augmentation improves the performance of our approach. $RQ4$ is designed to investigate what are the most influential words (which have the highest weights) in training the classifier.

B. Dataset

In order to evaluate the effectiveness of the proposed approach, we run experiments on a public available dataset provided by Maldonado *et al.* [1]. The dataset includes 1 462 058 source lines of code (SLOC) and 62 566 comments from 10 open-source projects, namely Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and Squirrel. In the procedure of data extraction, Maldonado *et al.* adopted the open-source Eclipse plug-in JDeodorant [25] to parse the source code and extract the code comments. Then, five heuristics rules were designed and applied to filter out comments that are irrelevant to SATD. Then, they developed a Java-based tool to recommend a series of possible types for each comment and manually determined the final type. Since the code comments do not contain all types of debt mentioned by Alves *et al.* [14], they mainly annotated five types of technical debt, including design debt, defect debt, documentation debt, implementation debt, and test debt. Given the impacts of the personal experience and subjectivity, Cohen's kappa coefficient is adopted to calculate the consistency between reviewers of the stratified samples. The result is +0.81, which indicates high consistency between the reviewers.

We download the dataset from the website.[7] With the continuous maintenance to this dataset, the label of each comment has been determined and corrected. The dataset consists of five different types of technical debt, namely design debt, defect debt, implementation debt, test debt, and documentation debt. Table I presents the detailed information of these ten projects. The columns show the names of projects, followed by the version information of projects, the number of SLOC, the number of contributors, the number of comments made by developers, and the number of comments belonging to different types of SATD in each project. As shown in the table, only 6.50% of code comments are SATD comments and the number of non-SATD is about 14.38 times of that of SATD comments. In addition, 11.84%, 67.81%, and 18.99% of technical debt are defect debt, design debt, and implementation debt, only 2.13% and 1.4% of technical debt are test debt and documentation debt. The number of comments belonging to different types of SATD is extremely unbalanced compared with that of non-SATD comments. Specifically, Hibernate does not include test debt and Ant, EMF, JEdit, and JFreeChart do not involve documentation debt. Since there are not sufficient data for building the classifier to identify test debt and documentation debt, we only identify three types of technical debt in this study, including defect debt, design debt, and implementation debt. Given that test debt and documentation debt comments may influence the classification results of these three types of technical debt, we remove relevant

[7][Online]. Available: https://github.com/tkdsheep/TechnicalDebt

TABLE I
DETAILS OF THE TEN PROJECTS USED IN EXPERIMENTS

Project	**Version**	**#SLOC**	**#Contributors**	**#Comments**	**# Defect**	**#Design**	**#Implementation**	**#Test**	**#Documentation**
Ant	1.70	115,881	74	4,137	13	95	13	10	0
JMeter	2.10	81,307	33	8,162	22	316	21	12	3
ArgoUML	0.34	176,830	87	9,548	127	801	411	44	30
Columba	1.4	100,200	9	6,478	13	126	43	6	16
EMF	2.41	228,191	30	4,401	8	78	16	2	0
Hibernate	3.3.2	173,467	8	2,968	52	355	64	0	1
JEdit	4.2	88,583	57	10,322	43	196	14	3	0
JFreeChart	1.0.19	132,296	19	4,423	9	184	15	1	0
JRuby	1.40	150,060	328	4,897	161	343	110	6	2
SQuirrel	3.3.0	215,234	46	7,230	24	209	50	1	2
Total	-	1,462,058	909	62,566	472	2,703	757	85	54

comments from the dataset straightforwardly. Therefore, the number of the retained comments of these ten projects should be 4127, 8147, 9474, 6456, 4382, 2967, 10 305, 4407, 4889, and 7227, respectively.

C. Evaluation Metrics

In this study, we aim to identify different types of technical debt (defect debt, design debt, and implementation debt), which can be regarded as a multiclass classification problem. MacroP, MacroR, and MacroF are the widely used metrics to evaluate the effectiveness of automated multiclass classification techniques [24], [26]. Therefore, we can adopt MacroP, MacroR, and MacroF to validate the effectiveness of our method. Notably, although non-SATD comments can also be identified in this study, we do not care the results of non-SATD comments. We first calculate the local results of each type of technical debt, then the global results can be obtained by averaging the results of these three types of technical debt.

Let c be the number of classes, TP_i represent the number of correctly predicted code comments belonging to class G_i, FP_i be the number of incorrectly predicted code comments belonging to class G_i, TN_i denote the number of correctly predicted code comments not belonging to class G_i, and FN_i represent the number of incorrectly predicted code comments not belonging to class G_i.

MacroP evaluates the degree of correctness between the predicted results of all classes and the ground truth [24], [26]. The formula for calculating MacroP is as follows:

$$\text{MacroP} = \frac{1}{c}\sum_{i=1}^{c}\frac{TP_i}{TP_i + FP_i}. \tag{9}$$

MacroR measures the level of consistency between the predicted results of all classes and the ground truth [24], [26]. The formula for calculating MacroR is as follows:

$$\text{MacroR} = \frac{1}{c}\sum_{i=1}^{c}\frac{TP_i}{TP_i + FN_i}. \tag{10}$$

MacroF is the tradeoff between MacroP and MacroR [24], [26]. The formula for calculating MacroF is as follows:

$$\text{MacroF} = \frac{2 * \text{MacroP} * \text{MacroR}}{\text{MacroP} + \text{MacroR}}. \tag{11}$$

D. Baseline Methods

To effectively evaluate the performance of our method, we select four state-of-the-art methods as baseline methods for comparison.

1) Baseline 1 (NLP-Based Method): Maldonado *et al.* [1] developed an effective method to automatically detect the most common types of SATD, namely design debt and requirement debt. In the method, they established a maximum entropy classifier to predict the type of the comment related to technical debt. Experimental results demonstrated that the NLP-based method can effectively identify these two types of SATD. Although they used the same dataset with our method, we do not directly use their results since they identified only two types of SATD and reported the precision, recall, and F1-score of each type, respectively. Therefore, we fully follow their method and the same settings to build the classifier in this study.

2) Baseline 2 (NBM-Based Method): Huang *et al.* [4] proposed an automated approach for identifying SATD comments. They first preprocessed code comments and adopted information gain [27] for feature selection. Then, they built a subclassifier for each project by using NBM [28] and integrated these subclassifiers to form a composite classifier. Finally, they leveraged a voting mechanism to predict the labels of comments from a new targeted project. Experimental results shown that the NBM-based method can effectively improve the results compared with the baseline methods.

3) Baseline 3 (CNN-Based Method): Ren *et al.* [8] leveraged CNN [8] to build a classifier for detecting SATD. They first identified five key challenges that affect the performance, generalizability, and adaptability of the pattern-based method. Then, CNN is leveraged to learn the informative text features of comments. To understand the learned text features, they used a backtracking method to highlight the prominent key phrases. In the evaluation, they also revealed many less obvious and less frequent commenting patterns for identifying SATD.

4) Baseline 4 (BiLSTM-Based Method): In our previous work [12], we attempted to leverage BiLSTM [29] to distinguish SATD from non-SATD. To improve the learning ability of BiLSTM, the attention mechanism was introduced. Meanwhile, a balanced cross entropy loss function is combined to overcome the class unbalance problem. In addition, generic algorithms were adopted to optimize the balancing factor of the loss function. Experimental results revealed that the BiLSTM-based method outperforms the baseline methods.

