The Journal of Systems & Software 187 (2022) 111219

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Exploiting gated graph neural network for detecting and explaining self-admitted technical debts[☆]

Jiaojiao Yu[a], Kunsong Zhao[a], Jin Liu[a,*], Xiao Liu[b], Zhou Xu[c], Xin Wang[a]

[a] School of Computer Science, Wuhan University, Wuhan, China
[b] School of Information Technology, Deakin University, Geelong, Australia
[c] School of Big Data and Software Engineering, Chongqing University, Chongqing, China

ARTICLE INFO

Article history:
Received 25 May 2021
Received in revised form 19 December 2021
Accepted 5 January 2022
Available online 10 January 2022

Keywords:
Technical debt
Self-admitted technical debt
Gated graph neural network
Attention mechanism

ABSTRACT

Self-admitted technical debt (SATD) refers to a specific type of technical debt that is introduced intentionally in the software development and maintenance processes. SATD enables practitioners to take some temporary solutions instead of making comprehensive decisions, which will lead to the high complexity of the software. However, most existing studies relied on manual methods for detecting SATDs. A recent study proposed a method HATD that used a hybrid attention-based method to automatically detect SATDs and it achieved the state-of-the-art performance. However, HATD mainly focused on the locality of the comment instances and lacked of the relationship between long-distance and discontinuous comment instances. To address such an issue, in this work, we propose a novel approach named GGSATD. Specifically, GGSATD first builds the graph for comment instances and then employs the gated graph neural network to iteratively update node representation. The global representation can be obtained by the soft attention mechanism and pooling operation. Experiments on 10 projects show that our GGSATD method obtains promising performance against five baseline methods in both within-project and cross-project scenarios. Extended experiments on seven real-world projects illustrate the effectiveness of our GGSATD method.

1. Introduction

Technical debt was proposed by Cunningham which was described as "not quite right code" (Cunningham, 1992). It is a metaphor that developers have to make compromises and decisions about shortcuts or workarounds to satisfy the needs of current software development goals. Due to some irresistible factors, such as fast delivery time and shortened budget, which make current goals too hard to reach, developers need to take sub-optimal measures to meet them (Lim et al., 2012). If developers constantly use sub-optimal solutions to reach current goals, the accumulation of such behaviors will generate technical debts during the development process. Technical debts exist in software projects commonly. For example, a developer chooses a technical framework to implement a function in a software module but it is obsoleted in the later version, meanwhile a test engineer just completes simple functional test in the module but the complexity test and stress test are forgotten. Those technical debts are intentionally or unintentionally introduced to software projects and there are no traceable materials to locate them in the future. As software versions and maintainers change, these unsolved technical debts will never be traced and remain in the software program forever. To alleviate this issue, Potdar and Shihab (Potdar and Shihab, 2014) first proposed the conception of self-admitted technical debts (SATDs), which is an intentionally introduced technical debts recorded by code comments.[1]

Technical debts need to repay interest (Tom et al., 2013; Wehaibi et al., 2016; Zazworka et al., 2013), which is the increased costs of manpower and resources for software maintenance in later. Thus, both intentionally and unintentionally technical debts can produce negative impacts on software maintenance in the long run (Izurieta et al., 2017; Li et al., 2015a; Point and from Dagstuhl). The impact of technical debts is exponential, which means that developers need to spend more time relieving them.[2]

The persistent impact of technical debts leads to a slowdown in development progress and a reduction in productivity (Ampatzoglou et al., 2015; Bavota and Russo, 2016; Kruchten et al., 2012). However, it is ubiquitous and inevitable in the process of software development (Xuan et al., 2017; Foucault et al., 2018). Furthermore, both Wehaibi et al. (2016), Miyake et al. (2017)

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

1 We also call this code comment instance in this work.
2 https://medium.com/serious-scrum/the-hidden-cost-of-technical-debt-1963b958e5ed.

mentioned that SATDs would have a long-term impact on the software quality. Therefore, how to automatically identify technical debts for software quality assurance and reduce software maintenance costs is a hot research topic in software engineering area (Li et al., 2015b; Marinescu, 2004; Zampetti et al., 2017). There are many types of technical debts and they are difficult to automatically identify and repair (Sierra et al., 2019). To solve these issues, previous studies concentrated on automatic identification of SATDs due to that the SATDs distinctly and widespreadly exist in the form of code comments, which is easier to be traced compared to other types of debts. Specifically, Potdar and Shihab (2014) summarized 62 identification patterns manually to identify SATDs in source code comments of Java projects. Mensah et al. (2016) proposed to use text mining based methods to identify SATDs. Yan et al. (2018) proposed a method to detect change-level SATDs using 25 software change features. Flisar et al. (Flisar and Podgorelec, 2019) trained a word embedding model to enhance the original feature set and used the enhanced feature set for SATD classification.

Nevertheless, the above-mentioned studies for detecting SATDs have two shortcomings (Liu et al.). First, for pattern based methods, they rely on manual modes and cannot identify SATDs automatically. Second, for machine learning based methods, they focus on the locality of the comment instances and lack of the relationship between long-distance and discontinuous comment instances (Ma et al., 2021; Mittal et al., 2021). To solve these problems, we propose a deep learning based method, namely GGSATD that applies the gated graph neural network with the inductiveness, to detect SATDs from source code comments (Zhang et al., 2020). Instead of treating the code comment instance sequence as the model input directly, our method converts the comment instance into the graph with co-occurrence relationship. More specifically, first, we use the code comment instance to build the graph, then we learn the features through the gated graph neural network (GGNN) that makes the features of nodes in each graph can be learnt more effectively. After the feature learning finished, the node representation the model learnt is sent into two multilevel perceptrons to obtain high-level feature representation followed by a softmax layer for SATD detection.

