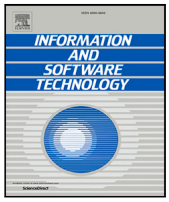
Information and Software Technology 158 (2023) 107190

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Detecting multi-type self-admitted technical debt with generative adversarial network-based neural networks

Jiaojiao Yu [a], Xu Zhou [b], Xiao Liu [c], Jin Liu [a,*], Zhiwen Xie [a], Kunsong Zhao [d]

[a] School of Computer Science, Wuhan university, Wuhan, China
[b] Wuhan United-Imaging Medical Technology Co., LTD, Wuhan, China
[c] School of Information Technology, Deakin University, Geelong, Australia
[d] Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China

ARTICLE INFO

Keywords:
Technical debt
SATD
Generative adversarial network
CodeBERT
Multi-head attention

ABSTRACT

Context: Developers often introduce the self-admitted technical debt (SATD), i.e., a compromised solution to satisfy the delivery of the current goals, in code comments but do not eliminate them timely in the following software development and maintenance process. Automatically identifying the SATDs to reduce potential harm to software has attracted the attention of researchers. However, existing approaches only identified SATDs at a coarse-grained level, which impacts developers to locate and remove them.

Objective: This paper proposes a novel model named GCF, which is a deep learning method to enhance the performance of multi-type SATD classification based on generative adversarial network. Method: The GCF model employs the JSD Generative Adversarial Network to solve the imbalance problem, utilizes CodeBERT to fuse information of code snippets and natural language for initializing the instances as embedding vectors, and introduces the feature extraction module to extract the instance features more comprehensively.

Results: The experimental results show that, the GCF model obtains better performance compared with the state-of-the-art method. Moreover, experiments on the GCF model variants and others with different GAN models show the superiority of the GCF model.

Conclusion: Our proposed GCF model effectively solves the problem of imbalanced types of SATD, fuses the information of code snippets and natural language, and extracts key features to achieve outstanding performance in detecting multi-type SATD. Therefore, the GCF model is an effective method for detecting multi-type SATD.

1. Introduction

Software projects often face unavoidable issues such as quality improvement, delivery time reduction, and budget cut [1]. When faced with these issues, software developers often need to choose limited solutions to satisfy the current software development goals, namely creating technical debt [2]. However, developers introduce technical debts unintentionally or intentionally. For example, a developer sets a wrong parameter because he or she is not fully familiar with the new architecture, which will cause a bug in the system. In that case, the developer introduces an unintentional debt. Moreover, unintentional debt is difficult to be traced. A developer applies the framework function, but this function has a defect in the design of the framework, which has not been fixed yet. Thus, he or she has documented where the function is used. When the framework is updated, the developer can modify the code according to the documentation to eliminate the potential risk of a defective function. In this scenario, the developer introduces an intentional debt. This debt can be eliminated at the later software maintenance stage by finding the relevant documented content [3]. In particular, we denote the intentionally introduced and documented debt as self-admitted technical debt (SATD) [4].

Whether the debt is unintentional or intentional, the technical debt had a negative impact on the maintenance of software for a long time [5–9]. However, previous studies [10] have shown that technical debt is inevitable and widespread in software development. Therefore, it is essential to invent automatic methods to detect technical debt and help developers have a better understanding of it so that they can fix it as quickly as possible [11]. Furthermore, it has the correct path to trace the debt as the developers mark it in the code comments [12]. Thus, it is also treated as a text classification task to identify SATD in code comments. In recent years, deep learning methods have been widely utilized in text classification tasks [13–15] as they can use semantic information and link contexts for efficient text category identification.

* Corresponding author.
E-mail address: jinliu@whu.edu.cn (J. Liu).

https://doi.org/10.1016/j.infsof.2023.107190
Received 29 August 2022; Received in revised form 27 February 2023; Accepted 1 March 2023
Available online 6 March 2023

```
// TODO: Tentative implementation. Do we want something that updates? the list of open projects or just simple open and close events? -tfm
```

Fig. 1. The code comment contains implementation debt in the ArgoUML project.

Therefore, applying the deep learning method is suitable for identifying SATD.

Ren et al. [16] proposed a convolutional neural network-based approach for classifying code comments as SATD or non-SATD. Their experimental results on 10 projects showed that their method obtained better performance than the state-of-the-art text mining-based SATD classification methods. Wang et al. [17] proposed a HATD method for classifying code comments as SATD or non-SATD, which utilized the Embeddings from Language Models (ELMO) [18] to get the word embedding. The attention mechanism was used to obtain the potential representation of the word, and then they used the sigmoid function to identify SATD. Their experimental results on 20 projects showed that their method could effectively identify SATD in both within-project and cross-project scenarios. In addition, Chen et al. [19] proposed the first method to perform the three classifications and obtain state-of-the-art performance in the multi-type SATD classification task (i.e., defect debt, design debt, and implementation debt). They employed the chi-square to select the features and eXtreme Gradient Boosting (XGBoost) to detect multi-types of SATDs.

1.1. Motivation

However, existing methods have the following limitations in identifying multi-types of SATDs. First, existing researches lack more detailed classification for SATD types (i.e., defect debt, design debt, and implementation debt). The previous studies mainly focused on identifying whether the comments contain SATD or not [20]. However, identifying specific types of SATD is a fundamental task that can help developers better understand technical debt [19] and guide developers to eliminate such SATD. As shown in Fig. 1, an implementation debt may need to be fixed after confirming with the project product-related personnel. Therefore, we need to focus on the identification of multi-types of SATDs.

Second, existing researches lack the use of semantic information as well as word order that SATD contains, and cannot expand the diversity of samples to balance data. Previous work [19] employed the traditional feature extraction methods to choose the key features that consider word frequency information, but lose a lot of semantics and word orders [21–23]. Moreover, the two methods used to deal with the imbalance SATD categories were the weighted cross-entropy loss and the traditional data augmentation method in previous studies [16,19]. However, these methods cannot expand the diversity of few number samples to balance data. Therefore, we need to make full use of the semantic information contained in SATD to obtain effective features in detecting multi-types of SATDs. In addition, we need to expand the diversity of samples to enhance the features containing few number data, so as to improve the performance of few number data.

Third, existing researches lack the fusion of the features of code snippets. SATD is represented by code comments composed of code snippets and natural language sequences, between which the semantics are different. Previous studies [16,17,24] regarded code comments as natural language sequences for modeling, but SATD contains not only natural language sequences, but also code snippets. Therefore, we need to integrate the feature of the code snippets and natural language sequences to enhance the performance of the model.

1.2. Our work

To overcome these problems, we propose a neural network model based on a generative adversarial network for detecting multi-types of SATDs. Specifically, we employ the Jensen–Shannon divergence Generative Adversarial Network (JSD-GAN) model [25] to balance the data, then we convert the instance to an embedding vector by CodeBERT model [26] which can fuse the information of code snippets and natural language. After that, we utilize the feature extraction module to extract the features of word embedding. Finally, we apply the softmax function to detect multi-types of SATDs in code comments containing SATD.

We conduct comprehensive experiments on a benchmark dataset that includes 10 open-source projects by previous work [27]. To evaluate the performance of our model in detecting multi-types of SATDs, we employ three indicators, including Precision, Recall, and F-score. Compared with the baseline methods, our model obtains better average performance in terms of Precision, Recall, and F-score, which are 0.6369, 0.5874, and 0.5866, respectively.

1.3. Contributions

The major contributions of this paper are summarized as follows:

- We propose a Generative Adversarial Network-based neural networks, named GCF model, to detect multi-types of SATDs. The GCF model consists of three parts: JSD-GAN, embedding vector, and feature extraction module. The trained GCF model can classify SATD and obtain more detailed categories.
- We employ the GAN to deal with the problem of data imbalance, which can effectively expand the diversity of instances and enhance the representation of features.
- We conduct comprehensive experiments to evaluate the performance of the GCF model and compare with the state-of-the-art (SOTA) method on public datasets. The results show that our method performs better than the SOTA method.

1.4. Paper organization

The remainder of this paper is organized as follows. Section 2 provides the related work. Our model is introduced in detail in Section 3. Sections 4 and 5 describe the experimental setup and analyze the experimental results, respectively. We discuss the parameter impacts and error analysis in Section 6. Section 7 clarifies the threats to validity. Finally, we draw the conclusion in Section 8.

2. Related work

2.1. SATD identification

There are currently three practical methods for SATD identification tasks, including pattern-based method, machine and deep learning method, and change-level method.

The first method is based on pattern recognition. de Freitas Farias et al. [28] proposed a contextualized vocabulary model that ranked the patterns that belonged to the vocabulary to determine the degree of determination of the patterns in SATD, and established the relationship between debt patterns and debt types, for identifying SATD. Guo et al. [29] introduced a match task annotation tags (MAT) method that obtained a task annotation tags list and employed the fuzzy matching mechanism with task annotation tags list to identify SATD in code comments. Huang et al. [10] proposed an automated text-mining method that utilized feature selection to select useful features, and constructed a composite classifier by combining multiple classifiers from different source projects to identify SATD in code comments. Liu et al. [30] introduced a SATD detector tool of the Integrated Development Environment (IDE) that employed Information Gain (IG) to select the key

feature, trained the sub-classifier by Naive Bayes Multinomial (NBM), built a composite classifier from all the sub-classifier to identify SATD in code comments.

The second method is based on machine learning and deep learning methods. Yu et al. [31] proposed a Jitterbug framework that used a fixed pattern to identify the "easy to find" SATDs, and utilized machine learning techniques to assist human experts in identifying "hard to find" SATDs. Yu et al. [32] introduced a deep learning method that employed Bidirectional Long Short Term Memory (BiLSTM) networks with the attention mechanism to capture the key features, utilized the balanced cross entropy loss function to overcome the unbalance problem, and trained the classifier to identify SATD. Wattanakriengkrai et al. [33] introduced an automated model that applied N-gram Inverse Document Frequency (IDF) to select the key features, and employed the auto-sklearn tool to find the best classifier and hyper-parameters, for identifying SATD.

The third method is based on the change-level document. Yan et al. [34] proposed a change-level SATD method, which identified SATD from source code comments of all versions of source code files. Then, they marked changes that first introduced SATD comments as changes that introduced technical debt. Next, they extracted 25 features from software changes and classified them into three dimensions: diffusion, history, and information, to build a determination model. They conducted experiments on 7 open source projects with 100, 011 software changes. The results showed that the model achieved a better performance than the baseline method in terms of AUC and cost-effectiveness. The diffusion was the most discriminative dimension among the three feature dimensions in determining technical debt-introduced changes.

Traditional pattern-based SATD recognition is based on massive manual work, and the results are marked with significantly biased personal information. Improved text mining-based methods are based on feature engineering techniques to extract relevant features, and this technique is highly dependent on expert effort. The quality of features severely affects classification performance. Machine learning and deep learning methods are mainly performed in the binary classification of SATD. Moreover, existing methods adopt for data imbalance on cross-entropy functions to increase weights or through traditional data augmentation, which does not expand the diversity of the dataset and enhance features. The change-level SATD requires many document versions for information collection.

Therefore, our proposed model utilizes the JSD-GAN to solve the data imbalance problem. Our model can effectively expand the diversity of instances and retain the semantic connection of instances, enhancing the features. The feature extraction module effectively fuses the sequence information in the word embedding and can better extract the feature and semantic information between instances.