TABLE II
PARAMETERS SETTINGS IN OUR EXPERIMENT

Algorithm	**Parameters**	**Values**
Ours	max_depth	6
	$n\_estimators$	1000
	$colsample\_bytree$	0.8
	$subsample$	0.8
	$learning\_rate$	0.06
	$early\_stopping\_rounds$	30
CNN	*dimensional of word vectors*	300
	number of filters	128
	batch size	16
	l2	0.01
	learning rate	0.0001
	dropout	0.5
BiLSTM	*dimensional of word vectors*	300
	number of hidden units	64
	batch size	64
	l2	0.01
	learning rate	0.0001
	dropout	0.5

E. Experimental Platform and Parameter Settings

In this article, all the experiments are conducted with Python 3.5, compiled with Pycharm 2017.3.1 and run on a PC with 64-b Ubuntu 16.04, an Intel Core(TM) i9-7900X CPU, and a GeForce GTX 1080Ti GPU.

In XGBoost, there are many hyperparameters that play important roles in achieving good performance for the classification model. In our experiment, we leverage a grid search [30] to obtain the optimal or approximate optimal hyperparameter values for XGBoost, as shown in Table II. In the table, max_depth represents the depth of trees, $n\_estimators$ is the number of maximum iterations, $colsample\_bytree$ and $subsample$ are the percentage of the used features in total features and the percentage of the used data in the training set when training each tree, respectively, $learning\_rate$ is the learning rate, and $early\_stopping\_rounds$ indicates that the model will stop when the results are not improved in continuous $early\_stopping\_rounds$ iterations. Similarly, we also adopt the grid search [30] to determine the suitable values of hyperparameters for CNN and BiLSTM, as shown in Table II.

Considering that the tuning of hyperparameters is not the focus, we do not present the detailed tuning results in Section V. In contrast, the percentage of selected features is an important parameter for identifying these three types of technical debt, we will show the tuning details in Section V-B. The default value is set to 10% in this article. Notably, we first generate the augmented samples for each small class and apply them to our method and all the baseline methods.

V. Experiment Results

In this section, we aim to evaluate the performance of our approach and answer the aforementioned RQs in Section IV-A.

A. Investigation to RQ1

1) Motivation: To the best of our knowledge, no study is conducted to identify these three types of SATD, namely defect debt, design debt, and implementation debt. Therefore, we select some state-of-the-art methods that are used to detect SATD or identify other types of technical debt as baselines to evaluate the effectiveness of our approach. The NLP-based method [1] is proposed to detect both design debt and requirement debt. It may be also effective in identifying defect debt or implementation debt. In addition, the NBM-based method, the CNN-based method, and the BiLSTM-based method are proven to have good performance in detecting SATD comments. They may be suitable for identifying different types of technical debt.

2) Approach: In our experiment, we fully reproduce these baseline methods based on the descriptions from the related studies. Considering that these methods except for the NLP-based method are not designed for multiclass SATD classification, we need to modify them to be suitable for identifying different types of SATD. For the CNN-based method and the BiLSTM-based method, we can directly output three different labels to represent different types of SATD. For the NMB-based method, we first select design debt comments as the positive samples (i.e., the label is 1) and other types of technical debt comments as the negative samples (i.e., the label is 0). Then, we train subclassifiers to predict the labels of comments from the targeted projects. We repeat this process to obtain the results of each type of technical debt.

In addition, since these methods implement different preprocessing operations and adopt different strategies for feature selection. In our experiment, we fully reuse their operations. For feature selection, we set the number of selected features according to the default values adopted in their studies. Moreover, we add the data augmentation strategy in all the baseline methods and employ the leave-one-out cross-validation to train the classifier. For the CNN-based method, the BiLSTM-based method, and the NLP-based method, we directly put the 1000 newly generated samples for each small class into the training set. For the NBM-based method, we randomly divide the 1000 newly generated samples for each small class into nine equivalent folds since the NBM-based method needs to build nine subclassifiers. Then, we combine the samples from each project and a fold of newly generated samples to train a subclassifier for each project.

Considering that MacroF is the tradeoff between MacroP and MacroR, we employ MacroF as the main evaluation metric in all experiments.

3) Results: Tables III –V show the experimental results of our method and the baseline methods in terms of MacroF, MacroP, and MacroR, respectively. In the tables, the best results on these ten projects are highlighted in bold.

As shown in Table III, our approach achieves higher results than other methods in terms of MacroF on all the ten projects but Ant and Columba. For example, our method obtains 67.01% in terms of MacroF on EMF and outperforms the NBM-based method, the CNN-based method, the BiLSTM-based method, and the NLP-based method by 24.3%, 26.23%, 28.97%, and

TABLE III
RESULTS OF ALL THE METHODS ON THESE TEN PROJECTS IN TERMS OF MACROF

Project	Ours	NBM	CNN	BiLSTM	NLP
Ant	52.32%	42.74%	38.97%	53.99%	**57.44%**
JMeter	**51.70%**	40.69%	45.26%	37.65%	40.32%
ArgoUML	**50.54%**	45.92%	45.64%	42.14%	46.58%
Columba	51.18%	**60.07%**	52.08%	42.47%	48.78%
EMF	**67.01%**	42.71%	40.78%	38.04%	56.29%
Hibernate	**58.51%**	51.44%	47.72%	48.49%	51.39%
JEdit	**56.62%**	48.70%	52.25%	38.79%	56.26%
JFreeChart	**57.80%**	42.77%	49.20%	42.06%	51.23%
JRuby	**53.85%**	45.07%	45.07%	42.14%	48.18%
SQuirrel	**67.10%**	55.39%	58.68%	54.13%	60.37%
Average	**56.66%**	47.55%	47.56%	43.99%	51.68%

TABLE IV
RESULTS OF ALL THE METHODS ON THESE TEN PROJECTS IN TERMS OF MACROP

Project	Ours	NBM	CNN	BiLSTM	NLP
Ant	56.11%	**63.11%**	39.60%	59.67%	58.23%
JMeter	48.21%	**51.94%**	41.05%	37.47%	38.13%
ArgoUML	53.40%	**55.32%**	47.85%	48.54%	50.80%
Columba	48.07%	**71.96%**	52.69%	44.25%	47.17%
EMF	**92.98%**	48.12%	43.15%	34.99%	60.40%
Hibernate	59.15%	**64.26%**	47.80%	53.14%	52.97%
JEdit	57.24%	**65.42%**	51.33%	42.10%	58.13%
JFreeChart	51.32%	**53.37%**	44.04%	40.42%	47.33%
JRuby	55.23%	**57.45%**	47.02%	46.05%	40.80%
SQuirrel	**68.97%**	65.02%	60.33%	57.25%	59.47%
Average	59.07%	**59.60%**	47.49%	46.39%	51.34%