We conduct the experiment on a benchmark dataset that includes 10 software projects released by a previous work (Maldonado et al., 2017) with three performance indicators. Our experiment results show that, in within-project scenario, the mean values of Precision, Recall, and F1-score obtained by our method are 0.916, 0.884, and 0.895, respectively. Compared to the five baseline methods, the F1-score improves from 8.75% to 38.54%. In cross-project scenario, the mean values of Precision, Recall, and F1-score obtained by our method are 0.879, 0.849, and 0.862, respectively. Compared to the five baseline methods, the F1-score improves from 7.75% to 36.83%. To further verify the generation of our model, we collect seven new software projects with 152,569 comments in real world. In the seven extended projects, our method also achieves better prediction performance than the five baseline methods, i.e., Precision of 0.899, Recall of 0.950, and F1-score of 0.921.

The main contributions of this paper are summarized as follows:

- We propose a novel method called GGSATD to detect SATD automatically. To the best of our knowledge, we are the first to introduce the gated graph neural network for this task.
- GGSATD incorporates the soft attention and pooling mechanisms, which has the potential to learn and update the embeddings of nodes and generate better graph level representation.
- Comprehensive experiments on 10 open-source software projects show that our GGSATD method performs better than five baseline methods with three performance indicators.

The remainder of the paper is organized as follows. Section 2 introduces the relevant background knowledge. Section 3 presents our method in detail. Section 4 introduces the experimental setup. Section 5 demonstrates the experimental results. Section 6 presents some discussions about our model. Section 7 analyzes the threats to validity. Section 8 concludes the paper and points out the future work.

2. Background and related work

2.1. The characteristics of SATDs

To reduce the cost of locating the defects when modifying the code later, developers usually use comments to indicate whether the code contains technical debts. However, code comments also have their characteristics so that SATDs are too hard to be found. SATDs have the following characteristics:

Length diversity: The code comments that contain SATDs can be represented as one word (such as *"wtf"*), two words (such as *"not implemented"* and *"ugly workaround"*), and three words (such as *"unused in parent"*) (Maldonado et al., 2017), which indicates that code comments have different lengths.

Project uniqueness: Generally, a project needs to be implemented by several developers and each developer has their own coding style. Different developers and various coding styles lead to the diversity of the code comments in the project. For example, the comment *"wtf"* in JEdit does not appear in other projects (Maldonado et al., 2017). Thus, code comments of each project are unique because of different developers.

Rich semantics: There is not just one way to describe things, and we can describe meaning of a sentence in many ways. Taking the sentence *"this may be negative"* as an example, this sentence can be replaced as *"this may be not positive"* (Maldonado et al., 2017), which also expresses the same meaning.

2.2. Identifying self-admitted technical debt

There are two popular ways to identify SATDs, i.e. pattern-based methods and machine learning based methods.

De Freitas Farias et al. (2015) proposed a contextualized vocabulary model for identifying SATDs. Their experimental results on two open source projects showed that the model may support the development team to detect SATD using code comment analysis. Maldonado and Shihab (2015) proposed an open-source dataset and classified SATDs into five types manually. Their experimental results on five open source projects showed that the design debt was the most common type of SATDs. Huang et al. (2018) proposed a text mining method to detect SATDs automatically. They utilized feature selection technique to select key features, and built a composite classifier that combined multiple classifiers from different projects. Their experiment results on eight open source projects showed that their method achieved higher F1-score.

Maldonado et al. (2017) employed a natural language processing technique to detect SATDs. They analyzed the production process of the dataset and used maximum entropy to identify SATDs. Their experiment results on 10 open source projects showed that their method achieved a good accuracy even with a relatively small training set. Wattanakriengkrai et al. (2018) proposed a machine learning model to identify SATDs which combined N-gram inverse document frequency and auto-sklearn machine learning. They conducted experiments on 10 projects and the results showed that their method obtained higher F1-score than the baseline methods to identify requirement and design debts. Ren et al. (2019) proposed a neural network based model to detect SATDs. They conducted experiments on 10 projects and the results showed a significant performance improvement of their method.

2.3. Graph neural network

In recent years, graph neural networks (GNN) (Scarselli et al., 2008) have been widely used in various fields, such as social networks (Fan et al., 2019; Islam et al., 2019), knowledge graphs (Koncel-Kedziorski et al., 2019; Park et al., 2019), recommendation systems (Yin et al., 2019; Wu et al., 2019b), and even life sciences (Wang et al., 2021; Yang et al., 2021). GNN has a powerful function in modeling the dependency relationship of graph structures.

Peng et al. (2018) proposed a graph-CNN based deep learning model to classify hierarchical text. Their experiment results on two datasets showed that their model significantly improved the performance. Yao et al. (2019) employed graph-based neural networks to classify text. They conducted experiments on five datasets and the results showed the effectiveness of their model. Wu et al. (2019a) explored the method that reduced excess complexity of graph convolutional networks. They conducted experiments on eight datasets and the results showed that their method did not negatively impact the accuracy in downstream applications. Huang et al. (2019) proposed a graph network model at the text level for text classification. Their experiment results on three datasets showed that their model outperforms existing models in text classification task. Pal et al. (2020) proposed a multi-label text classification model based on graph attention network. Their experiment results on five datasets showed that their model achieved better performance.