2.2. Generative adversarial networks

As our model uses generative adversarial networks, we also give an introduction about these studies. Goodfellow et al. [35] proposed the generative adversarial network model firstly, which included the generative and the discriminative models. The optimal generative mode can deceive the discriminative model with the synthetic samples, and the optimal discriminative model is distinguishable between synthetic and true samples. Arjovsky et al. [36] proposed a WGAN framework that replaced the JSD on the standard GAN framework with the Wasserstein Distance, and used the estimated difference measure from the discriminator to calculate the importance of the generated samples, thus providing a policy gradient for training the generator. Wang et al. [37] proposed an E-GANs framework that used pre-defined adversarial objective functions to train generators and discriminators alternately, and employed an evaluation mechanism to measure the quality and diversity of the generated samples so that only high-performance generators were retained and used for further training. Guo et al. [38] introduced a LeakGAN model that leaked the high-level extracted features to generators by discriminator, incorporated the information in generators, applied the extracted features of current generated words, and outputted a latent vector to guide the module for next word generation.

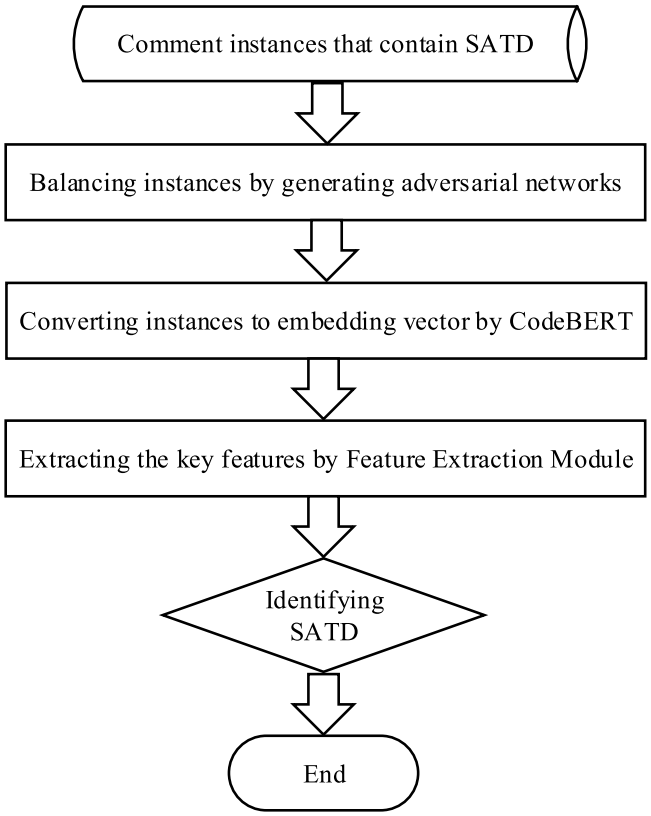


Fig. 2. Overview of our GCF model.

We use JSD-GAN [25], which replaces the discriminator in the standard GAN model with a closed form solution, which directly optimizes the model distribution and the empirical distribution of the training data. Thus, the minimax optimization procedure of JSD-GAN is implicitly performed for the generator and discriminator. Hence, it becomes computationally tractable to optimize the JSD to train generators that generate discrete data sequences.

3. Proposed method

3.1. Framework overview

Fig. 2 presents an overview of our proposed GCF model that consists of the following steps. First, we augment the comment by generating synthetic comments for the categories that only contain a few instances. Such an operation can relieve the imbalance issue and make the model more robust. To better represent each instance, we adopt the CodeBERT model to obtain the initialization embedding vectors because comment instances include some code snippets, and such model considers the code information. We then propose a feature extraction layer incorporating a global feature encoder with a key feature extraction to extract information from the embedding vector. The learned representations are passed into a fully connected layer, and followed by a softmax function to identify specific types of SATD. Below, we detail each component of our GCF model.

3.2. Data enhancement

Since the SATD comments are extraordinarily imbalanced [27], such an issue makes the feature learning for the minority instances (such as defect debt, design debt, and implementation debt) difficult and deteriorates the classification performance. To deal with this problem, we want to adopt data augmentation techniques to supplement the instances in minority categories. Traditional data augmentation techniques can resolve the imbalance categories, such as over-sampling and under-sampling . Over-sampling replicates the instances and does not expand the diversity of the instances, while under-sampling selects a portion of the instances that reduces the number of instances [39–43]. Consequently, traditional data augmentation methods cannot effectively enhance the features of the instance. A more helpful technique, i.e., Easy Data Augmentation (EDA), has been widely used in recent studies to generate synthetic data from original data [44]. EDA contains four tricks, including synonym replacement (SR), random insertion (RI), random swap (RS), and random deletion (RD) [44,45]. However, this technique has the following limitations: the generated synthetic data by SR and the original data can be considered the same, and there are no valid extended data in practice; RI makes the original data lose its semantic structure and semantic order; RS does not essentially change the vocabulary components of the original data; RD leads to data missing key features and reducing the correctness of the class labels. As an optimal solution, Generative Adversarial Network (GAN) has been widely used in many textual generation tasks [46–48]. Unlike traditional data augmentation techniques, GAN makes the generated data closer to the original data by preserving the semantic information of seed data. Our work use the JSD-GAN technique [25] for data augmentation.

The standard GAN model comprises a generator (denoted as G) and a discriminator (denoted as D). The goal of G is to generate synthetic data to deceive the discriminator D. The goal of D is to distinguish the data generated by G and the real data. Thus, G and D can be treated as a dynamic game process. Ideally, G can generate synthetic instances that look like the real data and make it difficult for D to recognize. We consider that the generator G generates suitable instances in this case. This optimization process can be formalized as follows.

$$\begin{aligned} min_G max_D L(D,G) = {} & E_{a\ \tilde{p}_o(a)}[log(D(a))] \\ & + E_{b\ \tilde{p}_g(b)}[log(1-D(b))] \end{aligned} \tag{1}$$

where a is the seed data and b is the generated data; $D(*)$ denotes the probability of D that determines whether one instance is original or generative data, $\tilde{p}_o$ denotes as the empirical distribution, $\tilde{p}_g$ denotes as the generative distribution.

However, the standard GAN has low performance in generating discrete data sequences because the output of the standard generator is discrete, and difficult to pass the gradient that updates from the discriminator to the generator [49,50]. Moreover, the standard GAN has a potential risk that is the disappearance of the gradient derived from the G in the optimal process by using neural networks as discriminators [25], and express unstable behaviors in practice [51,52]. Hence, we only use the neural network as the generator and adopt the closed solution as the optimal solution for the discriminator, the formula is as follows,

$$max_{D(x)} = \begin{cases} \frac{\tilde{p}_o(x)}{\tilde{p}_o(x)+p_g(x)} & \text{if } x \in T \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where T denotes the set of real data, $max_{D(x)}$ denotes as the optimal solution for the discriminator. Therefore, the maximum–minimum optimization process of the standard GAN as in Eq. (1), which is simplified in our model to minimizing G. We obtain the minimizing G by computing the JSD between the empirical and generative distributions. In particular, we simplify the Kullback Leibler (KL) scatter associated with the generative and empirical distributions to the sum of the training data. The KL scatter, and JSD formulas are as follows,

$$\begin{aligned} KL(\tilde{p}_o(x) \parallel (\tilde{p}_o(x)+\tilde{p}_g(x))) = {} & \sum_{x\in T}[\tilde{p}_o(x)log(\tilde{p}_o(x)) \\ & - \tilde{p}_o(x)log(\tilde{p}_o(x)+\tilde{p}_g(x))] \\ JSD(\tilde{p}_o(x) \parallel \tilde{p}_g(x)) = {} & \frac{1}{2}KL(\tilde{p}_o(x) \parallel (\tilde{p}_o(x)+\tilde{p}_g(x))) \\ & + \frac{1}{2}KL(\tilde{p}_g(x) \parallel (\tilde{p}_o(x)+\tilde{p}_g(x))) \end{aligned} \tag{3}$$

Next, we need to use the above equation to do the optimal solution for the generator, which is as follows,

$$\begin{aligned} L(G, max_{D(x)}) &= 2JSD(\tilde{p}_o(x)|\tilde{p}_g(x)) - log4 \\ &= KL(\tilde{p}_o(x) \parallel (\tilde{p}_o(x)+\tilde{p}_g(x))) \\ &+ KL(\tilde{p}_g(x) \parallel (\tilde{p}_o(x)+\tilde{p}_g(x))) \\ &= \sum_{x\in T}[\tilde{p}_o(x)log(\tilde{p}_o(x)) \\ &- \tilde{p}_o(x)log(\tilde{p}_o(x)+\tilde{p}_g(x))] \\ &+ \sum_{x\in T}[\tilde{p}_g(x)log(\tilde{p}_g(x)) \\ &- \tilde{p}_g(x)log(\tilde{p}_o(x)+\tilde{p}_g(x))] \end{aligned} \tag{4}$$

where $L(G, max_{D(x)})$ denotes the optimization process for G. In previous study [25], we can see that the optimal solution of the discriminator $max_{D(x)}$ was the closed form solution. If the comment instances are not the comment instances in T which should be considered false, $max_{D(x)}$ output value is 0, and $L(G, max_{D(x)})$ value is also 0. If the comment instances are the comment instances in T which should be considered true, $max_{D(x)}$ output value equals the ratio of the empirical distribution to the sum of the empirical and generative distributions, which means that the generator only needs the same comment instances as in T to maximize the value function by Eq. (4). The generator assigns a probability to each comment instance and evaluates the effect of each comment instance on the value function. Our method is equivalent to directly optimizing the JSD without sampling from the generators so that optimizing the JSD becomes computationally tractable to train generators that generate discrete text sequences. We optimize JSD to get the most effective generator for SATD instances. With the generator, we input all types of instances, except for the maximum percentage of SATD categories, into the generator to generate the required amount of comment instances for balancing the dataset. For example, the original sample contains three types of SATD which denotes as $C_{imbalance} = \{c_1^1, c_2^1, \dots, c_{10}^1, c_1^2, c_2^2, c_1^3, c_2^3, c_3^3\}$, c^1 denotes the first category SATD that contains 10 instances, c^2 denotes the second category SATD that contains two instances and c^3 denotes the third category SATD that contains three instances. After the generation, we can get the new dataset that denotes as $C = \{c_1^1, c_2^1, \dots, c_{10}^1, c_1^2, c_2^2, \dots, c_8^2, c_1^3, c_2^3, \dots, c_8^3\}$. In particular, the number of generated instances are set to $0.8 \times (c^2 - c^1)$ for $c2$ and $0.8 \times (c^3 - c^1)$ for $c3$.