TABLE V
RESULTS OF ALL THE METHODS ON THESE TEN PROJECTS IN TERMS OF MACROR

Project	Ours	NBM	CNN	BiLSTM	NLP
Ant	49.00%	32.31%	38.36%	49.29%	**56.67%**
JMeter	**55.72%**	33.45%	50.42%	37.84%	42.77%
ArgoUML	**47.97%**	39.24%	43.62%	37.23%	43.00%
Columba	**54.72%**	51.56%	51.48%	40.82%	50.50%
EMF	52.38%	38.39%	38.65%	41.67%	**52.70%**
Hibernate	**57.89%**	42.88%	47.64%	44.59%	49.90%
JEdit	**56.02%**	38.79%	53.20%	35.97%	54.50%
JFreeChart	**66.14%**	35.68%	55.74%	43.85%	55.83%
JRuby	52.53%	37.08%	43.27%	38.84%	**58.83%**
SQuirrel	**65.34%**	48.24%	57.12%	51.33%	61.30%
Average	**55.77%**	39.77%	47.95%	42.14%	52.60%

10.72%, respectively. Unfortunately, our method achieves worse results than the NLP-based method and the NBM-based method in terms of MacroF on Ant and Columba, respectively. Combining the results from Tables IV and V, the NBM-based method achieves the best result in terms of MacroP on Columba and the NLP-based method achieves the best result in terms of MacroR on Ant. The potential reason is that the number of selected features is not sufficient and the context has impacts on detecting these three types of technical debt on these two projects. For example, based on the results from RQ2, when selecting 30% of features, our method achieves the best results 60.33%, 65.68%, and 55.79% in terms of MacroF, MacroP, and MacroR on Ant, respectively, and outperforms all the baseline methods.

In Table IV, our method achieves the best results in terms of MacroP only on EMF and SQuirrel. On other projects, the NBM-based method achieves the best results. Combining the results from Table V, we observe that the NBM-based method achieves the best results in terms of MacroP on most of these ten projects, but the lowest results in terms of MacroR on all these ten projects. The potential reason is that implementing the data augmentation strategy for the NBM-based method is different from other methods. Adding newly generated samples in each project to train subclassifiers may lead to high false positive rate, i.e., many non-SATD comments are mistakenly identified as different types of SATD comments. Thus, the NBM-based method achieves high MacroP at expense of MacorR. Comparatively, although our method achieves worse results than the NMB-based method in terms of MacroP, the average result achieved by our method is close to that of the NBM-based method.

In Table V, the MacroR achieved by our method is obvious higher than that achieved by all the baselines on all the projects but Ant, EMF, and JRuby. Among the baseline methods, the NLP-based method obtains better performance than other methods. The reason may be that the NLP-based method is designed to detect different types of technical debt, thus it may be suitable for identifying other types of technical debt, such as defect debt and implementation debt. In these three tables, we can observe that the NLP-based method obtains the best average results in terms of MacroF and MacroR. Although the attention mechanism is introduced to the BiLSTM-based method, lacking sufficient data is the key challenge to learn important features form different types of technical debt comments.

In addition, we use Table VI to display the comparative results of all the methods in terms of F1-measure for each type of technical debt on these ten projects, where “Implem” represents implementation debt. As seen from the table, we observe that all the methods achieves higher F1-measure in detecting design debt than defect debt and implementation debt. A potential reason is that the number of defect debt comments and implementation debt comments is far less than that of design debt comments. Features extracted from the samples belonging to design debt comments may contain many features extracted from the samples belonging to defect debt comments or implementation debt comments. Thus, the weights of the features indicating defect debt and implementation debt are reduced in training the classifier. Although EDA can add the number of samples for defect debt and implementation debt, the diversity of features cannot be improved. For defect debt and implementation debt, our method achieves the best average result than all other methods. However, the BiLSTM-based method achieves the best result than all other methods in identifying design debt comments. The reason may be that the attention mechanism makes the BiLSTM model learn more features from design debt comments.

In summary, our approach is effective in identifying design debt and achieves better average results than the baseline methods in detecting defect debt and implementation debt.

B. *Investigation to RQ2*

1) Motivation: As mentioned earlier, feature selection plays a decisive role in training the classifier for identifying these three types of technical debt. In our method, CHI is applied to calculate the score of each feature and the features with the highest scores are selected. By default, we select 10% of features to train the classifier. However, the results achieved by our method

TABLE VI
RESULTS OF ALL METHODS ON THESE TEN PROJECTS IN TERMS OF F1-MEASURE FOR EACH TYPE OF TECHNICAL DEBT

Project	**Ours**			**NBM**			**CNN**			**BiLSTM**			**NLP**		
	Design	Defect	Implem	Design	Defect	Implem	Design	Defect	Implem	Design	Defect	Implem	Design	Defect	Implem
Ant	85.56%	36.52%	60.00%	21.27%	15.89%	24.93%	81.11%	14.29%	20.00%	90.63%	33.33%	33.33%	91.20%	33.30%	54.50%
JMeter	83.95%	21.62%	41.67%	43.15%	23.95%	35.27%	76.48%	18.39%	27.27%	86.71%	14.76%	21.28%	84.20%	29.10%	22.20%
ArgoUML	77.47%	38.72%	32.46%	60.18%	18.21%	37.07%	74.93%	35.44%	23.86%	78.32%	14.79%	12.50%	79.00%	19.00%	33.20%
Columba	67.82%	52.50%	26.67%	47.35%	51.00%	71.16%	75.76%	48.48%	30.00%	78.92%	15.00%	28.57%	70.90%	17.40%	50.00%
EMF	88.24%	23.33%	66.67%	21.16%	49.21%	16.96%	80.31%	16.67%	22.22%	83.33%	0.00%	30.77%	84.10%	33.30%	25.00%
Hibernate	81.21%	42.86%	48.84%	51.46%	25.96%	56.01%	79.11%	37.50%	23.08%	83.67%	17.39%	39.47%	82.30%	34.10%	35.60%
JEdit	87.26%	28.57%	41.51%	39.76%	22.60%	45.05%	88.14%	45.16%	13.33%	86.93%	0.00%	21.28%	87.20%	33.30%	44.40%
JFreeChart	78.67%	39.13%	40.00%	42.49%	53.27%	27.52%	79.25%	19.35%	38.46%	88.89%	37.04%	0.00%	76.60%	35.30%	31.10%
JRuby	76.59%	45.75%	36.94%	48.29%	36.05%	24.90%	72.61%	33.80%	25.50%	75.48%	21.14%	19.23%	76.00%	22.00%	36.40%
SQuirrel	86.57%	46.38%	65.22%	37.20%	58.54%	41.09%	83.38%	50.67%	40.91%	81.63%	34.38%	41.86%	81.40%	50.00%	49.40%
Average	81.33%	**37.54%**	**46.00%**	41.23%	35.47%	38.00%	79.11%	31.98%	26.46%	**83.45%**	18.78%	24.83%	81.29%	30.68%	38.18%