3. Method

3.1. Overview

Fig. 1 presents an overview of our proposed GGSATD method that consists of the following four steps. Firstly, we build the document graph by using the comment instance from the source code and generate the corresponding adjacent matrix. Noting that, the edges in this graph are non-weighted in our work. Secondly, the gated graph neural network (Li et al., 2015c) is applied to update iteratively the node representation with the adjacent matrix generated by the previous step to learn more representative features for each node. After the graph information update finished, in the third step, the representation layout is used to process high-level features. Finally, the feature vectors are input into the softmax layer for identifying whether this comment instance is a SATD. Below, we present the detail of each step.

3.2. Build document graph

A graph consists of a set of nodes and relationship-edges. For a given graph $G(V, E)$ in which $V = \{v_1, v_2, \ldots, v_{|V|}\}$ denotes the node set and $E = \{e_1, e_2, \ldots, e_{|E|}\}$ denotes the edge set. $|V|$ and $|E|$ denote the number of nodes and edges, respectively. If the edge between two nodes in the graph is undirected, the graph is deemed to an undirected graph. 2 shows an example of an undirected graph. Assume that nodes v_1 and v_2 have an edge, they are called mutually adjacent to each other. We use the adjacency matrix to store the edge relationship between the nodes. 2 shows

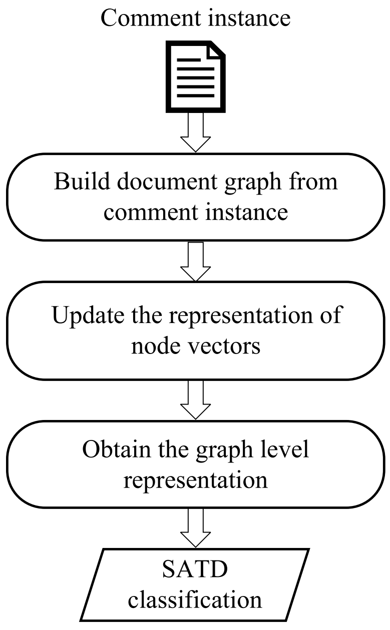
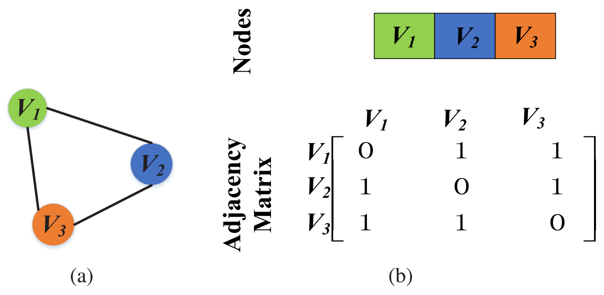

Comment instance
Build document graph from comment instance
Update the representation of node vectors
Obtain the graph level representation
SATD classification

Fig. 1. The overview process of our method.



$$\begin{array}{c|ccc} & V_1 & V_2 & V_3 \\ \hline V_1 & 0 & 1 & 1 \\ V_2 & 1 & 0 & 1 \\ V_3 & 1 & 1 & 0 \end{array}$$

Fig. 2. (a) Undirected graph. A simple undirected graph is composed of three nodes v_1, v_2, and v_3, and the undirected edges between them. (b) The storage method of the adjacency matrix.

the node set of the undirected graph and the adjacency matrix of this graph.

We construct the node set of the graph in three steps. First, we preprocess the original data, such as delete redundant spaces and meaningless symbols. Second, we convert the preprocessed data into word vectors through the pre-trained model Global vectors for word representation (Glove) (Pennington et al., 2014), which can better represent the semantics and grammars. Third, for the out of vocabulary (OOV) words that do not appear in the pre-trained model Glove, we randomly sample values from −0.01 to 0.01 as the word vector. We construct the edge set of the graph using the co-occurrence relationship between words in a sliding window.

Taking the comment instance "*Fixme we should really assert a value here*", as an example, we convert this instance into the word vector and mark it as $h(h_1, h_2, h_3, \ldots, h_V)$, $h \in \mathbb{R}^{|V|\times d}$, in which d is the dimension of the word embedding. As shown in Fig. 3, the co-occurrence relationship in a sliding window (the dotted line) is converted as the edges in the document graph.

3.3. Graph-gated neural network

Gate mechanism can control the capacity of information needed to be retained or discarded, and the capacity of new state information needed to be stored in the memory unit. In order to

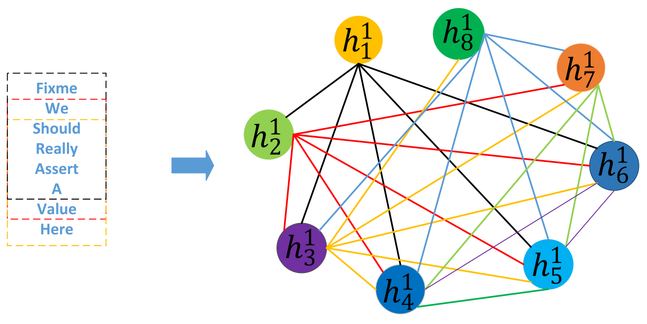


Fig. 3. An example for building document graph.

effectively propagate and update information in the graph, the gate mechanism is introduced in our method.