3.3. Embedding vector initialization

As SATD is expressed as the code comment written by developers, it is essentially a textual sequence. Differently, it always contains some code snippets to make a comment more readable and understandable. To obtain the initialization embedding vectors for comment instances, we adopt the advanced CodeBERT model [26,53] because it inherently takes the natural languages and code information into account. CodeBERT is a pre-training model for programming languages and natural languages based on transformer-based neural network architecture. It designs two pre-training objectives, including the masked language model and replacement token detection, to make the model learn more semantic information from corpora. When training the model, natural textual and code sequences are spliced together, separated by a specific token [SEP]. Then, two tokens [CLS] and [EOS] are put in the front and

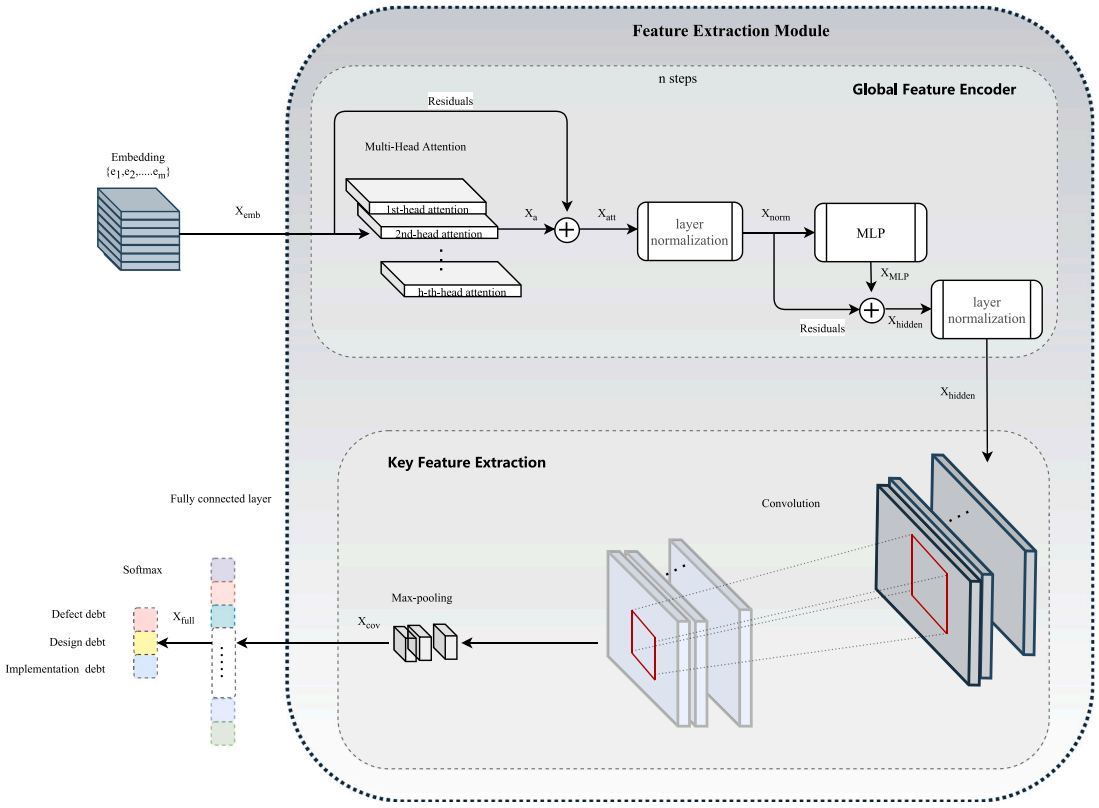


Fig. 3. Flowchart of the feature extraction module.

at the end of this spliced sequence, respectively, in which [CLS] holds the overall representation of the whole sequence and [EOS] represents the ending.

In this work, we first tokenize the comment instance and then initialize the embedding vectors for each word in the sequence using CodeBERT. Formally, for a given comment instance $X = \{x_1, x_2, \dots, x_m\}$ where $x \in C$, x_i represents the ith word and m denotes the sequence length, we can produce the corresponding embedding vectors X_{emd} based on the following equation.

$$X_{emd} = CodeBERT(X) \tag{5}$$

where $X_{emd} = \{e_1, e_2, \dots, e_m\}$ represents the embedding vectors and $e_i \in \mathbb{R}^k$ represents the embedding of ith word, k is the dimension of embeddings.

3.4. Feature extraction module

After we initialize the embedding vector for each comment instance, we want to learn implicit semantic features to represent each comment instance better, aiming to identify more SATDs. For this purpose, we propose a feature extraction module shown in Fig. 3 that consists of the following components.

Global Feature Encoder. To extract features from the embedded comment instances, we design a global feature encoder for integrating the sequence information [54]. Moreover, we employ the multi-head attention mechanism in the encoder to learn discrepancy feature representation and pay more attention to features that are more relevant for the final goal (i.e., identifying SATD). For the input embedding X_{emb}, we generate three matrices K, Q, V as follows,

$$\begin{aligned} Q &= X_{emd} * \omega_Q \\ K &= X_{emd} * \omega_K \\ V &= X_{emd} * \omega_V \end{aligned} \tag{6}$$

where $\omega_Q, \omega_K, \omega_V$ are learnable parameters, Q, K, V are produced by a linear mapping with X_{emb}. Q denotes the query, K denotes the key, V denotes the value. We obtain the sets (Q, K, V) by h linear transformations, where h denotes the number of multi-head. Next, we calculate the self-attention of each set which the formula as follows,

$$head_j(Q_j, K_j, V_j) = softmax\left(\frac{Q_j K_j^T}{\sqrt{d_k}}\right) V_j \tag{7}$$

where j denotes the jth set by linear transformations, d_k denotes the variance of K. we concatenate the h-heads $Head\{head_1, head_2, .., head_h\}$ of self-attention and the linear mapping to the final output which denotes as X_a. Then, we introduce the residual network that fuses the attention information with word embedding and prevents network degradation, and employ layer normalization to normalize the output of the residual network which accelerates convergence by the standard normal distribution. The formulas are as follows,

$$\begin{aligned} X_{att} &= X_{emd} + X_a \\ X_{norm} &= LayerNorm(X_{att}) \end{aligned} \tag{8}$$

where X_{att} denotes the result of residual network, X_{norm} denotes the result of layer normalization.

Next, we put X_{norm} into a feed-forward network which consists of two linear layers and an activation function, and the output is X_{mlp}. Moreover, we combine X_{mlp} and X_{norm} to repeat the Eq. (8). Repeating n steps of the above operations can obtain the global feature encoder X_{hidden}.

Key Feature Extraction. In the previous study [14], the convolutional neural network (CNN) can effectively extract critical information from the text. Therefore, we select CNN to extract the critical information from the global feature encoder X_{hidden} in our model. To obtain the better feature representation, we set the convolution kernel to different window sizes, and slide the whole row on X_{hidden} by convolution kernel. The multiple convolution kernels generate feature representation

Table 1
The information collection for the dataset.

Project	Release	#Comments	#Defect	#Design	#Implementation
Apache Ant	1.7.0	4,137	13 (0.31%)	95 (2.30%)	13 (0.31%)
JMeter	0.4	9,548	22 (0.23%)	316 (3.31%)	21 (0.22%)
ArgoUML	1.4	6,478	127 (1.96%)	801 (12.36%)	411 (6.34%)
Columba	2.4.1	4,401	13 (0.30%)	126 (2.86%)	43 (0.98%)
EMF	3.3.2	2,968	8 (0.27%)	78 (2.63%)	16 (0.54%)
Hibernate	4.2	10,322	52 (0.50%)	355 (3.44%)	64 (0.62%)
JEdit	0.0.19	4,423	43 (0.97%)	196 (4.43%)	14 (0.32%)
JFreeChart	2.1	8,162	9 (0.11%)	184 (2.25%)	15 (0.18%)
JRuby	1.4.0	4,897	161 (3.29%)	343 (7.00%)	110 (2.25%)
SQuirrel	3.0.3	7,230	24 (0.33%)	209 (2.89%)	50 (0.69%)

in different dimensions and utilize the maximum pooling to take out a key feature representation. Finally, we concatenate the whole critical features to form a vector as the output of the feature extraction module X_{cov}, where X_{cov} denotes the key feature representation obtained by the feature extraction module.

3.5. Model training and test

Once the output of the feature extraction module is obtained, we utilize the fully-connected layer to fuse the critical feature extraction information. We then take the results of the fully-connected layer to obtain the probability distribution of each category by the softmax function. We use the cross-entropy loss function to optimize our model, which is defined as:

$$Loss(\tilde{p}_p, \tilde{p}_t) = -\sum_{i=1}^{z} \tilde{p}_t^i log(\tilde{p}_p^i) \tag{9}$$

where z is the total number of comment instances; $\tilde{p}_p$ and $\tilde{p}_t$ represents the predictive category distribution and truly category distribution, respectively.

After the model training, we initialize the new SATD instance by the CodeBERT model to the embedding vector. Then we employ the feature extraction module to obtain the key feature representation, which including global feature encoder and key feature extraction. Finally, we output the category to which this SATD instance belongs.

4. Experimental setup

4.1. Dataset

To evaluate the performance of our model, we conduct experiments on a benchmark dataset proposed by [27], which collected source code from 10 open source projects, extracted 62,566 code comments from the source code of the 10 projects and classified these comments into 6 comment categories. The dataset included Apache Ant, JMeter, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JRuby, and SQuirrel, respectively. Table 1 shows the details of the dataset, including the release of the dataset, the number of code comments (#Comments), the number of code comments that contain defect debt (#Defect), the number of code comments that contain design debt (#Design), and the number of code comments that contain implementation debt (#Implementation). In particular, parentheses indicate the proportion of each type of SATD in the total number of comment instances.

From the Table 1, we can see that the comments that contain design debt are the highest percentage of total comments and the average percentage of total comments is 4.32%. The comments that contain defect debt average percentage of total comments is 0.75%. The comments that contain implementation debt average percentage of total comments is 1.21%. Based on the above analysis, we can see that SATD has a low percentage of code comments and the number of SATDs in each category is highly imbalanced.

4.2. Performance indicators

Our model aims to identify a category to which a SATD comment belongs. Indeed, it is a multi-classification task for text. In order to evaluate the performance of our model, we set up two sets of indicators, one set is general indicators which are widely used in SATD identification [16,17], and another set of indicators is special indicators for evaluating imbalance data in multi-classification tasks. In general indicators, we employ three indicators, i.e., Precision, Recall, and F-score, and their formulas are as follows,

$$\begin{aligned} Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \\ F-score &= \frac{2 \times Precision \times Recall}{Precision + Recall} \end{aligned} \tag{10}$$

where TP denotes the number of technical debt comments correctly predicted as a category, FP denotes the number of debt comments with other categories but predicted as that category, and FN denotes the number of debt comments with that category but predicted as other categories [55]. For the three metrics, the higher values of each metric represent the better performance of the model. The three metrics have been widely used in the fields of both software engineering [24,43,56,57] and machine learning [58–63].

In special indicators, we utilize three subset indicators as evaluation metrics, i.e., weighted indicators (Precision-weight, Recall-weight, and F-weight), indicators for each category (Precision-class, Recall-class, and F-class), and M- atthews correlation coefficient (MCC). For convenience, we denote Precision-weight, Recall-weight, and F-weight as weight-P, weight-R, and weight-F, respectively. We denote Precision-class, Recall-class, and F-class as Precision, Recall, and F-score in each type of SATD, respectively.

4.3. Statistic test

We first apply Wilcoxon signed-rank test [64] to analyze performance differences between our method and the baseline method at significance level $\alpha = 0.05$. Wilcoxon signed-rank test is a non-parametric test that compares the median of the sample with the median of the hypothesis [40,65]. Therefore, we employ the p-value of Wilcoxon signed-rank to analyze the significance of the differences between median results achieved by the two methods. If the p-value is lower than 0.05, it means that there is a statistically significant difference between the two methods. In addition, we calculate the effect size (Cliff's delta) to quantify the difference between the two methods. The value of Cliff's delta (δ) ranges from −1 to 1. In particular, there are three groups of differences: negligible ($0 < |\text{Cliff's } \delta| < 0.147$), small ($0.147 < |\text{Cliff's } \delta| < 0.33$), medium ($0.33 < |\text{Cliff's } \delta| < 0.474$), or large ($|\text{Cliff's } \delta| > 0.474$).