TABLE VII
MACROF ACHIEVED BY OUR APPROACH WITH VARYING PERCENTAGES OF SELECTED FEATURES

Project	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
Ant	52.32%	54.41%	**60.33%**	53.60%	50.80%	49.28%	49.89%	56.41%	50.44%	50.44%
JMeter	51.70%	**52.85%**	51.98%	51.13%	51.18%	49.75%	49.43%	50.02%	49.61%	49.05%
ArgoUML	50.54%	52.30%	53.55%	53.44%	53.33%	53.15%	**55.26%**	54.41%	53.16%	52.53%
Columba	51.18%	51.82%	53.08%	56.28%	54.06%	52.69%	**56.75%**	56.00%	54.47%	53.11%
EMF	**67.01%**	61.70%	57.20%	62.95%	54.10%	58.33%	57.20%	59.92%	63.75%	61.01%
Hibernate	**58.51%**	54.62%	57.98%	56.74%	56.57%	55.29%	56.06%	56.27%	54.22%	53.65%
JEdit	56.62%	56.70%	55.57%	54.50%	53.94%	57.74%	56.93%	56.39%	**58.84%**	57.22%
JFreeChart	57.80%	**59.25%**	59.11%	51.40%	54.17%	51.72%	56.16%	52.53%	50.50%	49.85%
JRuby	**53.85%**	50.78%	51.91%	49.53%	50.34%	50.75%	50.03%	50.18%	48.83%	48.85%
SQuirrel	**67.10%**	66.04%	65.49%	58.54%	62.00%	63.99%	66.02%	63.89%	62.83%	62.93%
Average	**56.66%**	56.05%	56.62%	54.81%	54.05%	54.27%	55.37%	55.60%	54.67%	53.86%

TABLE VIII
MACROP ACHIEVED BY OUR APPROACH WITH VARYING PERCENTAGES OF SELECTED FEATURES

Project	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
Ant	56.11%	57.33%	**65.68%**	60.59%	55.32%	52.29%	53.20%	60.59%	54.98%	54.98%
JMeter	48.21%	49.21%	**50.00%**	49.18%	48.96%	46.98%	45.76%	47.67%	46.84%	46.61%
ArgoUML	53.40%	56.09%	58.35%	58.16%	58.26%	58.68%	**61.34%**	60.27%	58.55%	57.74%
Columba	48.07%	50.02%	51.09%	**56.16%**	52.35%	51.20%	55.09%	54.40%	52.40%	50.09%
EMF	**92.98%**	76.22%	70.57%	82.13%	65.11%	69.07%	70.57%	73.71%	84.91%	81.78%
Hibernate	59.15%	57.40%	59.87%	60.19%	58.50%	58.49%	57.73%	**61.01%**	56.95%	56.73%
JEdit	57.24%	59.98%	57.27%	59.57%	55.69%	**69.70%**	59.57%	63.38%	62.24%	59.99%
JFreeChart	51.32%	53.41%	**54.30%**	48.90%	50.95%	49.17%	53.33%	48.39%	48.18%	47.41%
JRuby	55.23%	54.81%	**56.23%**	53.21%	54.17%	55.10%	53.31%	53.49%	51.95%	51.73%
SQuirrel	**68.97%**	67.24%	66.94%	59.15%	62.27%	**68.07%**	67.20%	65.86%	64.53%	64.77%
Average	59.07%	58.17%	**59.03%**	58.72%	56.16%	57.88%	57.71%	58.88%	58.15%	57.18%

TABLE IX
MACROR ACHIEVED BY OUR APPROACH WITH VARYING PERCENTAGES OF SELECTED FEATURES

Project	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
Ant	49.00%	51.78%	**55.79%**	48.05%	46.96%	46.59%	46.96%	52.76%	46.59%	46.59%
JMeter	55.72%	**57.07%**	54.12%	53.24%	53.62%	52.86%	53.75%	52.61%	52.74%	51.75%
ArgoUML	47.97%	49.00%	49.48%	49.43%	49.17%	48.57%	**50.28%**	49.59%	48.68%	48.18%
Columba	54.72%	53.75%	55.23%	56.39%	55.89%	54.26%	**58.52%**	57.70%	56.71%	56.52%
EMF	**52.38%**	51.83%	48.10%	51.03%	46.27%	50.48%	48.10%	50.48%	51.03%	48.65%
Hibernate	**57.89%**	52.10%	56.20%	53.66%	54.76%	52.43%	54.49%	52.22%	51.73%	50.88%
JEdit	**56.02%**	53.76%	53.97%	50.22%	52.29%	49.28%	54.51%	50.79%	55.80%	54.69%
JFreeChart	**66.14%**	66.52%	64.87%	54.17%	57.81%	54.55%	59.31%	57.44%	53.05%	52.56%
JRuby	**52.53%**	47.30%	48.21%	46.33%	47.03%	47.04%	47.13%	47.25%	46.07%	46.27%
SQuirrel	**65.34%**	64.89%	64.11%	57.95%	61.73%	60.36%	64.89%	62.03%	61.22%	61.19%
Average	**55.77%**	54.80%	55.01%	52.05%	52.55%	51.64%	53.79%	53.29%	52.36%	51.73%

may vary when selecting different percentage of features. For convenience, we use k to represent the percentage of selected features. In addition, feature selection may remove some features with important semantic information related to these three types of technical debt. In this RQ, we mainly investigate the impacts of different k values on our approach and attempt to seek the appropriate k value to ensure that our method can achieve good results on all projects.

2) Approach: To answer this RQ, we compare the performance of our approach with feature selection and without feature selection. In the experiment, we vary the number of selected features (namely k) from 10% to 100% with the default step 10%. Actually, when 100% of features are selected, it indicates that feature selection is no longer useful. In order to determine the appropriate k value, we count the number of the best results achieved by our method under each k value. One "best" refers to the best result in terms of a certain evaluation metric on a certain project. More specially, for each evaluation metric (MacroF, MacroP, and MacroR), we first count the local number of the best results achieved by our method under each k value. Then, the total number under each k value is calculated by summing the local results. Thus, the appropriate k value with the maximum number can be determined.

3) Results: Tables VII– IX present the results of our approach in terms of MacroF, MacroP, and MacroR on each project when selecting different percentage of features, respectively. In the tables, the best results on the ten projects are highlighted in bold.

By an observation on the results in Tables VII–IX, our approach achieves different results in terms of MacroF, MacroP, and MacroR with the change of k on all these ten projects. When k is equal to 10%, our approach obtains the best results in terms of MacroF on EMF, Hibernate, JRuby, and SQuirrel, in terms of MacroP on EMF and SQuirrel, and in terms of MacroR on EMF, Hibernate, JEdit, JFreeChart, JRuby, and SQuirrel. Therefore, our method obtains the best results 12 times when k is set to 10%. Certainly, the best results can be obtained when k is set to other values. For example, our method obtains the best results six times (namely in terms of MacroF on Ant, int terms of MacroP on Ant, JMeter, JFreeChard, and JRuby, and in terms of MacroR on Ant) when k is equal to 30%. The potential reason may be that some important features are mistakenly removed during feature selection. Specifically, the smaller the parameter value, the higher the possibility of important features being removed is. Based on the comparative results, we select 10% as the default parameter value, i.e., $k = 10\%$.