Li et al. (2015c) incorporated the Gated Recurrent Unit (GRU) (Cho et al., 2014) into GNN, which unrolled the loop in a fixed number of steps T and used back propagation over time to calculate the gradient. GGNN is a classical spatial domain message passing model (Gilmer et al., 2020) that includes three operations: message passing operation, update operation, and read operation. Inspired by this, we integrate gate mechanism to the update process of the graph representation, which takes the advantage of updating information by fixed steps and ensuring convergence without constraining parameters (Wu et al., 2020).

We start with the flow chart of the GRU as shown in Fig. 4. First, we use the adjacency matrix of the node to transmit information. We set the information to be passed as I, which is formulized as follows:

$$I^t = Xh^{t-1}W_I \tag{1}$$

where X represents an adjacency matrix, h^{t-1} represents the feature vector of the node under step $t-1$, and W^I presents the trainable parameter.

Second, we need to set up an update gate and z^t is used as a substitute symbol. It can control how many historical states (i.e., h^1, h^2, ..., and h^{t-1}) and candidate states needed to be retained for the output state h^t at the current step, which is formulized as follows:

$$z^t = \sigma(W_z I^t + U_z h^{t-1} + b_z) \tag{2}$$

where W_z, U_z, and b_z are trainable parameters, and σ represents the sigmoid function.

Third, we need to set up a reset gate and we define it as r^t. The goal of reset gate is to determine whether the previous state information is needed and how much is it needed for the current candidate state, which is formulized as follows:

$$r^t = \sigma(W_r I^t + U_r h^{t-1} + b_r) \tag{3}$$

where W_r, U_r, and b_r are trainable parameters.

Fourth, we define the candidate state $\widetilde{h}^t$ including the current node representation and previous information, which is formulized as follows:

$$\widetilde{h}^t = \tanh(W_h I^t + U_h(r^t \odot h^{t-1}) + b_h) \tag{4}$$

where W_h, U_h, and b_h are trainable parameters. $\odot$ is element-wise multiplication.

Finally, we update the current state h^t by forgetting the unnecessary information and reserving key information from candidate state $\widetilde{h}^t$, which is formulized as follows:

$$h^t = \widetilde{h}^t \odot z^t + h^{t-1} \odot (1 - z^t) \tag{5}$$

We repeat the above operations until the nodes are fully updated.

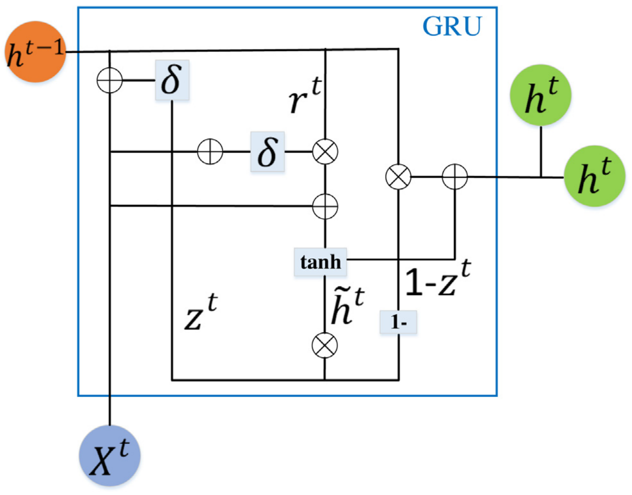


Fig. 4. The structure diagram of the GRU.

3.4. Representation layout

After the completion of graph updating with the GGNN, we want to obtain a more effective representation at graph level. First, we use two multilayer perceptrons to get the enhanced representation of the node, which is formulized as follows:

$$H = \sigma(m_1(H^t) \odot \tanh(m_2(H^t))) \tag{6}$$

where m_1 is a multilayer perceptron with soft attention mechanism and m_2 is a multilayer perceptron with non-linear feature transformation. Soft attention mechanism is a global calculation way proposed by Bahdanau et al. (2014), which calculates the weight probability for all nodes and each node has its own weight. In addition, we use the non-linear model m_2 to enhance the node representation.

We merge node information to form a graph-level representation. We use the average-pooling algorithm (Yu et al., 2014) and the max-pooling algorithm (Christlein et al., 2019) to obtain the graph-level representation based on the node representation. Average pooling retains the characteristics of global node information, which is defined as follows:

$$h_{g-a} = \frac{1}{|V|}\sum H \tag{7}$$

where h_{g-a} represents the graph-level representation produced by the avg-pooling operation.

Maximum pooling can extract the most prominent features, which is defined as follows:

$$h_{g-m} = maxpooling(h_1, \ldots, h_V) \tag{8}$$

where h_{g-m} represents the graph-level representation produced by the max-pooling operation.

Then, we concatenate the results of the above two operations to get a graph-level representation of the code comment instance, which is defined as follows:

$$h_g = h_{g-a} \oplus h_{g-m} \tag{9}$$

where $\oplus$ represents the concatenation operation. Finally, we take the graph-level representation h_g as the final embedding vector followed by the softmax layer to detect SATDs. This process is shown in Fig. 5.

Table 1
Statistics of the 10 projects.