4.4. Parameter settings

In the JSD-GAN step, we set the epoch of adversarial networks as 500, the batch size as 64, learning rate as 0.01. In particular, the MLE epoch as 0. We use the CodeBERT model to initialize every word, which sets the dimension of the word embedding as 768. In the global feature encoder, we set the steps of encoder as 2, and the number of heads as 5. In key feature extraction, we set three filters with different window sizes [2,3,4]. We set the learning rate as 1e-4, the batch size as 128, and the epoch as 200. In addition, We take turns to choose in turn one of the ten projects as the test set and the others are treated as training set to verify the performance of our model. We have released the code scripts and benchmark dataset at https://figshare.com/articles/dataset/GCF-1_0/21746957 for reproducing our experiments.

4.5. Research questions

We set the following three research questions (RQs) in this work to evaluate our proposed model.

RQ1: Is our proposed model outperforming the state-of-the-art methods?

Motivation: Our study focus on detecting multi-types of SATDs tasks (i.e., defect debt, design debt, implementation debt). A recent study proposed machine learning techniques to classify SATD in three types and obtained state-of-the-art performance. Therefore, this research question investigates whether the GCF model performs better in multi-types of SATD tasks than the existing method.

RQ2: Is our model better than its variants?

Motivation: Our GCF model first introduces the data augmentation technique to generate synthetic instances for solving the imbalance issue. Then, we employ CodeBERT to initialize the embedding vectors for the input instances and propose a feature extraction module to learn representative features. This question investigates whether each component of our GCF model effectively improves the performance of multi-classification SATD.

RQ3: Does our model perform better than other methods of generating adversarial networks to balance samples?

Motivation: Our GCF model employs the JSD-GAN to generate synthetic instances to balance the original dataset, which obtains better performance in discrete sequence generative. This question investigates whether other mainstream GAN models are more practical to augment the data and relieve the imbalance issue, aiming to promote identification performance.

5. Experimental results

5.1. RQ1: Is our proposed model outperforming the state-of-the-art methods?

Methods: To answer this question, we compare our model with the state-of-the-art method XGBoost proposed in [19] to identify multi-types of SATDs on code comments that contain SATD. Firstly, the raw data was processed by five steps: filtering, change record deletion, tokenization, stop word deletion, and lemmatization. Then, the method utilized EDA to solve the problem of data imbalance, employed chi-square to make the feature selection of the data, applied the CountVectorizer technique to convert features into a feature vector representation, and used XGBoost to train the classifier for debt classification of SATD comments. XGBoost obtained state-of-the-art performance on the three debt classification tasks. In this RQ, we use general indicators to verify the effectiveness of our method for identifying SATD. In particular, we also use special indicators to show the effectiveness of our method for SATD imbalanced multi-classification tasks.

Results: Table 2 shows the results of our model and state-of-the-art method for Precision, Recall, and F-score, respectively. From this table, we can see that our model improves the average value over the XGBoost method by 10.41%, 26.69%, and 27.31% in terms of Precision, Recall, and F-score, respectively. In terms of Precision, 7 projects out of 10 projects obtain a better performance than the XGBoost method. The Precision of JMeter is the lowest performance in our model which is 0.4325. The Precision of EMF obtains the best performance in our model which is 0.8295. In terms of Recall, 9 projects in 10 projects obtain a better performance than the XGBoost model. The Recall of JMeter is the lowest performance in our model which is 0.5011. The Recall of EMF obtains the best performance in our model which is 0.6681. In terms of F-score, 9 projects in 10 projects obtain a better performance than the XGBoost model. The F-score of JMeter is the lowest performance in our model which is 0.4502. The F-score of EMF obtains the best performance in our model which is 0.7107. Table 3 presents the p-value of Wilcoxon signed-rank test and the Cliff's δ for three indicators. Noting that, in most of the projects, our GCF model is significantly superior to the XGBoost method with large effectiveness level in terms of all three indicators.

In addition, Table 4 shows the results of our model and XGBoost in terms of weight-P, weight-R, weight-F, and MCC. From this table, we can see that our model obtains the average value of 0.7840, 0.7654, 0.7606, and 0.4074 in terms of four indicators, respectively. Our model performs better weight-P on 9 out of 10 projects and improvements in the average value of 7.98% than the XGBoost method. Our model performs better weight-R on 6 out of 10 projects and improvements in the average value of 4.25% than the XGBoost method. Our model performs better weight-F on 9 out of 10 projects and improvements in the average value of 8.16% than the XGBoost method. Our model performs better MCC on 9 out of 10 projects and improvements in the average value of 55.61% than the XGBoost method.

Table 5 presents the results of our GCF model and XGBoost method on each SATD type in terms of Precision, Recall, and F-score. From this table, we can observe that our model obtains the average value of 0.8365, 0.8753, and 0.8455 on the type of design debt in terms of Precision, Recall, and F-score, respectively. Our model achieves improvement by 5.94%, −4.23%, and 0.68% than the XGBoost method in terms of three indicators. Our model obtains the average value of 0.5500, 0.4730, and 0.4714 on the type of implementation debt in terms of Precision, Recall, and F-score, respectively. Our model achieves improvement by 30.79%, 94.27%, and 83.80% than the XGBoost method in terms of three indicators. our model obtains the average value of 0.5241, 0.4137, and 0.4386 on the type of defect debt in terms of Precision, Recall, and F-score, respectively. Our model achieves improvement by 0.41%, 63.31%, and 41.32% than the XGBoost method in terms of three indicators. In particular, the average Recall of our model shows worse performance than XGBoost in the type of design debt. This is because the number of design debts is much more, leading to XGBoost can learn more knowledge. By contrast, our method performs better than XGBoost despite the few numbers of implementation and defect debt, which indicates that our method effectively handles the imbalance problem.

The XGBoost model has been compared with some mainstream methods for multi-types of SATD comments, and XGBoost outperforms these methods. Since our model performs better than XGBoost, we can conclude that our model can also outperform other mainstream methods for multi-types of SATD comments.

Answer to RQ1

Our proposed model performs better than the state-of-the-art method in terms of three indicators for detecting multi-types of SATD comments.

Table 2
The results of our model and XGBoost in terms of Precision, Recall, and F-score.

Project	Precision			Recall			F-score		
	XGBoost	GCF	Gains	XGBoost	GCF	Gains	XGBoost	GCF	Gains
Apache Ant	0.5933	0.7096	19.60%	0.4226	0.5869	38.88%	0.4397	0.6289	43.02%
JMeter	0.5024	0.4325	−13.91%	0.5124	0.5011	−2.20%	0.5051	0.4502	−10.87%
ArgoUML	0.5348	0.6672	24.75%	0.4137	0.5220	26.19%	0.4046	0.5474	35.31%
Columba	0.5757	0.6418	11.48%	0.5452	0.6493	19.11%	0.5570	0.6222	11.71%
EMF	0.6489	0.8295	27.83%	0.4149	0.6681	61.04%	0.4223	0.7107	68.31%
Hibernate	0.5966	0.5766	−3.36%	0.4547	0.5425	19.33%	0.4863	0.5516	13.42%
JEdit	0.5928	0.5972	0.75%	0.4270	0.5878	37.64%	0.4374	0.5766	31.85%
JFreeChart	0.5126	0.5069	−1.11%	0.5470	0.5858	7.09%	0.4836	0.5034	4.09%
JRuby	0.4991	0.6889	38.03%	0.3962	0.5703	43.97%	0.3745	0.5926	58.26%
SQuirrel	0.7183	0.7187	0.06%	0.5692	0.6597	15.89%	0.5783	0.6826	18.04%
Average	0.5774	0.6369	10.41%	0.4703	0.5874	26.69%	0.4689	0.5866	27.31%

Table 3
p-value and Cliff's Delta(δ) for GCF and XGBoost in terms of Precision, Recall, and F-score.

Project	Precision		Recall		F-score	
	p-value	δ	*p*-value	δ	*p*-value	δ
Apache Ant	9.77e−03	0.6(L)	9.77e−03	0.6(L)	1.95e−03	1.0(L)
JMeter	1.95e−03	−1.0(L)	1.95e−03	−1.0(L)	1.95e−03	−1.0(L)
ArgoUML	1.95e−03	1.0(L)	1.95e−03	1.0(L)	1.95e−03	1.0(L)
Columba	9.77e−03	0.76(L)	9.77e−03	0.76(L)	1.95e−03	0.76(L)
EMF	3.71e−02	0.68(L)	3.71e−02	0.68(L)	1.95e−03	1.0(L)
Hibernate	9.77e−03	−0.6(L)	9.77e−03	−0.6(L)	1.95e−03	1.0(L)
JEdit	4.32e−01	−0.28(S)	4.32e−01	−0.28(S)	1.95e−03	1.0(L)
JFreeChart	9.22e−01	−0.12(N)	9.22e−01	−0.12(N)	5.57e−01	−0.04(N)
JRuby	1.95e−03	1.0(L)	1.95e−03	1.0(L)	1.95e−03	1.0(L)
SQuirrel	9.22e−01	−0.04(N)	9.22e−01	−0.04(N)	1.95e−03	1.0(L)

Table 4
The results of our model and XGBoost in terms of weight-P, weight-R, and weight-F.

Project	weight-P		weight-R		weight-F		MCC	
	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF
Apache Ant	0.7434	**0.8129**	0.7868	**0.8297**	0.7923	**0.8137**	0.2475	**0.4631**
JMeter	**0.8383**	0.8321	**0.8323**	0.7621	**0.8349**	0.7918	**0.2564**	0.1982
ArgoUML	0.5836	**0.7156**	0.6155	**0.7102**	0.5389	**0.6769**	0.1735	**0.4236**
Columba	0.7024	**0.7547**	**0.7132**	0.7099	0.7063	**0.7123**	0.3509	**0.4387**
EMF	0.7246	**0.8464**	0.7745	**0.8490**	0.7014	**0.8291**	0.2171	**0.5621**
Hibernate	0.7119	**0.7315**	**0.7503**	0.7414	0.7136	**0.7335**	0.2330	**0.3250**
JEdit	0.7587	**0.7765**	**0.7858**	0.7739	0.7319	**0.7683**	0.2579	**0.3722**
JFreeChart	0.8621	**0.8806**	0.7192	**0.7808**	0.7612	**0.8167**	0.2720	**0.3476**
JRuby	0.5509	**0.6910**	0.5765	**0.6909**	0.5036	**0.6648**	0.1947	**0.4389**
SQuirrel	0.7843	**0.7983**	0.7873	**0.8057**	0.7478	**0.7986**	0.4151	**0.5047**
Average	0.7260	**0.7840**	0.7341	**0.7654**	0.7032	**0.7606**	0.2618	**0.4074**

Table 5
The each SATD type results of our model and XGBoost in terms of Precision, Recall, and F-score.