In addition, it is worth noting that our approach does not apply feature selection when selecting 100% of features. As shown in

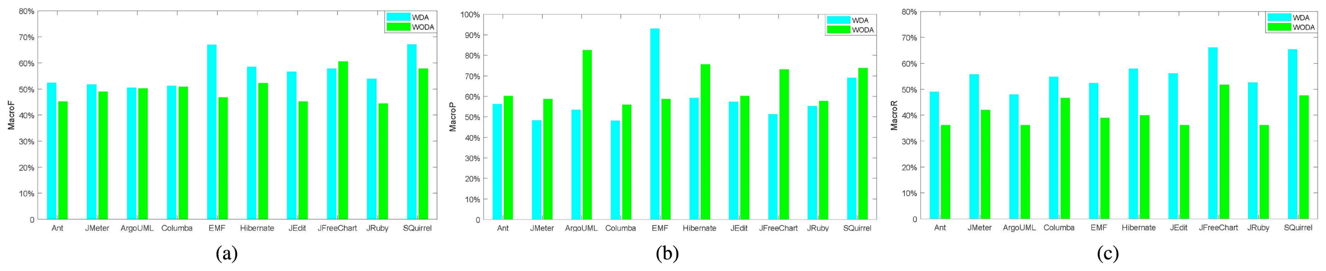

Fig. 3. Comparative results of our approach with data augmentation and without data augmentation in terms of (a) MacroF, (b) MacroP, and (c) MacroR.

Table VIII, when $k = 100\%$, MacroF achieved by our method ranges from 48.85% to 62.93%. In this case, our method does not achieve one "best" compared with other parameter values. In comparison, when $k = 10\%$, our approach with feature selection achieves 56.66% in terms of MacroF on average and outperforms our approach without feature selection by 2.80%. Meanwhile, when k is equal to 100%, the average result is smaller than those of some other parameter values, such 70%, 80%, and 90%. This indicates that feature selection is effective in identifying these three types of technical debt by selecting important features containing more semantic information. The selected features may convey more semantic information.

Our approach achieves different results with the change of the percentage of selected features. Comparatively, when selecting 10% of features, our approach works well. Therefore, 10% is the default parameter value.

C. Investigation to RQ3

1) Motivation: As mentioned earlier, the number of samples belonging to different classes varies differently in different projects, even in the same project. For example, JMeter contains 22 defect debt comments, 316 design debt comments, and 21 implementation debt comments. The number of design debt comments is 14.36 and 15.05 times of that of defect debt comments and implementation debt comments, respectively. The class unbalanced problem may have a great impact on the performance of the classifier. Therefore, we adopt data augmentation to increase the number of defect debt comments and implementation debt comments. However, it is unsure whether the data augmentation strategy is effective or not in improving the performance of classifier. Therefore, in this RQ, we investigate the role of the data augmentation strategy.

2) Approach: To answer this RQ, we compare the results of our approach with data augmentation versus without data augmentation. More specifically, in this experiment, we remove the data augmentation strategy directly and keep other operations unchanged. Meanwhile, we leverage the original samples from nine projects to train the classifier according to the leave-one-out cross-validation. Since the number of selected features has an impact on the classification results, we experimentally determine the best percentage of selected features for our method without data augmentation. Eventually, the percentage is set to 10% in this RQ. For convenience, we use "WDA" and "WODA" to represent our method with data augmentation and without data augmentation, respectively.

3) Results: Fig. 3(a)–(c) presents the comparative results of WDA and WODA in terms of MacroF, MacroP, and MacroR on each project.

As shown in Fig. 3(a) and (c), we observe that WDA achieves better results than WODA in terms of MacroF and MacroR on all the projects. For example, MacroR achieved by WODA ranges from 36.11% to 51.68% on the ten projects, with an average of 41.09%. Comparatively, WDA improves WODA by 8.12%–19.91% on these ten projects, with an average 14.68%. In addition, WDA outperforms WODA in terms of MacroP on EMF, but achieves lower results than WODA on all other projects. The reason may lie in the data augmentation strategy. This strategy will generate new samples for the small classes. Our method may learn the duplicate features from augmented samples and predicts many non-SATD comments with the same features as different types of technical debt comments. Therefore, it may cause that MacroP decreases while MacroR increases.

In our method, we introduce EDA to increase the samples in small classes to break through the class unbalanced problem. However, an important problem is that the generated samples may be with incorrect labels, which may influence the actual results of our approach. Therefore, we ask the second author, a senior, to peruse each generated comment and determine the corresponding label. By comparing the manually analyzed labels with the generated labels, we find that there are indeed 3.4–8.7% of comments with incorrect labels in the generated ones. Most of these comments with incorrect labels belong to non-SATD ones. In such a way, they may lead to that some similar non-SATD comments will be identified as SATD comments by mistake. Thus, our method has a decrease in terms of MacroF. However, the data augmentation can significantly improve our method in terms of MacroR. Thus, our method achieves better results in terms of MacroF when introducing the EDA strategy. In this study, we do not remove the generated comments with incorrect labels since this needs to take extra effort for determining the label for each generated sample and would violate the automation of our approach.

In summary, the data augmentation strategy can effectively improve the performance of our approach.

D. Investigation to RQ4

1) Motivation: In comments, developers may use different words to express different types of SATD. Specifically, the importance of different words may vary sharply. In our approach, we implement feature selection to select features containing

TABLE X
TOP TEN FEATURES OF DEFECT DEBT AFTER FEATURE SELECTION ON EACH PROJECT

Ant	Jmeter	ArgoUML	Columba	EMF	Hibernate	Jedit	JFreeChart	Jruby	Squirrel
work	work	work	work	work	work	work	work	work	work
bug	bug	fixm	fixm	fixm	fixm	fixm	fixm	doe	fixm
cas	problem	bug	bug	bug	problem	bug	problem	bug	bug
problem	cas	nul	test	issu	bug	issu	bug	problem	issu
issu	lin	cur	issu	problem	issu	fix	issu	test	lin
lin	issu	cas	problem	correct	test	problem	lin	cau	test
test	nul	valu	cau	nul	lin	select	select	fixm	cau
nul	cau	tim	wrong	wrong	without	cau	test	fix	problem
rub	correct	ha	select	index	cau	correct	wrong	issu	old
wrong	without	correct	lin	select	may	fail	cau	correct	warn

TABLE XI
TOP TEN FEATURES OF DESIGN DEBT AFTER FEATURE SELECTION ON EACH PROJECT

Ant	Jmeter	ArgoUML	Columba	EMF	Hibernate	Jedit	JFreeChart	Jruby	Squirrel
us	us	us	us	us	us	us	us	us	us
doe	class	doe	doe	nee	nee	doe	nee	return	doe
class	doe	class	cre	return	class	class	doe	method	method
set	set	hack	class	method	doe	set	class	cur	class
return	method	get	set	doe	return	return	return	class	return
method	return	set	ad	set	list	method	set	typ	set
fixm	ha	workaround	method	list	method	check	method	check	check
list	workaround	cod	return	class	ad	cre	cur	cre	cur
cod	fixm	not	cod	cod	set	cur	check	valu	ad
hack	hack	bet	hack	cur	check	cas	cas	cod	hack

TABLE XII
TOP TEN FEATURES OF IMPLEMENTATION DEBT AFTER FEATURE SELECTION ON EACH PROJECT

Ant	Jmeter	ArgoUML	Columba	EMF	Hibernate	Jedit	JFreeChart	Jruby	Squirrel
impl	impl	ad	impl	impl	impl	impl	impl	list	impl
yet	yet	check	read	contain	show	show	show	impl	support
show	show	impl	yet	show	support	yet	yet	show	show
log	jar	allow	show	miss	yet	fdietz	log	support	en
read	log	read	log	log	en	bytel	nd	lin	yet
nd	drag	log	en	yet	miss	en	opt	yet	transl
dirt	langu	com	miss	lin	log	guard	fdietz	put	dirt
ov	stylepanel	auth	bottom	stylepanel	read	emptyfig	en	algorithm	read
jar	eq	yet	button	nd	langu	rang	attrwritermethod	optim	stil
langu	bytel	stil	offend	trycatch	opt	threadsaf	langu	en	log

more semantic information for classification. By XGBoost, we can mine the most important words indicating different types of technical debt. These words will help developers understand technical debt more deeply and provide insights for future research directions. Meanwhile, these words also reveal the intuition behind how our method works to detect these three types of technical debt. In this RQ, we present the top ten words for each type of technical debt on each project.