Project	Version	#Contributors	#Comments	#F-Comments	#SATDs	%SATDs
Apache Ant	1.7.0	74	21,587	4,137	131	0.6%
ArgoUML	0.34	87	67,716	9,548	1,413	2.08%
Columba	1.4	9	33,895	6,478	204	0.6%
EMF	2.4.1	30	25,229	4,401	104	0.41%
Hibernate	3.3.2	226	11,630	2,968	472	4.05%
JEdit	4.2	57	16,991	10,322	256	1.5%
JFreechart	1.0.19	19	23,475	4,423	209	0.89%
JMeter	2.1	33	20,084	8,162	374	1.86%
JRuby	1.4.0	328	11,149	4,897	662	5.57%
SQuirrel	3.0.3	46	27,474	7,230	285	1.04%
Average	–	–	25,923	6,257	411	1.86%

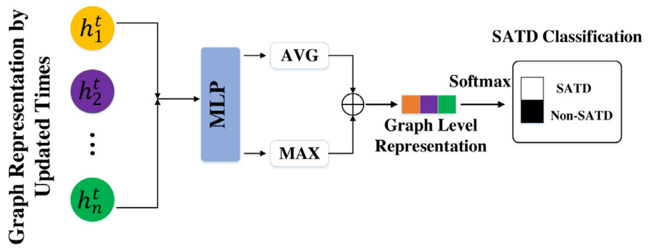


Fig. 5. The structure of representation layout layer.

4. Experimental setup

4.1. Research questions

To measure the performance of our method, we design and answer the following three research questions (RQs).

RQ1 : Does our proposed GGSATD method effectively detect SATDs in both within-project and cross-project scenarios?

As our proposed GGSATD method exploits gated graph neural network to effectively update the information derived from the document graph and learn the global representation at graph level, this question is designed to evaluate the effectiveness of our GGSATD for automatic SATD classification in within-project and cross-project scenarios.

RQ2 : Is our proposed GGSATD method superior to other existing methods?

Recently, researchers have proposed some methods for textual classification task and have shown promising performance (He and Zhu, 2019; Hu et al., 2020; Sachan et al., 2019). As the goal of our proposed GGSATD method help developers to identify the intentional technical debt in the comment instance, aiming at reducing the difficulty of later software maintenance and improving software quality, it can also be deemed as a text classification task. This question is designed to investigate whether our GGSATD method can achieve better performance than existing methods for automatically identifying SATDs.

RQ3 : How effective is our proposed GGSATD method for detecting SATDs in real-world projects?

As our proposed GGSATD can automatically identify SATDs in publicly available projects released by the previous work (Ren et al., 2019), this question is designed to explore whether our method is also effective in real-world projects, which is very important for engineering applications.

4.2. Dataset

In this work, we conduct experiments on a benchmark dataset collected by Maldonado et al. (2017), which consists of the following 10 open source projects:

Apache Ant is a tool to automate software compilation, testing, deployment and other steps in Java environment; ArgoUML is an open source UML modeling tool for Java platform; Columba is an email client based on Java; EMF is an Eclipse-based model framework which can transform the model into efficient, correct, and easy-to-customize Java code; Hibernate is a framework for object relationship mapping; JEdit is a text editor based on Java; JFreechart is an open chart drawing library on Java platform; JMeter is a stress testing tool based on Java; JRuby is an implementation of the Ruby language; SQuirrel is a database client.

The procedure for collecting this dataset is as follows. First, Maldonado et al. (2017) used an Eclipse plug-in (i.e. JDeodorand[3]) to extract code comments from 10 open source projects. Second, they used five heuristic rules to filter the comments and classified them manually. Finally, they used the Cohen's Kappa coefficient (Cohen, 1968) to demonstrate that the dataset had a high confidence. Table 1 shows the basic statistic information of the 10 projects, including the projects version, the number of contributors (#Contributors), the number of comments (#Comments), the number of filtered comments (#F-Comments), the number of comment instances that are labeled as SATD (#SATDs), and the proportion of SATDs (%SATDs).

4.3. Parameter settings

We map each node into a 300-dimensional vector using the Glove technique. For building graph from code comment instance, we choose the size of the sliding window as 6. For training model, we set the learning rate as 0.001 which can make the objective function locally converge. We set the batch size as 32, the dropout as 0.5, and the size of hidden layer as 96. We stack two layers of GGNN units to iteratively update the node representation. We have released the code scripts and benchmark dataset at https://figshare.com/articles/dataset/JSS-Graph/16869737 for reproducing our experiments.

4.4. Performance indicators

To evaluate the performance of our proposed GGSATD method, we employ the Precision, Recall, and F1-score as the evaluation indicators following the previous studies (Huang et al., 2018; Ren et al., 2019; Wang et al., 2020). There are four basic items used in the indicators.

- TP: the number of the comment instances that are SATDs and are predicted as SATDs;
- FP: the number of the comment instances that are non-SATDs but are predicted as SATDs;
- TN: the number of the comment instances that are non-SATDs and are predicted as non-SATDs;

3 https://marketplace.eclipse.org/content/jdeodorant.

Table 2
The classification results of our model with three indicators in two scenarios.

Project	Within			Cross		
	Precision	Recall	F1-score	Precision	Recall	F1-score
Apache Ant	0.826	0.754	0.779	0.806	0.818	0.811
ArgoUML	0.937	**0.955**	**0.945**	0.909	**0.946**	0.926
Columba	0.944	0.914	0.927	0.936	0.942	**0.939**
EMF	0.929	0.844	0.905	0.823	0.786	0.802
Hibernate	0.923	0.891	0.877	0.927	0.894	0.909
JEdit	0.854	0.797	0.820	0.864	0.720	0.782
JFreeChart	0.948	0.921	0.923	0.831	0.753	0.786
JMeter	**0.951**	0.919	0.933	0.905	0.899	0.902
JRuby	0.934	0.938	0.930	**0.938**	0.914	0.926
SQuirrel	0.912	0.903	0.907	0.851	0.820	0.834
Average	0.916	0.884	0.895	0.879	0.849	0.862

- FN: the number of the comment instances that are SATDs but are predicted as non-SATDs.