Project	Design debt						Implementation debt						Defect debt					
	Precision		Recall		F-score		Precision		Recall		F-score		Precision		Recall		F-score	
	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF	XGBoost	GCF
Apache Ant	0.8096	**0.8620**	0.9600	**0.9453**	0.8810	**0.9016**	0.3250	**0.5700**	0.1538	**0.3846**	0.1981	**0.4580**	0.6400	**0.6966**	0.1538	**0.4308**	0.2400	**0.5271**
JMeter	0.9122	**0.9172**	**0.9025**	0.8177	**0.9071**	0.8645	**0.2827**	0.1931	**0.3619**	0.3143	**0.3172**	0.2385	**0.3127**	0.1872	0.2727	**0.3714**	**0.2911**	0.2478
ArgoUML	0.6287	**0.7034**	0.9316	**0.9406**	0.7507	**0.8047**	0.5421	**0.8097**	0.1363	**0.4161**	0.2164	**0.5471**	0.4336	**0.4884**	0.1732	**0.2094**	0.2465	**0.2904**
Columba	0.7841	**0.8483**	**0.8222**	0.7365	**0.8026**	0.7365	**0.5595**	0.5491	0.5209	**0.6884**	0.5379	**0.5733**	0.3835	**0.5279**	0.2923	**0.5231**	0.3305	**0.5130**
EMF	0.7801	**0.8576**	**0.9821**	0.9667	0.8694	**0.9086**	0.4667	**0.7988**	0.0375	**0.3625**	0.0681	**0.4886**	0.7000	**0.8321**	0.2250	**0.6750**	0.3293	**0.7350**
Hibernate	0.7873	**0.8222**	**0.9251**	0.8541	**0.8492**	0.8378	0.3669	**0.4531**	0.1813	**0.4813**	0.2426	**0.4646**	**0.6381**	0.4544	0.2577	**0.2923**	**0.3671**	0.3523
JEdit	0.8084	**0.8587**	**0.9745**	0.8816	**0.8838**	0.8689	0.2841	**0.5204**	0.1857	**0.3674**	0.2233	**0.4147**	**0.6857**	0.4127	0.1209	**0.5143**	0.2050	**0.4463**
JFreeChart	0.9378	**0.9608**	**0.7522**	0.8152	0.8203	**0.8816**	**0.2270**	0.2232	0.5333	**0.6533**	0.2854	**0.3322**	**0.3730**	0.3366	**0.3556**	0.2889	**0.3452**	0.2965
JRuby	0.6376	**0.6853**	**0.9329**	0.8915	0.7560	**0.7748**	0.3691	**0.7306**	0.1400	**0.5540**	0.1817	**0.6290**	0.4906	**0.6508**	0.1155	**0.2655**	0.1856	**0.3741**
SQuirrel	0.8104	**0.8499**	**0.9569**	0.9043	0.8776	**0.8762**	**0.7818**	0.6517	0.1840	**0.5080**	0.2938	**0.5677**	0.5625	**0.6544**	**0.5667**	**0.5667**	0.5634	**0.6038**
Average	0.7896	**0.8365**	**0.9140**	0.8753	0.8398	**0.8455**	0.4205	**0.5500**	0.2435	**0.4730**	0.2565	**0.4714**	0.5220	**0.5241**	0.2533	**0.4137**	0.3104	**0.4386**

5.2. RQ2: Is our model better than its variants?

Methods: To answer this question, we explore the effectiveness of our model from different aspects. Specifically, we generate the following three variants for comparison by removing and substituting some components used in our model. In addition, we choose three representative methods to replace the CNN part in our model for SATD comments classification (i.e., BiLSTM, BiLSTM+attention, and GRU), aiming to explore how CNN impacts the performance of our model. As a result, we have the following six variants.

- NoGAN: This variant removes the JSD-GAN in our model, aiming to explore how JSD-GAN impacts the performance of our model.
- Glove: This variant utilizes the statical Glove technique [66] to replace the CodeBERT for initializing the word embedding, aiming at exploring how CodeBERT impacts the performance of our model.
- NoEncoder: This variant removes the global feature encoder in our model, exploring how the global feature encoder impacts the performance of our model.
- biLSTM (Zhang et al. [67]): Bi-directional Long Short-Term Memory (BiLSTM) is a combination of forward LSTM and backward LSTM. It is often used to model contextual information in natural language processing tasks. BiLSTM could better capture bidirectional semantic dependencies. It is widely used in text classification tasks.
- BiLSTM+attention (Zhou et al. [68]): This approach proposed a bi-directional long short-term memory network with attention mechanism for text classification task. The method used BiLSTM to encode the text feature sequence temporally and employed the attention mechanism to discover the weights of temporal text features for overall text identification at different moments. We abbreviate this method name as bi+attention.
- GRU (Chung et al. [69]): GRU is a variant of LSTM that solved the problems of long-term memory and gradient in backpropagation. GRU is commonly used methods in text classification.

Results: Table 6 shows the performance of our model and its 6 variants in terms of Precision. From this table, we can observe that the performance of our model is better than all variant models in 9 projects in terms of precision. The average Precision value of our model is 0.6369, which achieves improvements by 7.18%, 18.47%, 13.38%, 25.61%, 17.90%, and 24.71% compared with NoGAN, Glove, NoEncoder, BiLSTM, Bi+attention, and GRU, individually. Table 7 shows the performance of our model and its 6 variants in terms of Recall. From this table, we can observe that the performance of our model is better than all variant models in 9 projects in terms of Recall. The average Recall value of our model is 0.5874, which achieves improvements by 9.03%, 13.78%, 28.90%, 20.55%, 14.80%, and 16.07% compared with NoGAN, Glove, NoEncoder, BiLSTM, Bi+attention, and GRU, individually. Table 8 shows the performance of our model and its 6 variants in terms of F-score. From this table, we can observe that the performance of our model is better than all variant models in 8 projects in terms of F-score. The average F-score value of our model is 0.5866, which achieves improvements by 8.91%, 15.68%, 27.47%, 23.59%, 15.83%, and 19.75% compared with NoGAN, Glove, NoEncoder, BiLSTM, Bi+attention, and GRU, individually.

Compared with NoGAN, the JSD-GAN effectively solves the data imbalance problem and obtains outstanding performance in multi-types of SATD classification task. CodeBERT fuses the information between code snippets and natural language, and improves the performance compared to the Glove technique. Compared with NoEncoder, the global feature encoder enhances the representation of data features. In addition, compared with BiLSTM, Bi+attention, and GRU, the CNN module in our model also achieves better performance than the three baseline methods.

Answer to RQ2

Our model performs better than its variants, and all modules employed in our model have positive impact on its performance.

5.3. RQ3: Does our model perform better than other methods of generating adversarial networks to balance samples?

Methods: To answer this question, we choose four mainstream GAN methods to generate synthetic instances to solve the imbalance problem. We substitute the GAN module and make other components of our model the same. Below, we introduce these techniques.

- RelGAN (Weili et al. [70]): This approach introduced three improved modules to standard GAN. Firstly, the generator employed relational memory that made strong expressive power and modeling ability on long text. Secondly, on discrete data, the RelGAN was trained by the gumbel-softmax relaxation model, which made the model simpler. Thirdly, multi-layer word vector representation was used in the discriminator, which updated the generator towards more diversity.
- MaliGAN (Tong et al. [71]): This approach proposed the maximum likelihood augmented discrete generative adversarial network (MaliGAN). The method followed the discriminator of the standard GAN, and optimized the generator by using significant sampling to make the training process closer to the maximum likelihood (MLE) training of auto regressive model, resulting in more stability and minor gradient variance.
- SeqGAN (Lantao et al. [72]): This approach treated the sequence generation problem as a sequential decision process. In this approach, under the standard GAN model, the currently generated token was viewed as a state, the next generated token was viewed as an action, and a discriminator was used to evaluate the whole sequence to guide the generator learning. This method treated the generator as a random policy to solve the problem that the gradient was hard to pass to the generator. In the policy gradient, MCT search was used to approximate the value of the state–action pair.
- DGSAN (Ehsan et al. [73]): This approach proposed a domain-based GAN framework in which the gradient transfer problem was solved by considering the explicit distribution of generators (due to the advantage of finite discrete domains) and found a closed relationship between the following generator, the current generator and the current discriminator. The method unified the generators and discriminators in a single network.

Results: Table 9, 10, and 11 represent the performance of our model and the four comparative methods in terms of Precision, Recall, and F-score, respectively. From Table 9, we can see that, in terms of Precision, the average value by our model achieves improvement by 16.07%, 13.94%, 19.31%, and 21.97% compared with RelGAN, MaliGAN, SeqGAN, and DGSAN, individually. Our model obtains the best average Precision of 0.6369 and the average improvement by 17.82%. From Table 10, we can see that, in terms of Recall, the average value by our model achieves improvement by 11.29%, 9.98%, 15.01%, and 8.91% compared with RelGAN, MaliGAN, SeqGAN, and DGSAN, individually. Our model obtains the best average Recall of 0.5874 and an average improvement of 11.30%. From Table 11, we can see that, in terms of F-score, the average value by our model achieves improvement by 12.54%, 10.84%, 16.06%, and 14.80% compared with RelGAN, MaliGAN, SeqGAN, and DGSAN, individually. Our model obtains the best average F-score value of 0.5866 and achieves an average improvement of 13.56%. Moreover, our model achieves better performance in all three indicators for each project than the four baseline methods.

As the generated synthetic instances have some noise, only by choosing a suitable method can we improve the classification performance of model. The analysis in this section shows that the method we choose, JSD-GAN, has relatively less noise in the generated synthetic instances compared to other methods.

Answer to RQ3

Our model performs better than others with different GAN modules, and the JSD-GAN module in our model has a positive impact on the performance of multi-type SATD classification tasks.

Table 6
The detailed results for Precision in our model and six baseline methods.

Project	NoGAN	Glove	NoEncoder	BiLSTM	Bi+attention	GRU	GCF
Apache Ant	0.6860	0.6601	0.6222	0.5732	0.5065	0.5141	**0.7096**
JMeter	**0.4412**	0.4245	0.4122	0.3768	0.4175	0.4014	0.4325
ArgoUML	0.6381	0.6348	0.6557	0.5821	0.5973	0.5613	**0.6672**
Columba	0.5982	0.5848	0.5717	0.5127	0.5769	0.5370	**0.6418**
EMF	0.7467	0.5633	0.6012	0.5807	0.6749	0.6077	**0.8295**
Hibernate	0.5792	0.5357	0.5394	0.4619	0.4835	0.4670	**0.5766**
JEdit	0.5466	0.4720	0.5200	0.4988	0.5557	0.4974	**0.5972**
JFreeChart	0.4883	0.4314	0.3898	0.4131	0.4078	0.4026	**0.5069**
JRuby	0.5442	0.4655	0.6182	0.5476	0.5591	0.5760	**0.6889**
SQuirrel	0.6735	0.6041	0.6868	0.5236	0.6228	0.5425	**0.7187**
Average	0.5942	0.5376	0.5617	0.5071	0.5402	0.5107	**0.6369**

Table 7
The detailed results for Recall in our model and six baseline methods.

Project	NoGAN	Glove	NoEncoder	BiLSTM	Bi+attention	GRU	GCF
Apache Ant	0.5668	0.5468	0.4510	0.5085	0.4928	0.4921	**0.5869**
JMeter	0.4508	0.4567	0.4186	0.3887	0.4590	0.4400	**0.5011**
ArgoUML	0.5180	0.5202	0.4643	0.5032	0.5154	0.4975	**0.5220**
Columba	0.5553	0.5705	0.5160	0.5279	0.5576	0.5776	**0.6493**
EMF	0.5809	0.5140	0.4488	0.5351	0.5693	0.5806	**0.6681**
Hibernate	**0.5437**	0.5140	0.4530	0.4177	0.4724	0.4634	0.5425
JEdit	0.5270	0.4927	0.4251	0.4959	0.5532	0.4955	**0.5878**
JFreeChart	0.5604	0.4943	0.4324	0.4820	0.4585	0.4596	**0.5858**
JRuby	0.4815	0.4383	0.4470	0.5028	0.5105	0.5107	**0.5703**
SQuirrel	0.6030	0.6150	0.5008	0.5108	0.5280	0.5437	**0.6597**
Average	0.5387	0.5163	0.4557	0.4873	0.5117	0.5061	**0.5874**

Table 8
The detailed results for F-score in our model and six baseline methods.