2) Approach: According to the training data, XGBoost can distinguish the important features by calculating the corresponding weights. Then, the features and their weights are used to determine whether the comment belongs to a specific type of SATD. Intuitively, the larger the corresponding weight, the more important the feature is. Therefore, when each project is regarded as the testing set, we can output the used features from training set and the corresponding weights. We rank these features on the base of their weights and the top ten features are displayed.

3) Results: Tables X–XII present the top ten features of each class when each project is regarded as the testing set. Notably, since we implement the lemmatization operation, the displayed symbols are actually the root form of some words. For example, "us" may represent "using", "use", or "used".

As seen from the tables, we find that the top ten features for the same type of technical debt are basically similar among these ten projects. For example, in Table X, all the ten projects contain the features "work" and "bug", and nine projects contain the feature "issu". Some features (such as "work" and "bug" in Table X) always have high rankings on all the projects, which indicates that if a comment includes these features, its type can be easily identified. In addition, although some features may be not in the tables on some projects (for example, ArgoUML does not contain the feature "issu" in Table X), this does not indicate that these features are not important. The reason is that we only display the top ten features in the tables. For the same type of technical debt, even though all the projects contain the same features, their rankings may vary since their weights are different in different projects.

According to the statistical result from the tables, we find that defect debt, design debt, and implementation debt involve 26, 23, and 40 different features, respectively. In addition, different types of technical debt may contain the same features in the tables. For example, Tables X and XI contain the features "cas(case)", "cur (cursory)". "does (do)", "fixm (fixme)", and "ha have", which indicates that these features are related to defect debt and design debt. Also, there is not a same feature in these three tables. Overall, most of the important features related to each type of technical debt are different, especially features in implementation debt and other types of technical debt. From the perspective of semantics, features in Table X generally convey the information that there are bugs or defects in software or whether the software works normally or not. Meanwhile, features in Table XII indicate that some components or functions may be incomplete.

In the study, Huang *et al.* [4] also summarized the top 30 features with the highest weights for each project in identifying SATD comments. They did not use the common stop word list for stop word removal. By removing the stop words based on the common stop word list and duplicate words, 103 different features are retained. In this study, we show 79 different features for identifying different types of technical debt on these 10 projects. By comparing the features presented in our work and their work, we find that there are only 18 same features, including "work", "hack", "fixm", "use", and "check". There is a big difference between the features. The main reason may be that our work focuses on identifying different types of technical debt, whereas their work focuses on detecting whether a comment is SATD. In such a way, our work aims to mine the specific features for identifying different types of technical debt. For example, comments containing "bug", "problem", "issu", or "wrong" may be defect debt comments. Therefore, some features (which may indicate that a comment is SATD) cannot obtain high weights by the XGBoost method. For example, these three tables do not contain the feature "Todo", which is an important feature in identifying technical debt comments, but it is useless to distinguish different types of technical debt.

We find that different types of technical debt comments usually contain some specific features. With these features, developers can easily judge whether a comment is technical debt and what type it is. We also find that some features are effective for distinguishing SATD from non-SATD, but may be ineffective in judging its type.

VI. Discussion

In software development, technical debt is unavoidable and ubiquitous [4]. It has three aspects of negative impact on the software, including maintainability, evolvability, and visibility [31]. Technical debt can be introduced deliberately or inadvertently. Many studies focus on inadvertent technical debt [32] by source code analysis. However, these methods based on source code analysis usually suffer from high false positive rates [33]. Another majority of work focuses on deliberate technical debt detection by using code comments (namely SATD). Compared with the methods based on source code analysis, leveraging code comments to detect technical debt has two aspects of advantages [1]. First, it is more lightweight since it does not need to construct abstract syntax trees or other complex source code representations. Second, it does not need to set arbitrary metric threshold values, which are usually hard to determine. Actually, studying SATD is useful to prioritize the pay back of debt since developers admit them with comments [1]. A survey with 152 developers from ING Netherlands shows that 88% of the participants mark technical debt with code comments and remove them by refactoring using automated tools (71%) or manually (29%) [34]. Another empirical study also shows that there is a large amount of SATD in 2.4–31% of the files and developers with rich experience tend to introduce SATD [35].

The main impact of technical debt is in making the system more difficult to change in the future [36]. Previous study shows that different types of technical debt has different impacts on software, and design debt has the highest impact [13], [14]. Identifying its type is helpful to remove technical debt. If the type of technical debt can be easily identified, it can help developers understand technical debt better, thus reducing the maintenance cost. To do that, we attempt to identify design debt, defect debt, and implementation debt and propose a new method that adopts the EDA strategy to overcome the class unbalanced problem and leverages XGBoost to build the classifier. First, compared with the baseline methods, our method achieves more or less improvement in terms of F1-measure with respect to different types of technical debt. Our method may help developers save more time in detecting technical debt. Second, the class unbalanced problem is the main challenge in resolving this task. Although we have made some effort to address this problem, it still is not promising, which has thrown light on a practicable direction for the future research. Third, we mine some important features that may help developers to judge the type of technical debt.

VII. Threats to Validity

In this study, we combine the CHI feature selection method and XGBoost to identify three types of SATD, namely defect debt, design debt, and implementation debt. The results show that the proposed approach has better performance than the baseline methods in resolving this problem. However, the effectiveness of this approach is still limited by the following aspects.

A. External Validity

Although we run experiments on a public available dataset, which is also frequently used in existing studies [4], [8], we are unsure whether the proposed approach can be applied to different projects or projects based on other programming languages. This may produce a threat to the generalizability of our method. However, the dataset includes ten projects from different fields. They involve different contributors, different number of SLOC, different number of comments, and different types of technical debt. Many state-of-the-art methods for identifying SATD are validated by this dataset [4], [8]. In addition, our method focuses on processing code comments consisting of natural language information. Even though the projects may be based on other programming languages, such as C++, comments are still composed of natural language information. Certainly, further investigations for the proposed approach are necessary by collecting more data from different projects and fields.

B. Internal Validity

In this study, we adopt the CHI method to conduct feature selection. We first calculate the score of each feature and then select a certain percentage of features with the highest scores. This may bias the experimental results since different projects may achieve the optimal results under different percentage of features. However, we perform a detailed experiment to evaluate the results of the classifier with respect to different percentage of selected features. The experimental results have shown that the classifier achieves good performance on most of these ten projects when the percentage is set to 10%. Although some projects may achieve better results on different percentage, but they also obtain relatively good results when selecting 10% of features. Therefore, this bias is minimized.