Precision represents the reliability of the prediction, which is formulized as follows:

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

Recall represents whether the model has the ability to predict all data, which is formulized as follows:

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

F1-score is a harmonic mean of Precision and Recall, which is formulized as follows:

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{12}$$

The above-mentioned three evaluation indicators are widely-used in software engineering tasks (Valdivia Garcia and Shihab, 2014; Xia et al., 2015; Xu et al., 2020; Zhao et al., 2021b,c,a; Xu et al., 2019).

5. Results

5.1. Answer to RQ1: the effectiveness of our proposed GGSATD method for SATD classification

Methods: To answer this question, we explore the performance of SATD classification in two scenarios, i.e. within-project and cross-project.

Within-project: we choose 10-fold cross-validation technique (Efron, 1983) to evaluate the effectiveness of GGSATD for detecting SATD. Firstly, we divide the comment instances into two parts, i.e., non-SATD and SATD instances. Secondly, we separately divide these two parts into ten subsets randomly with equal size. Thirdly, we combine a subset with SATD instances and a subset with non-SATD instances into a new subset every time, and we get 10 such subsets. Then, we select nine subsets as the candidate set and the remainder subset as the test set. We randomly select 10% comment instances as the validation set from the candidate set and the remainder as the training set. We report the average values in our work.

Cross-project: we integrate nine projects as the source project and the remaining one project as the target project to conduct the cross-project SATD classification task. Similarly, we randomly select 10% of the comment instances from the source project as the validation set. We repeat this process 10 times and report average values.

Results: Table 2 presents the classification results of our GGSATD model on each project in two scenarios. Noting that, the best results are marked in bold and the worst results are marked with underline. From this table, we can see that, in the within-project scenario, our model obtains the best Precision value of 0.951 in JMeter, the best Recall value of 0.955 and the best F1-score value of 0.945 in ArgoUML. Our GGSATD method obtains the average performance values with 0.916, 0.884, and 0.895 in terms of Precision, Recall, and F1-score, respectively. In the cross-project scenario, our model obtains the best Precision value of 0.938 in JRuby, the best Recall value of 0.946 in ArgoUML, and the best F1-score value of 0.939 in Columba. Our method acquires the average performance values with 0.879, 0.849, and 0.862 in terms of Precision, Recall, and F1-score, respectively.

In Summary, our proposed GGSATD method achieves satisfactory performance for identifying SATDs in within-project and cross-project scenarios.

5.2. Answer to RQ2: the performance of our method compared to other existing methods

Methods: In order to answer this question, we set up five baseline methods for comparison, which are introduced as follows:

- HATD: this method (Wang et al., 2020) used a two-layer Bi-LSTM network and single-head attention to compose a SAE model, and used positional encoding and multi-head attention mechanism to compose a MAE model. Then the representation of SAE and MAE is concatenated to detect SATDs. HATD is the current state-of-the-art method.
- CNN: this method (Ren et al., 2019) used convolutional neural network to identify SATDs. The CNN model extracted feature vectors from input comment instances through volume base layer and 1-max pooling operation, and classified the comment instances according to the feature vectors.
- GCN: this method (Yao et al., 2019) built a graph based on all the data in which each node is corresponding to the word and text in the document and each edge is defined as the co-occurrence relationship between nodes. The graph was trained through two fully-connected layers, and the output is classified through the softmax function.
- TLGNN: this method (Huang et al., 2019) was used GNN to classification task on text level. The model constructed a graph for each input text with global parameter sharing, instead of constructing a single graph for the entire corpus.
- LSTM: this method (Silva, 1997) was a special recurrent neural network that could retain long-term memory, which solved the problems of gradient vanishing and gradient explosion in long sequence training.

Results: Tables 3–5 show the Precision, Recall and F1-score values in within-project scenario, respectively. From Table 3, we can see that, our GGSATD method obtains the best Precision value in seven projects and gets average improvements by 18.19%, 22.13%, 26.87%, 23.45%, and 32.18% compared with HATD, CNN, GCN, TLGNN, and LSTM, respectively. Our method obtains the best average Precision value and achieves an average improvement by 24.57% among five baseline methods. From Table 4, we can find that, our GGSATD method obtains the best Recall value in seven projects and achieves average improvements by 4.12%, 12.04%, 15.56%, 28.49%, and 35.58% compared with five baseline methods, respectively. Our method obtains the best average Recall value and achieves an average improvement by 19.16% among five baseline methods. From Table 5, we can observe that, our GGSATD method obtains the best F1-score value in nine projects and achieves average improvements by 8.75%, 19.81%, 21.44%, 26.95%, and 38.54% compared with five baseline methods, respectively.