Project	NoGAN	Glove	NoEncoder	BiLSTM	Bi+attention	GRU	GCF
Apache Ant	0.6003	0.5817	0.4744	0.5278	0.4937	0.4974	**0.6289**
JMeter	0.4323	0.4310	0.4128	0.3743	0.4228	0.3907	**0.4502**
ArgoUML	0.5405	0.5377	0.4627	0.5122	0.5266	0.5086	**0.5474**
Columba	0.5517	0.5723	0.5232	0.5030	0.5435	0.5292	**0.6222**
EMF	0.6111	0.5262	0.4753	0.5121	0.5835	0.5734	**0.7107**
Hibernate	0.5417	0.5214	0.4594	0.4225	0.4693	0.4638	**0.5516**
JEdit	0.5185	0.4634	0.4250	0.4855	0.5369	0.4729	**0.5766**
JFreeChart	0.4885	0.4243	0.4000	0.4253	0.4186	0.4084	**0.5034**
JRuby	0.4772	0.4097	0.4273	0.4868	0.5156	0.5207	**0.5926**
SQuirrel	0.6242	0.6031	0.5418	0.4966	0.5539	0.5334	**0.6826**
Average	0.5386	0.5071	0.4602	0.4746	0.5064	0.4898	**0.5866**

Table 9
The detailed results for Precision in our model and four GAN baseline methods.

Project	RelGAN	MaliGAN	SeqGAN	DGSAN	GCF
Apache Ant	0.6358	0.6342	0.6277	0.5734	**0.7096**
JMeter	0.4130	0.4159	0.4177	0.4000	**0.4325**
ArgoUML	0.5803	0.5763	0.5779	0.5621	**0.6672**
Columba	0.5404	0.5536	0.5305	0.5219	**0.6418**
EMF	0.6657	0.6775	0.6464	0.5431	**0.8295**
Hibernate	0.5320	0.5491	0.5396	0.5061	**0.5766**
JEdit	0.5588	0.5322	0.5113	0.5688	**0.5972**
JFreeChart	0.4170	0.4313	0.3583	0.4078	**0.5069**
JRuby	0.4728	0.5823	0.5089	0.5327	**0.6889**
SQuirrel	0.6716	0.6374	0.6199	0.6058	**0.7187**
Average	0.5487	0.5590	0.5338	0.5222	**0.6369**

Table 10
The detailed results for Recall in our model and four GAN baseline methods.

Project	RelGAN	MaliGAN	SeqGAN	DGSAN	GCF
Apache Ant	0.5838	0.5691	0.5796	0.5625	**0.5869**
JMeter	0.4518	0.4545	0.4463	0.4634	**0.5011**
ArgoUML	0.5149	0.5121	0.5091	**0.5284**	0.5220
Columba	0.5716	0.5581	0.5619	0.5763	**0.6493**
EMF	0.5552	0.5952	0.5470	0.5576	**0.6681**
Hibernate	0.5148	0.5116	0.5143	0.5141	**0.5425**
JEdit	0.5159	0.5112	0.4936	0.5481	**0.5878**
JFreeChart	0.4916	0.4642	0.3677	0.4800	**0.5858**
JRuby	0.4489	0.5226	0.5024	0.5299	**0.5703**
SQuirrel	0.6294	0.6422	0.5855	0.6332	**0.6597**
Average	0.5278	0.5341	0.5107	0.5393	**0.5874**

Table 11
The detailed results for F-score in our model and four GAN baseline methods.

Project	RelGAN	MaliGAN	SeqGAN	DGSAN	GCF
Apache Ant	0.6002	0.5857	0.5967	0.5556	**0.6289**
JMeter	0.4189	0.4224	0.4212	0.3990	**0.4502**
ArgoUML	0.5291	0.5234	0.5204	0.5346	**0.5474**
Columba	0.5458	0.5436	0.5253	0.5213	**0.6222**
EMF	0.5670	0.6119	0.5353	0.5330	**0.7107**
Hibernate	0.5179	0.5185	0.5154	0.5028	**0.5516**
JEdit	0.5312	0.5064	0.4941	0.5446	**0.5766**
JFreeChart	0.4197	0.4108	0.3545	0.3783	**0.5034**
JRuby	0.4368	0.5340	0.4983	0.5250	**0.5926**
SQuirrel	0.6457	0.6354	0.5930	0.6155	**0.6826**
Average	0.5212	0.5292	0.5054	0.5110	**0.5866**

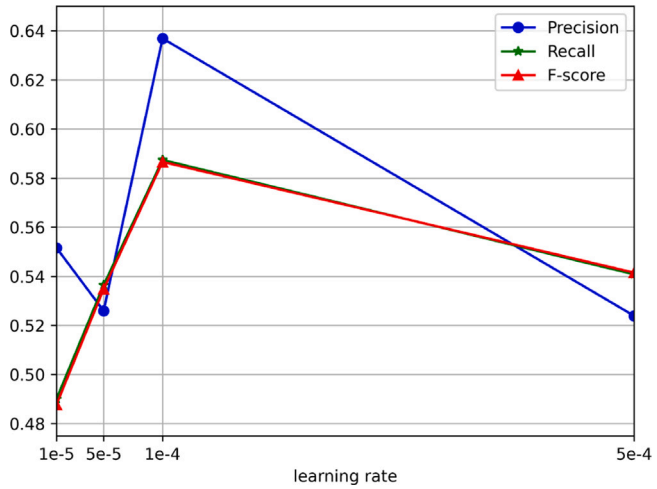


Fig. 4. The line chart for three indicators among different learning rates.

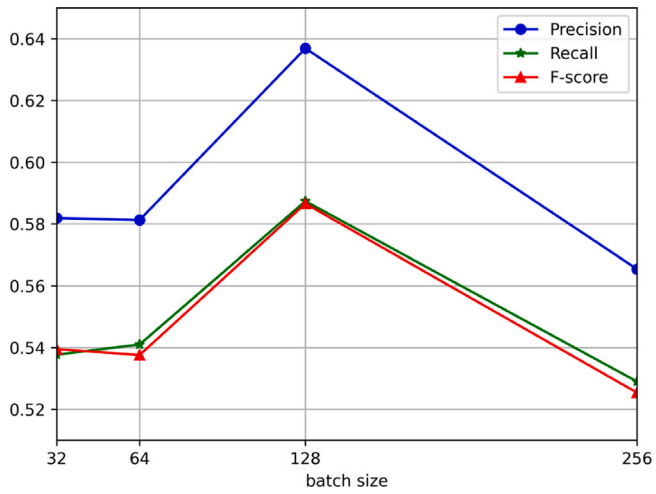


Fig. 5. The line chart for three indicators among different batch sizes.

Table 12
The detailed results for Precision in different proportions of generative samples.

Project	0.2	0.4	0.6	0.8	1.0
Apache Ant	0.6373	0.6443	0.6648	**0.7096**	0.5854
JMeter	0.4083	0.4056	0.4177	**0.4325**	0.4016
ArgoUML	0.6148	0.6416	0.6336	**0.6672**	0.6231
Columba	0.6107	0.5804	0.6087	**0.6418**	0.5684
EMF	0.7077	0.7151	0.7776	**0.8295**	0.7080
Hibernate	0.5434	0.5357	0.5548	**0.5766**	0.5127
JEdit	0.5343	0.5503	0.5637	**0.5972**	0.5395
JFreeChart	0.4490	0.4257	0.4208	**0.5069**	0.4175
JRuby	0.5100	0.5258	0.5776	**0.6889**	0.5437
SQuirrel	0.6994	0.6971	0.6691	**0.7187**	0.6483
Average	0.5715	0.5722	0.5889	**0.6369**	0.5548

6. Discussion

6.1. The impact of different parameter settings

In this section, we first discuss the impacts of hyper-parameters on the performance of our model and the impacts of the number of generative synthetic instances on the performance of our model. We investigate the hyper-parameters learning rate (LR) and batch size (BS) separately. Then, we investigate the number of generative synthetic instances (MT). We only modify the parameters that need to be discussed while ensuring that other parameters remain unchanged, and observe the impact of those parameters to our model.

The impact of LR. We empirically select four different LR settings from {1e-5, 5e-5, 1e-4, 5e-4} and conduct experiments with each set. Fig. 4 presents the results of each LR in terms of three indicators. From this figure, we can see that, LR that is too small or too large does not perform the best in our model. Therefore, we choose the LR as 1e-4 that obtains the better performance in terms of Precision, Recall, and F-score.

The impact of BS. We empirically select four different BS settings from {32, 64, 128, 256} and conduct experiments with each set. Fig. 5 presents the results of each BS in terms of three indicators. From this figure, we can see that, BS that is too small or too large does not perform the best in our model. Therefore, we choose the BS as 128 which obtains better performance in terms of Precision, Recall, and F-score.

The impact of MT. We define the number of generated synthetic instances as follows: first, we select the category that contains the maximum number of instances, then we subtract each of the other categories with this category, and the difference obtained from the subtraction is multiplied by 0.8 times which is the number of generated synthetic instances we choose for each category. To investigate the effect of different multiples (MT) on our model, we empirically select five different MT settings from {0.2, 0.4, 0.6, 0.8, 1.0} and conduct experiments with each set. Table 12, 13, 14 present the results of each MT in terms of Precision, Recall, and F-score. From those tables, we can see that the performance increases with the increase of MT before 0.8 and decreases after 0.8, which indicates that the much more generated synthetic instances have the more noisy and a negative impact on the classification performance of our model. Therefore, we choose the MT as 0.8 that obtains the better performance and the better average performance in terms of Precision, Recall, and F-score.

Due to different data labels and distributions, there are different parameter choices. Therefore, our parameters provide basic guidance.

6.2. Error analysis

In this section, we analyze the error results of our GCF method. We choose examples from JMeter because it obtains the worse performance. We use design debt as an example and discuss two types of errors. The first scenario is that the original instance is design debt, and our model incorrectly identifies it as another debt. The second scenario is that the original instance is another debt category, and our model incorrectly identifies it as design debt. Fig. 6 shows examples of our model incorrect classification. *Example 1* represents that the original instance is a design debt. Our model identifies it as a defect debt. *Example 2* represents that the original instance is an implementation debt, and our model identifies it as design debt. *Example 3* represents

Table 13
The detailed results for Recall in different proportions of generative samples.

Project	0.2	0.4	0.6	0.8	1.0
Apache Ant	0.5794	0.5409	0.5829	**0.5869**	0.5441
JMeter	0.4404	0.4612	0.4728	**0.5011**	0.4554
ArgoUML	0.5192	0.5178	0.5261	0.5220	**0.5271**
Columba	0.5812	0.5703	0.5816	**0.6493**	0.5857
EMF	0.5878	0.6043	0.5988	**0.6681**	0.5952
Hibernate	0.5154	0.5029	0.5131	**0.5425**	0.5081
JEdit	0.5355	0.5383	0.5378	**0.5878**	0.5395
JFreeChart	0.4490	0.4684	0.4883	**0.5858**	0.4564
JRuby	0.4740	0.4650	0.5080	**0.5703**	0.4855
SQuirrel	0.6192	0.6133	0.6306	**0.6597**	0.6218
Average	0.5301	0.5282	0.5440	**0.5874**	0.5319

Table 14
The detailed results for F-score in different proportions of generative samples.