Given the class unbalanced problem, we adopt the data augmentation strategy to increase the number of samples in small classes. The strategy adopts the random swap operation to generate new samples based on existing samples. However, the labels of new samples may be not the same with the original samples, which may impact the actual results of the trained classifier. However, the operation neither adds new features to the sample space nor deletes features from original samples. Although the newly generated samples are difficult to understand, but the semantic almost does not change compared with the original samples. Meanwhile, we ask the senior to determine the true labels of the generated comments and run an experiment to obtain the results of all the generated comments with correct labels. The results show that the difference is negligible. Therefore, this impact can be significantly reduced.

C. Construct Validity

In the literature, Huang *et al.* pointed out that some stop words are actually useful for classification [4]. They built a specific stop word list for identifying SATD comments. In our study, we employ the commonly used stop word list rather than the specific stop word list for stop word removal. This may produce a bias to our model. However, the common stop word list is widely adopted in various software engineering problems and its effectiveness has been proven [37]. Therefore, this bias will be greatly reduced. In the future, we will investigate the role of the specific stop word list in detecting different types of technical debt.

As mentioned in [1], only depending on code comments to detect different types of technical debt is not adequate. The reason is because sometimes developers may not realize that they have made technical debt in the software. In addition, the effectiveness of technical debt detection greatly relies on manually inspecting code comments. If the code is not documented well, manual inspection may lead to incorrect annotation of code comments. Therefore, this will produce a threat to our work. However, detecting technical debt by code comments is a complement to existing methods based on source code. In the future, combining code comments and source code to detect technical debt is an important research direction.

VIII. Related Work

Technical debt is a metaphor introduced by W. Cunningham who regards not-quite-right code as a form of debt [2]. So far, technical debt has become a widely accepted and broadly used term in the agile development community [38]. Many researchers also have conducted extensive studies to investigate various aspects of technical debt, such as introduction, diffusion, evolution, impact, and removal [9], [39]–[41] or resolve the tasks related to technical debt [4], [7].

In managing technical debt, one of the most important tasks is technical debt detection. Many researchers have proposed state-of-the-art methods for identifying technical debt. These methods can be divided into two categories, namely methods based on source code and methods based on code comments. The methods based on source code mainly reply on static analysis tools to measure various indicators within source code and set a relevant threshold to determine whether the code contains technical debt. Marinescu *et al.* [13] proposed a measurement based technical debt detection method to help developers directly locate the classes or methods, which violate the design principles of object orientation. Li *et al.* [5] used two modularity metrics, i.e., index of package changing impact and index of package goal focus, to identify architectural technical debt. In addition, Zampetti *et al.* [6] developed a technical debt identification system (TEDIOUS), which leveraged various method-level features (including source code structural metrics, readability metrics, and warnings) as independent variables, and adopted random forest to train a classifier for identifying technical debt.

The methods based on code comments mainly leverage code comments to detect technical debt, which is called SATD. There are two kinds of methods for detecting SATD. The first is methods based on patterns. For example, Potdar and Shihab summarized 62 basic patterns by analyzing source code comments and then adopted pattern matching to identify SATD. Based on this, Ichinose *et al.* [42] developed a visualization tool that is used to visualize the instances related to technical debt based on the 62 basic patterns. The second is methods based on machine learning techniques. Maldonado *et al.* first applied NLP techniques to detect design debt and requirement debt [1]. In the study, they carefully annotated the technical debt samples and formed a public available dataset. Huang *et al.* [4] proposed a text-mining based technical debt detection method, which applied naive Bayes to build a subclassifier for each project and adopted the noting mechanism to predict the label of each code comment. Ren *et al.* [8] first extracted informative features from text and leveraged CNN to construct a classifier for identifying SATD. In addition, they also investigated the role of each textual feature by reverse backing to obtain the important for identifying SATD comments.

Some studies focus on other tasks related to technical debt [6], [39]. For example, Zampetti *et al.* [6] attempted to resolve the problem of technical debt removal. They created a repository of technical debt removal patterns and leveraged CNN to train a multilevel classifier for recommending suitable removal patterns for specific technical debt, including changing API calls, conditionals, method signatures, exception handling, return statements, or telling that a more complex change is needed. Mensah *et al.* [39] focused on the problem of prioritization of technical debt items. They constructed a framework containing a six-step prioritization scheme to minimize SATD based on identified textual indicators. They also empirically investigated the root causes of SATD and evaluated the rework cost of removing technical debt. In addition, Falessi and Reichel [43] presented their attempts on resolving the problem of technical debt measurement. They defined a taxonomy of indicators and developed an effective open-source tool to assess and visualize the interest of technical debt. Besides, Fernández-Sánchez *et al.* [44] conducted a systematic mapping study to discriminate the key elements for managing technical debt. They divided the relevant elements into three groups, i.e., basic decision-making factors, cost estimation techniques, and practices and techniques for decision-making.

To deeply understand technical debt, researchers have performed many empirical studies to reveal various aspects of technical debt. For example, Tom *et al.* [45] presented a theoretical framework to help developers understand the overall phenomenon of technical debt from different dimensions, attributes, precedents, and outcomes. They provided a taxonomy that describes and encompasses different forms of technical debt phenomenon. By investigating five open-source projects, Maldonado and Shihab [9] discovered that technical debt may live in a project for a long time, even more than ten years. Specifically, technical debt will be unconsciously removed by developers in long-term software development. The time for removing technical debt is usually 18–172 days. Nacimento *et al.* [46] performed a qualitative study to analyze the cost factors of technical debt in reflection activities of agile projects. Based on this, they further validated the applicability of cost factors in measuring technical debt. Becker *et al.* [47] found that technical debt caused quality issues in the short terms, but with the long-term of accumulation of technical debt, interest costs further affected the quality. However, different from bugs or defects in a software system, technical debt is invisible since the software often works well from the perspective of users [48].

In this article, we also attempt to leverage code comments to detect technical debt. However, different from aforementioned studies, we focus on identifying three different types of technical debt, including defect debt, design debt, and implementation debt.

IX. Conclusion

To help developers understand technical debt better, we attempted to identify three types of technical debt, including design debt, defect debt, and implementation debt. We therefore proposed a new approach, which was based on XGBoost. More specifically, we first filtered out some identical samples and samples that do not belong to these three types. Then, the data augmentation strategy was adopted to increase the number of samples in small classes. After that, we preprocessed the retained samples and leveraged CHI to extract representative features from the textual feature set. Finally, XGBoost was applied to train the classifier, and the trained classifier was leveraged to predict the corresponding label of each newly arrived code comment. Experimental results showed that the proposed method significantly outperforms the comparative methods. Meanwhile, the data augmentation strategy can improve the learning ability of XGBoost, but it may lead to high false positive. Moreover, by investigating the features of different types of technical debt, we discovered that different types of technical debt comments usually involve some specific features. With these features, developers may not only judge whether a comment is technical debt, but also identify what type the comment belongs to.

In the future, we will collect more data from other fields and projects to evaluate the effectiveness and generalizability of our method. Meanwhile, we will attempt to explore new feature selection methods and data augmentation methods to improve the performance of our method. In addition, we will develop a practicable tool implementing our method and deploy it in a real scenario. Based on this work, we should explore new methods or techniques to identify other types of technical debt, such as test, documentation, and architecture debt. Moveover, features are important in identifying different types of SATD comments. Therefore, we should conduct an empirical study to investigate the role of some specific features or patterns for different types of technical debt.

References

[1] E. da S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt", *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.