Table 3
Precision values of our method and the five baselines in within-project scenario.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Apache Ant	**0.826**	0.705	0.323	0.550	0.517	0.484
ArgoUML	0.937	0.822	**0.938**	0.822	0.865	0.927
Columba	0.944	0.907	**0.954**	0.743	0.799	0.746
EMF	**0.929**	0.738	0.438	0.680	0.636	0.488
Hibernate	**0.923**	0.898	0.692	0.769	0.812	0.596
JEdit	0.854	0.502	**0.856**	0.640	0.590	0.771
JFreeChart	**0.948**	0.898	0.766	0.815	0.869	0.526
JMeter	**0.951**	0.734	0.935	0.709	0.764	0.855
JRuby	**0.934**	0.882	0.847	0.786	0.860	0.788
SQuirrel	**0.912**	0.664	0.750	0.711	0.707	0.744
Average	0.916	0.775	0.750	0.722	0.742	0.693

Table 4
Recall values of our method and the five baselines in within-project scenario.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Apache Ant	**0.754**	0.611	0.719	0.583	0.513	0.500
ArgoUML	**0.955**	0.918	0.943	0.825	0.876	0.951
Columba	0.914	**0.952**	0.827	0.787	0.745	0.708
EMF	0.844	**0.853**	0.719	0.698	0.594	0.500
Hibernate	0.891	0.887	**0.916**	0.793	0.744	0.506
JEdit	**0.797**	0.640	0.556	0.711	0.539	0.729
JFreeChart	**0.921**	0.903	0.583	0.860	0.735	0.502
JMeter	**0.919**	0.917	0.827	0.814	0.705	0.794
JRuby	**0.938**	0.931	0.918	0.826	0.817	0.753
SQuirrel	**0.903**	0.876	0.888	0.755	0.607	0.578
Average	0.884	0.849	0.789	0.765	0.688	0.652

Table 5
F1-score values of our method and the five baselines in within-project scenario.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Apache Ant	**0.779**	0.655	0.445	0.559	0.513	0.492
ArgoUML	**0.945**	0.867	0.941	0.823	0.870	0.938
Columba	**0.927**	**0.927**	0.877	0.757	0.763	0.704
EMF	**0.905**	0.891	0.887	0.779	0.770	0.471
Hibernate	**0.877**	0.776	0.532	0.683	0.610	0.494
JEdit	**0.820**	0.556	0.591	0.664	0.552	0.743
JFreeChart	**0.923**	0.900	0.636	0.827	0.779	0.491
JMeter	**0.933**	0.892	0.872	0.748	0.726	0.811
JRuby	0.930	**0.931**	0.881	0.803	0.834	0.733
SQuirrel	**0.907**	0.836	0.813	0.730	0.636	0.582
Average	0.895	0.823	0.747	0.737	0.705	0.646

Table 6
Precision values of our method and the five baselines in cross-project scenario.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Apache Ant	**0.806**	0.657	0.584	0.578	0.618	0.706
ArgoUML	**0.909**	0.818	0.812	0.576	0.823	0.890
Columba	**0.936**	0.794	0.830	0.741	0.711	0.843
EMF	**0.823**	0.664	0.793	0.657	0.643	0.717
Hibernate	0.927	0.756	**0.930**	0.647	0.851	0.893
JEdit	**0.864**	0.699	0.773	0.603	0.649	0.747
JFreeChart	**0.831**	0.678	0.686	0.717	0.674	0.761
JMeter	**0.905**	0.701	0.873	0.674	0.778	0.852
JRuby	**0.938**	0.758	0.805	0.665	0.819	0.894
SQuirrel	**0.851**	0.671	0.794	0.556	0.681	0.736
Average	0.879	0.719	0.788	0.641	0.725	0.804

Table 7
Recall values of our method and the five baselines in cross-project scenario.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Apache Ant	**0.818**	0.780	0.758	0.573	0.639	0.732
ArgoUML	0.946	0.925	**0.950**	0.720	0.860	0.942
Columba	0.942	**0.956**	0.875	0.669	0.877	0.928
EMF	**0.786**	0.785	0.594	0.561	0.684	0.688
Hibernate	**0.894**	0.839	0.743	0.747	0.834	0.830
JEdit	**0.720**	0.713	0.489	0.623	0.588	0.665
JFreeChart	0.753	0.748	**0.802**	0.617	0.722	0.756
JMeter	**0.899**	0.871	0.787	0.693	0.842	0.866
JRuby	0.914	0.913	**0.930**	0.707	0.791	0.889
SQuirrel	0.820	**0.851**	0.692	0.572	0.736	0.776
Average	0.849	0.838	0.762	0.648	0.757	0.807

Table 8
F1-score values of our method and the five baselines in cross-project scenario.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Apache Ant	**0.811**	0.713	0.660	0.575	0.626	0.713
ArgoUML	**0.926**	0.855	0.878	0.595	0.839	0.913
Columba	**0.939**	0.848	0.852	0.697	0.767	0.878
EMF	**0.802**	0.669	0.679	0.581	0.658	0.690
Hibernate	**0.909**	0.776	0.826	0.676	0.841	0.857
JEdit	**0.782**	0.671	0.599	0.612	0.609	0.694
JFreeChart	**0.786**	0.703	0.739	0.643	0.694	0.754
JMeter	**0.902**	0.747	0.828	0.683	0.806	0.858
JRuby	**0.926**	0.798	0.863	0.682	0.803	0.888
SQuirrel	**0.834**	0.707	0.739	0.562	0.704	0.751
Average	0.862	0.749	0.766	0.630	0.735	0.800

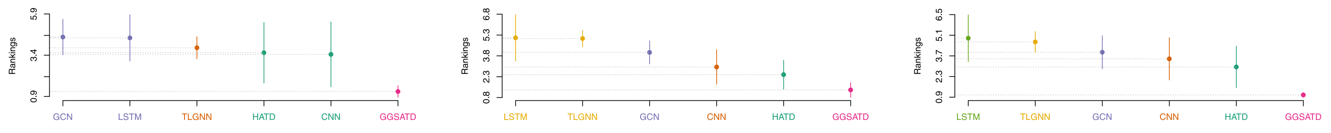
Our method obtains the best average F1-score value and achieves an average improvement by 23.10% among five baseline methods.