Project	0.2	0.4	0.6	0.8	1.0
Apache Ant	0.5971	0.5751	0.6070	**0.6289**	0.5457
JMeter	0.4133	0.4164	0.4264	**0.4502**	0.4124
ArgoUML	0.5388	0.5405	**0.5483**	0.5474	0.5471
Columba	0.5722	0.5598	0.5795	**0.6222**	0.5614
EMF	0.6180	0.6086	0.6391	**0.7107**	0.6211
Hibernate	0.5203	0.5141	0.5072	**0.5516**	0.5058
JEdit	0.5274	0.5386	0.5352	**0.5766**	0.5318
JFreeChart	0.4443	0.4136	0.4088	**0.5034**	0.4084
JRuby	0.4592	0.4625	0.5177	**0.5926**	0.4885
SQuirrel	0.6424	0.6353	0.6306	**0.6826**	0.6291
Average	0.5333	0.5265	0.5400	**0.5866**	0.5251

that the original instance is a defect debt, and our model identifies it as design debt.

For *Example 1*, as the expression does not contain the semantic meaning that it is obviously due to design, it is more doubtful whether the setup is the correct setup and tends to be a defect. Hence, our model identifies it as a defect debt. Moreover, we invite three developers with more than 5 years of working experience. We first give them the relevant background knowledge, and then let them identify this comment. Two of them identify the comment as a defect debt and one as design debt. So there are also some classifications in the dataset that are not necessarily accurate.

For *Example 2*, the sentence is a long sentence while the key feature is in the last, and the preceding sentences tend to be design debt. Nevertheless, we intercept the long sentence because the number of short sentences far exceeds the number of long sentences in the whole data set, and long sentences result in excessively redundant information during word embedding, which can lead to unnecessary program consumption. So in this sentence, we remove the critical feature that causes the judgment failure. Thus, in the later work, we need to consider improving the classification performance of long sentences by intercepting long sentences without compromising the features of long sentences.

For *Example 3*, the first sentence has no explicit debt tendency, and the second sentence looks like design debt. However, the first sentence combines with the second sentence, which tends to the type of defect debt. Therefore, in future work, our model needs to be improved for contextual linkage to perform contextual linkage identification better.

6.3. Implications

Our study provides implications for subsequent research on multi-types of SATDs.

Our model achieved better performance on the average values of 0.6369, 0.5874, and 0.5866 in terms of Precision, Recall, and F-score, respectively. Based on our results, it can be traced back to relevant locations in the source code through code comments which contain debt. Therefore, developers can spend less effort locating the corresponding source code and fixing errors [19]. Based on our findings, developers can train a cross-project model or reuse our already-trained model to handle multi-type SATD identification in extended data. In particular, our model performs better on few number data, so we recommend our model for multi-type SATD prediction when there is only a few data on hand.

For researchers, identifying multi-types of SATDs can better analyze the distribution of debt and investigate the causes of debt [74]. Moreover, it helps to analyze the impact of SATD on software quality [75], the understanding of the action, the motivation for deleting SATD in software systems [76,77], and the retention time of SATD [8]. In addition, our work helps to study the interest paid on SATD [78].

7. Threats to validity

7.1. Threats to internal validity

Threats to internal validity derive from programming errors during the experiments and personal bias related to the label in the dataset. Our model is implemented based on Pytorch and third-party libraries to avoid programming errors. All baseline methods are also implemented based on PyTorch and third-party libraries, and we carefully modify the codes provided by previous studies to satisfy our needs. To reduce personal bias in the manually labeled dataset, we use the dataset by [27] with a high inter-rater agreement (Cohen's Kappa coefficient of 0.81). Moreover, hyper-parameter tuning poses a threat to internal validity. To reduce the threat, we fine-tune the learning rate of {1e-5, 5e-5, 1e-4, 5e-4} and the batch size of {32, 64, 128, 256}, in which we choose the best parameter settings, i.e., the learning rate of 1e-4 and batch size of 128 for our experiments.

7.2. Threats to external validity

Threats to external validity are focused on the generalizability of our model. We study our experiments on a publicly reliable dataset by [27]. The dataset contains 10 open source projects with 62,566 code comments. These 10 open source projects are characterized by different functions, different contributors, and the different number of comments. Moreover, it is a public dataset, which can help future researchers replicate our results. We realized that it would be better to validate our model with other project data from more diverse domains, and we left the exploration for future work. In addition, we selected the state-of-the-art SATD three classification model XGBoost and seven deep learning-based methods as baseline methods that have achieved satisfactory performance in previous studies.

7.3. Threats to construct validity

Threats to construct validity are related to the employed performance indicators and applicability. In this work, we use three widely used performance indicators, namely, Precision, Recall, and F-score, to evaluate the performance of our model. In addition, we employ a set of specific indicators, including Precision-weight, Recall-weight, F-weight, MCC, and each SATD type performance (i.e., Precision, Recall, and F-score), to evaluate the performance of our model on imbalanced data. We adopt Wilcoxon signed-rank test and Cliff's delta for analysis of method pairs, which makes our evaluation more convincing. Another threat to construct validity is related to the suitability of our model. Our work aims to determine which SATD category a code comment belongs to, intending to find the type of SATD from code comments to satisfy the need to locate different debts for developers. The experimental results demonstrate the superiority of the model and show that it applies to such tasks.

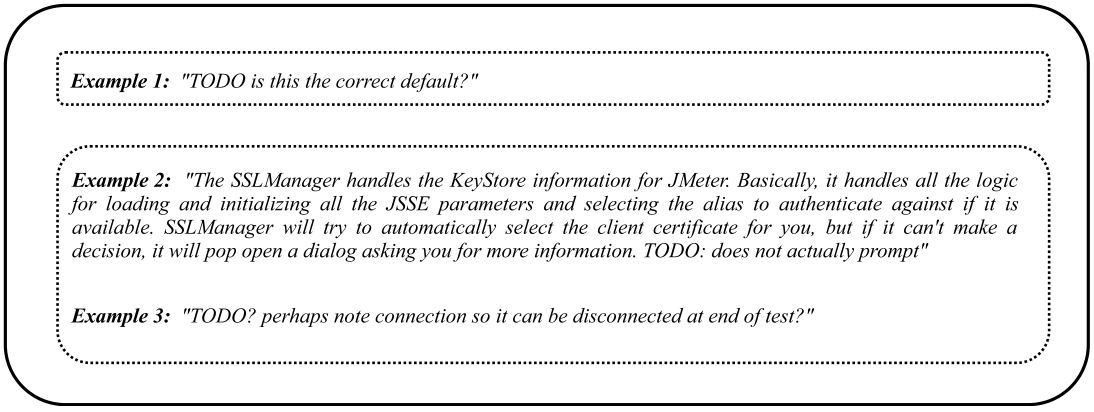


Fig. 6. The error predicted instances in JMeter.

On the other hand, there is also related research on the relationship between debt interest and repayment costs [78]. Different from such work, we input code comments into our trained model to automatically detect a specific SATD type. The cost of this process is negligible compared to the interest brought by the debt itself. Of course, repayment cost and interest are very important research. In the future, we will conduct in-depth related research on the combination of repayment cost and debt interest [79] for identifying debt and deleting debt.

8. Conclusion

In this work, we proposed a new GCF model to determine which type of technical debt a comment containing SATD belongs to. Specifically, GCF first uses the JSD-GAN model to address the problem of unimpressive classification performance due to unbalanced datasets. The CodeBERT model, fusing information in code snippets and natural language, was then employed to convert the instances into the embedding vector. A feature extraction module was also designed to take the embedding vector as input and conduct global feature fusion and extraction of the embedding vector through the global feature encoder and key feature extraction. We conducted comprehensive experiments on the public dataset using three performance indicators, and the results showed that our proposed GCF model outperformed the baseline methods including the state-of-the-art. In the future, we plan to collect more real SATD comments to validate the generalization ability of our model.

CRediT authorship contribution statement

Jiaojiao Yu: Writing – original draft, Methodology, Data curation. **Xu Zhou:** Methodology, Software, Visualization. **Xiao Liu:** Conceptualization, Writing – review & editing. **Jin Liu:** Supervision, Project administration. **Zhiwen Xie:** Formal analysis. **Kunsong Zhao:** Writing – review & editing.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2023.107190. Jin Liu reports financial support was provided by Wuhan University.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grants (No. 61972290).

References

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al., Managing technical debt in software-reliant systems, in: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, 2010, pp. 47–52.

[2] Z. Codabux, B. Williams, Managing technical debt: An industrial case study, in: 2013 4th International Workshop on Managing Technical Debt, MTD, IEEE, 2013, pp. 8–15.

[3] S. Freire, N. Rios, B. Gutierrez, D. Torres, M. Mendonça, C. Izurieta, C. Seaman, R.O. Spínola, Surveying software practitioners on technical debt payment practices and reasons for not paying off debt items, in: Proceedings of the Evaluation and Assessment in Software Engineering, 2020, pp. 210–219.

[4] A. Potdar, E. Shihab, An exploratory study on self-admitted technical debt, in: 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2014, pp. 91–100.

[5] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, The financial aspect of managing technical debt: A systematic literature review, Inf. Softw. Technol. 64 (2015) 52–73.

[6] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, in: Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 315–326.

[7] Y. Li, M. Soliman, P. Avgeriou, L. Somers, Self-admitted technical debt in the embedded systems industry: An exploratory case study, 2022, arXiv preprint arXiv:2205.13872.

[8] E.d.S. Maldonado, R. Abdalkareem, E. Shihab, A. Serebrenik, An empirical study on the removal of self-admitted technical debt, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2017, pp. 238–248.

[9] J. Xuan, Y. Hu, H. Jiang, Debt-prone bugs: technical debt in software maintenance, 2017, arXiv preprint arXiv:1704.04766.

[10] Q. Huang, E. Shihab, X. Xia, D. Lo, S. Li, Identifying self-admitted technical debt in open source projects using text mining, Empir. Softw. Eng. 23 (1) (2018) 418–451.

[11] N.S. Alves, T.S. Mendes, M.G. de Mendonça, R.O. Spínola, F. Shull, C. Seaman, Identification and management of technical debt: A systematic mapping study, Inf. Softw. Technol. 70 (2016) 100–121.

[12] H. Tu, T. Menzies, DebtFree: minimizing labeling cost in self-admitted technical debt identification using semi-supervised learning, Empir. Softw. Eng. 27 (4) (2022) 1–37.

[13] L. Huang, D. Ma, S. Li, X. Zhang, H. Wang, Text level graph neural network for text classification, 2019, arXiv preprint arXiv:1910.02356.

[14] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, Association for Computational Linguistics (ACL), 2014, pp. 1746–1751.

[15] S. Lai, L. Xu, K. Liu, J. Zhao, Recurrent convolutional neural networks for text classification, in: Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.

[16] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, J. Grundy, Neural network-based detection of self-admitted technical debt: From performance to explainability, ACM Trans. Softw. Eng. Methodol. 28 (3) (2019) 1–45.

[17] X. Wang, J. Liu, L. Li, X. Chen, X. Liu, H. Wu, Detecting and explaining self-admitted technical debts with attention-based neural networks, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 871–882.