[2] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[3] E. da S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*,2017, pp. 238–248.

[4] 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. Eng.*, vol. 23, no. 1, pp. 418–451, 2018.

[5] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt," in*Proc. 10th Int. ACM SIGSOFT Conf. Qual. Softw. Architectures (Part CompArch*2014), Lille, France, 2014, pp. 119–128.

[6] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. D. Penta, "Recommending when design technical debt should be self-admitted," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Shanghai, China, 2017, pp. 216–226.

[7] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th Int. Conf. Softw. Maintenance*, Chicago, IL, USA, 2004, pp. 350–359.

[8] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 1–45, 2019.

[9] E. daS. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. 7th IEEE Int. Workshop Manag. Tech. Debt*, Bremen, Germany, 2015, pp. 9–15.

[10] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, San Francisco, CA, USA, 2016, pp. 785–794.

[11] S. M. Sekhar, G. Siddesh, M. Raj, and S. S. Manvi, "Protein class prediction based on count vectorizer and long short term memory," *Int. J. Inf. Technol.*, vol. 13, no. 1, pp. 341–348, 2021.

[12] D. Yu, L. Wang, X. Chen, and J. Chen, "Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt," *Frontiers Comput. Sci.*, vol. 15, no. 4, 2021, Art. no. 154208.

[13] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Develop.*, vol. 56, no. 5, pp. 9:1–9:13, 2012.

[14] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *Proc. 6th Int. Workshop Manag. Tech. Debt*, Victoria, BC, Canada, 2014, pp. 1–7.

[15] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Comput. Linguistics*, vol. 16, no. 1, pp. 22–29, 1990.

[16] W. Liu, S. Wang, X. Chen, and H. Jiang, "Predicting the severity of bug reports based on feature selection," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 28, no. 4, pp. 537–558, 2018.

[17] H. Jiang, X. Chen, T. He, Z. Chen, and X. Li, "Fuzzy clustering of crowdsourced test reports for apps," *ACM Trans. Internet Techn.*, vol. 18, no. 2, pp. 18:1–18:28, 2018.

[18] P. Vuttipittayamongkol and E. Elyan, "Neighbourhood-based undersampling approach for handling imbalanced and overlapped data," *Inf. Sci.*, vol. 509, pp. 47–70, 2020.

[19] X. Ye, H. Li, A. Imakura, and T. Sakurai, "An oversampling framework for imbalanced classification based on Laplacian eigenmaps," *Neurocomputing*, vol. 399, pp. 107–116, 2020.

[20] J. W. Wei and K. Zou, "EDA: Easy data augmentation techniques for boosting performance on text classification tasks," in *Proc. Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process.*, 2019, pp. 6381–6387.

[21] G. Salton, A. Wong, and C. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[22] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *Proc. Adv. Neural Inf. Process. Syst. 30: Annu. Conf. Neural Inf. Process. Syst.*, Long Beach, CA, USA, 2017, pp. 3146–3154.

[23] G. Petrosyan, M. P. Robillard, and R. D. Mori, "Discovering information explaining API types using text classification," in *Proc. 37th IEEE/ACM Int. Conf. Softw. Eng.*, Florence, Italy, 2015, vol. 1, pp. 869–879.

[24] X. Chen *et al.*, "A systemic framework for crowdsourced test report quality assessment," *Empirical Softw. Eng.*, vol. 25, no. 2, pp. 1382–1418, 2020.

[25] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proc. 12th Eur. Conf. Softw. Maintenance Reengineering*, Athens, Greece, 2008, pp. 329–331.

[26] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescap, "Multi-classification approaches for classifying mobile app traffic," *J. Netw. Comput. Appl.*, vol. 103, pp. 131–145, 2018.

[27] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, 2002.

[28] A. Mccallum and K. Nigam, "A comparison of event models for naive Bayes text classification," in *Proc. AAAI-98 Workshop Learn. Text Categorization*, 1998, pp. 41–48.

[29] N. Beringer, "Fast and effective retraining on contrastive vocal characteristics with bidirectional long short-term memory nets," in *Proc. 9th Int. Conf. Spoken Lang. Process.*, Pittsburgh, PA, USA, 2006, Art. no. 1602-Mon3CaP.8.

[30] Y. Lecun and L. Bottou, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[31] R. Nord, "*The future of managing technical debt*," 2016. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2016/08/the-future-of-managing-technical-debt.html

[32] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Architecture Eur. Conf. Softw. Architecture*, Helsinki, Finland, 2012, pp. 91–100.

[33] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng.*,Osaka, Japan, 2016, vol. 1, pp. 609–613.

[34] C. Vassallo *et al.*, "Continuous delivery practices in a large financial organization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Raleigh, NC, USA, 2016, pp. 519–528.

[35] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol.*, Victoria, BC, Canada, 2014, pp. 91–100.

[36] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng.*, Osaka, Japan, 2016, vol. 1, pp. 179–188.

[37] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng.*, Minneapolis, MN, USA, 2007, pp. 499–510.

[38] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, 2015.

[39] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, and Q. Mi, "On the value of a prioritization scheme for resolving self-admitted technical debt," *J. Syst. Softw.*, vol. 135, pp. 37–54, 2018.

[40] J. Yli-Huumo, A. Maglyas, and K. Smolander, "How do software development teams manage technical debt?—An empirical study," *J. Syst. Softw.*, vol. 120, pp. 195–218, 2016.

[41] N. Zazworka, M. A. Shaw, F. Shull, and C. B. Seaman, "Investigating the impact of design debt on software quality," in *Proc. 2nd Workshop Manag. Tech. Debt*, Honolulu, HI, USA, 2011, pp. 17–23.

[42] T. Ichinose, K. Uemura, D. Tanaka, H. Hata, and K. Matsumoto, "ROCAT on KATARIBE: Code visualization for communities," in *Proc. Appl. Comput. Inf. Technol./ Int. Conf. Comput. Sci./Intell. Appl. Inf./ Int. Conf. Big Data, Cloud Comput., Data Sci. Eng.*, 2017, pp. 158–163.

[43] D. Falessi and A. Reichel, "Towards an open-source tool for measuring and visualizing the interest of technical debt," in *Proc. 7th IEEE Int. Workshop Manag. Tech. Debt*, Bremen, Germany, 2015, pp. 1–8.

[44] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Pérez, "Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study," *J. Syst. Softw.*, vol. 124, pp. 22–38, 2017.

[45] E. Tom, A. Aurum, and R. T. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, 2013.

[46] C. Nacimento, S. Matalonga, and J. C. R. Hauck, "Identifying technical debt cost factors in reflection activities of an agile projects," in *Proc. XL Latin Amer. Comput. Conf.*, Montevideo, Uruguay, 2014, pp. 1–11.

[47] C. Becker, R. Chitchyan, S. Betz, and C. McCord, "Trade-off decisions across time in technical debt management: A systematic literature review," in *Proc. Int. Conf. Tech. Debt*, Gothenburg, Sweden, 2018, pp. 85–94.

[48] K. Dai and P. Kruchten, "Detecting technical debt through issue trackers," in *Proc. 5th Int. Workshop Quantitative Approaches Softw. Qual. Co-Located With 24th Asia-Pacific Softw. Eng. Conf.*, Nanjing, China, 2017, pp. 59–65.