[18] M.E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, L. Zettlemoyer, Deep contextualized word representations, in: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, Volume 1 (Long Papers), Association for Computational Linguistics, 2018, pp. 2227–2237.

[19] X. Chen, D. Yu, X. Fan, L. Wang, J. Chen, Multiclass classification for self-admitted technical debt based on xgboost, IEEE Trans. Reliab. (2021).

[20] G. Sierra, E. Shihab, Y. Kamei, A survey of self-admitted technical debt, J. Syst. Softw. 152 (2019) 70–82.

[21] B. Bansal, S. Srivastava, Sentiment classification of online consumer reviews using word vector representations, Procedia Comput. Sci. 132 (2018) 1147–1153.

[22] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: International Conference on Machine Learning, ICML, PMLR, 2014, pp. 1188–1196.

[23] K. Torkkola, Discriminative features for text document classification, Formal Pattern Anal. Appl. 6 (4) (2004) 301–308.

[24] J. Yu, K. Zhao, J. Liu, X. Liu, Z. Xu, X. Wang, Exploiting gated graph neural network for detecting and explaining self-admitted technical debts, J. Syst. Softw. 187 (2022) 111219.

[25] Z. Li, T. Xia, X. Lou, K. Xu, S. Wang, J. Xiao, Adversarial discrete sequence generation without explicit neuralnetworks as discriminators, in: The 22nd International Conference on Artificial Intelligence and Statistics, PMLR, 2019, pp. 3089–3098.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.

[27] E. da Silva Maldonado, E. Shihab, N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, IEEE Trans. Softw. Eng. 43 (11) (2017) 1044–1062.

[28] M.A. de Freitas Farias, M.G. de Mendonça Neto, M. Kalinowski, R.O. Spínola, Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary, Inf. Softw. Technol. 121 (2020) 106270.

[29] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study, ACM Trans. Softw. Eng. Methodol. 30 (4) (2021) 1–56.

[30] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, S. Li, SATD detector: A text-mining-based self-admitted technical debt detection tool, in: Proceedings of the 40th ACM/IEEE International Conference on Software Engineering, vol. 3, ICSE, 2013, pp. 9–12.

[31] Z. Yu, F.M. Fahid, H. Tu, T. Menzies, Identifying self-admitted technical debts with jitterbug: A two-step approach, IEEE Trans. Softw. Eng. (2020).

[32] D. Yu, L. Wang, X. Chen, J. Chen, Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt, Front. Comput. Sci. 15 (4) (2021) 1–12.

[33] S. Wattanakriengkrai, R. Maipradit, H. Hata, M. Choetkiertikul, T. Sunetnanta, K. Matsumoto, Identifying design and requirement self-admitted technical debt using n-gram idf, in: 2018 9th International Workshop on Empirical Software Engineering in Practice, IWESEP, IEEE, 2018, pp. 7–12.

[34] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, X. Yang, Automating change-level self-admitted technical debt determination, IEEE Trans. Softw. Eng. 45 (12) (2018) 1211–1229.

[35] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, Adv. Neural Inf. Process. Syst. 27 (2014).

[36] M. Arjovsky, S. Chintala, L. Bottou, Wasserstein generative adversarial networks, in: International Conference on Machine Learning, PMLR, 2017, pp. 214–223.

[37] C. Wang, C. Xu, X. Yao, D. Tao, Evolutionary generative adversarial networks, IEEE Trans. Evol. Comput. 23 (6) (2019) 921–934.

[38] J. Guo, S. Lu, H. Cai, W. Zhang, Y. Yu, J. Wang, Long text generation via adversarial training with leaked information, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, no. 1, 2018.

[39] X. Yu, J. Keung, Y. Xiao, S. Feng, F. Li, H. Dai, Predicting the precise number of software defects: Are we there yet? Inf. Softw. Technol. 146 (2022) 106847.

[40] X. Yu, J. Liu, J.W. Keung, Q. Li, K.E. Bennin, Z. Xu, J. Wang, X. Cui, Improving ranking-oriented defect prediction using a cost-sensitive ranking SVM, IEEE Trans. Reliab. 69 (1) (2019) 139–153.

[41] S. Feng, J. Keung, X. Yu, Y. Xiao, M. Zhang, Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction, Inf. Softw. Technol. 139 (2021) 106662.

[42] Y. Zhen, J.W. Keung, Y. Xiao, X. Yan, J. Zhi, J. Zhang, On the significance of category prediction for code-comment synchronization, ACM Trans. Softw. Eng. Methodol. (2022).

[43] X. Ma, J. Keung, Z. Yang, X. Yu, Y. Li, H. Zhang, CASMS: Combining clustering with attention semantic model for identifying security bug reports, Inf. Softw. Technol. 147 (2022) 106906.

[44] J.W. Wei, Z. Kai, EDA: easy data augmentation techniques for boosting performance on text classification tasks, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP, Association for Computational Linguistics (ACL), 2019, pp. 6381–6387.

[45] K. Akbar, R. Leonardo, P. Andrea, AEDA: an easier data augmentation technique for text classification, in: Findings of the Association for Computational Linguistics: EMNLP, Association for Computational Linguistics (ACL), 2021, pp. 2748–2754.

[46] Y. Cao, X. Wan, Divgan: Towards diverse paraphrase generation via diversified generative adversarial network, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 2411–2421.

[47] L. Liu, Y. Lu, M. Yang, Q. Qu, J. Zhu, H. Li, Generative adversarial network for abstractive text summarization, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, no. 1, 2018.

[48] J. Zhao, Z. Zhan, T. Li, R. Li, C. Hu, S. Wang, Y. Zhang, Generative adversarial network for table-to-text generation, Neurocomputing 452 (2021) 28–36.

[49] T. Karras, S. Laine, T. Aila, A style-based generator architecture for generative adversarial networks, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 4401–4410.

[50] H. Zhang, I. Goodfellow, D. Metaxas, A. Odena, Self-attention generative adversarial networks, in: International Conference on Machine Learning, ICML, PMLR, 2019, pp. 7354–7363.

[51] X. Gao, F. Deng, X. Yue, Data augmentation in fault diagnosis based on the Wasserstein generative adversarial network with gradient penalty, Neurocomputing 396 (2020) 487–494.

[52] M. Zheng, T. Li, R. Zhu, Y. Tang, M. Tang, L. Lin, Z. Ma, Conditional Wasserstein generative adversarial network-gradient penalty-based approach to alleviating imbalanced data classification, Inform. Sci. 512 (2020) 1009–1023.

[53] X. Zhou, D. Han, D. Lo, Assessing generalizability of CodeBERT, in: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2021, pp. 425–436.

[54] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

[55] Z. Xu, L. Li, M. Yan, J. Liu, X. Luo, J. Grundy, Y. Zhang, X. Zhang, A comprehensive comparative study of clustering-based unsupervised defect prediction models, J. Syst. Softw. 172 (2021) 110862.

[56] Z. Xu, K. Zhao, M. Yan, P. Yuan, L. Xu, Y. Lei, X. Zhang, Imbalanced metric learning for crashing fault residence prediction, J. Syst. Softw. 170 (2020) 110763.

[57] K. Zhao, J. Liu, Z. Xu, X. Liu, L. Xue, Z. Xie, Y. Zhou, X. Wang, Graph4Web: A relation-aware graph attention network for web service classification, J. Syst. Softw. 190 (2022) 111324.

[58] Y. Chen, S. Xiong, L. Mou, X.X. Zhu, Deep quadruple-based hashing for remote sensing image-sound retrieval, IEEE Trans. Geosci. Remote Sens. 60 (2022) 1–14.

[59] C. He, J. Wu, Q. Zhang, Proximity-aware research leadership recommendation in research collaboration via deep neural networks, J. Assoc. Inf. Sci. Technol. 73 (1) (2022) 70–89.

[60] Y. Chen, X. Lu, S. Wang, Deep cross-modal image–voice retrieval in remote sensing, IEEE Trans. Geosci. Remote Sens. 58 (10) (2020) 7049–7061.

[61] Z. Yang, J. Keung, M.A. Kabir, X. Yu, Y. Tang, M. Zhang, S. Feng, AComNN: Attention enhanced Compound Neural Network for financial time-series forecasting with cross-regional features, Appl. Soft Comput. 111 (2021) 107649.

[62] C. He, J. Wu, Q. Zhang, Characterizing research leadership on geographically weighted collaboration network, Scientometrics 126 (5) (2021) 4005–4037.

[63] Y. Chen, X. Lu, X. Li, Supervised deep hashing with a joint deep network, Pattern Recognit. 105 (2020) 107368.

[64] F. Wilcoxon, Individual comparisons by ranking methods, in: Breakthroughs in Statistics, Springer, 1992, pp. 196–202.

[65] K. Zhao, Z. Xu, T. Zhang, Y. Tang, M. Yan, Simplified deep forest model based just-in-time defect prediction for android mobile apps, IEEE Trans. Reliab. 70 (2) (2021) 848–859.

[66] J. Pennington, R. Socher, C.D. Manning, Glove: Global vectors for word representation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, 2014, pp. 1532–1543.

[67] S. Zhang, D. Zheng, X. Hu, M. Yang, Bidirectional long short-term memory networks for relation classification, in: Proceedings of the 29th Pacific Asia Conference on Language, Information and Computation, 2015, pp. 73–78.

[68] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, B. Xu, Attention-based bidirectional long short-term memory networks for relation classification, in: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2016, pp. 207–212.

[69] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014, arXiv preprint arXiv:1412.3555.

[70] N. Weili, N. Nina, P. Ankit, RelGAN: Relational generative adversarial networks for text generation, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, la, USA, May 6–9, 2019, OpenReview.net, 2019.

[71] C. Tong, L. Yanran, Z. Ruixiang, H. R. Devon, L. Wenjie, B. Yangqiu, Maximum-likelihood augmented discrete generative adversarial networks, 2017, CoRR arXiv:1702.07983.

[72] Y. Lantao, Z. Weinan, W. Jun, Y. Yong, SeqGAN: Sequence generative adversarial nets with policy gradient, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence,February 4–9, 2017, San Francisco, California, USA, AAAI Press, 2017, pp. 2852–2858.

[73] M. Ehsan, A. Danial, B. Mahdieh Soleymani, DGSAN: Discrete generative self-adversarial network, Neurocomputing 448 (2021) 364–379.

[74] N.S. Alves, L.F. Ribeiro, V. Caires, T.S. Mendes, R.O. Spínola, Towards an ontology of terms on technical debt, in: 2014 Sixth International Workshop on Managing Technical Debt, IEEE, 2014, pp. 1–7.

[75] Y. Miyake, S. Amasaki, H. Aman, T. Yokogawa, A replicated study on relationship between code quality and method comments, in: Applied Computing and Information Technology, Springer, 2017, pp. 17–30.

[76] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. Di Penta, A. Zaidman, Continuous delivery practices in a large financial organization, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2016, pp. 519–528.

[77] F. Zampetti, A. Serebrenik, M. Di Penta, Was self-admitted technical debt removal a real removal? An in-depth perspective, in: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories, MSR, IEEE, 2018, pp. 526–536.

[78] Y. Kamei, E.d.S. Maldonado, E. Shihab, N. Ubayashi, Using analytics to quantify interest of self-admitted technical debt, in: QuASoQ/TDA@ APSEC, 2016, pp. 68–71.

[79] G. Deshpande, G. Ruhe, Beyond accuracy: Roi-driven data analytics of empirical data, in: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, 2020, pp. 1–6.