Tables 6–8 show the Precision, Recall and F1-score values in cross-project scenario, respectively. From Table 6, we can observe that, our method obtains the best Precision value in nine projects and achieves average improvements by 22.25%, 11.55%, 37.13%, 21.24%, and 9.33% compared with HATD, CNN, GCN, TLGNN, and LSTM, respectively. Our method obtains the best average Precision value and achieves an average improvement by 20.30% among five baseline methods. From Table 7, we can find that, our GGSATD method obtains the best Recall value in five projects and achieves average improvements by 1.31%, 11.42%, 31.02%, 12.15%, and 5.20% compared with five baseline methods, respectively. Our method obtains the best average Recall value and achieves an average improvement by 12.22% among five baseline methods. From Table 8, we can see that, our method obtains the best F1-score value among all projects and achieves average improvements by 15.09%, 12.53%, 36.83%, 17.28%, and 7.75% compared with five baseline methods, respectively. Our method gets the best average F1-score value and achieves an average improvement by 17.89% among five baseline methods.

We apply a state-of-the-art method, namely Scott–Knott Effect Size Difference (short for SKESD) test (Tantithamthavorn et al., 2016), to analyze the significant differences between our GGSATD method and the five baseline methods. The SKESD test corrects the dataset of non-normal distribution and merges two groups with negligible effect size into one group. A method gets the lower ranking in SKESD test means that it obtains better performance. Figs. 6 and 7 visualize the SKESD statistical test results for our method and the five baseline methods in terms of three indicators in within-project scenario and cross-project scenario, respectively. In these two figures, the point represents the mean value and the line represents the interval of the standard deviation. These figures illustrate that our method always ranks the first and has significant differences compared with the five baseline methods in terms of all indicators in both scenarios.

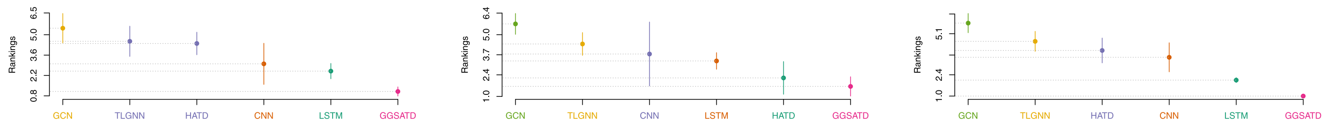
It is worth mentioning that, the previous work (Ren et al., 2019) has confirmed that the CNN model obtained better classification performance than traditional text mining based and pattern based methods. From the above analysis, we can find that our method performs better than the CNN model. Thus, we can consider that our method can obtain better classification performance than traditional text mining based and pattern based methods for identifying SATDs.

Besides, we also perform efficiency analysis and record the execution time of our method and the state-of-the-art method



(a) Precision (b) Recall (c) F1-score

Fig. 6. SKESD test results for our method and five baseline methods with three indicators in within-project scenario.



(a) Precision (b) Recall (c) F1-score

Fig. 7. SKESD test results for our method and five baseline methods with three indicators in cross-project scenario.

Table 9
Statistics of 7 extension projects.

Project	Release	#Comments	#SATDs	%SATDs
Kafka	2.7.0	23,084	141	0.61%
Logback	1.2.0	5,184	44	0.85%
Mybatis3	3.5.6	3,267	52	1.59%
React	17.0.1	23,657	826	3.49%
Spring	4.3.30	48,532	1,564	3.22%
Tomcat	7.0.x	41,037	903	2.20%
zookeeper	3.5.9	7,808	96	1.23%
Average	–	–	518	1.89%

Table 10
The number of SATDs predicted by our method and two manual strategies.

Project	GGSATD	Majority	All-agree
Kafka	13	15	12
Logback	4	6	3
Mybatis3	7	11	6
React	9	12	8
Spring	14	16	13
Tomcat	16	18	15
Zookeeper	15	17	13

HATD. We find that the average time of predicting one comment instance among all 10 projects by HATD is 1.45 s and 12.45 s in within- and cross-project scenarios respectively, whereas our GGSATD model just spends 0.08 s and 2.34 s respectively. This implies that our model is more efficient than the state-of-the-art method HATD.

In summary, our method is significantly superior to the five comparative baseline methods for classifying SATDs.

5.3. Answer to RQ3: the performance of our method in real-world projects

Methods: To answer this question, we collect seven real-world projects from GitHub as follows to evaluate the prediction performance of our GGSATD method. Kafka is an open-source distributed event streaming platform. Logback is a logging component that intends as a successor to log4j project. Mybatis is a persistence framework that supports for customized SQL, stored procedures, and advanced mappings. React is a javascript library for building user interfaces. Spring is an open-source application framework based on J2EE. Tomcat is a web application server. Zookeeper is a distributed application service. We select these seven projects because they are the most popular projects and have tens of thousands of individual and business users.

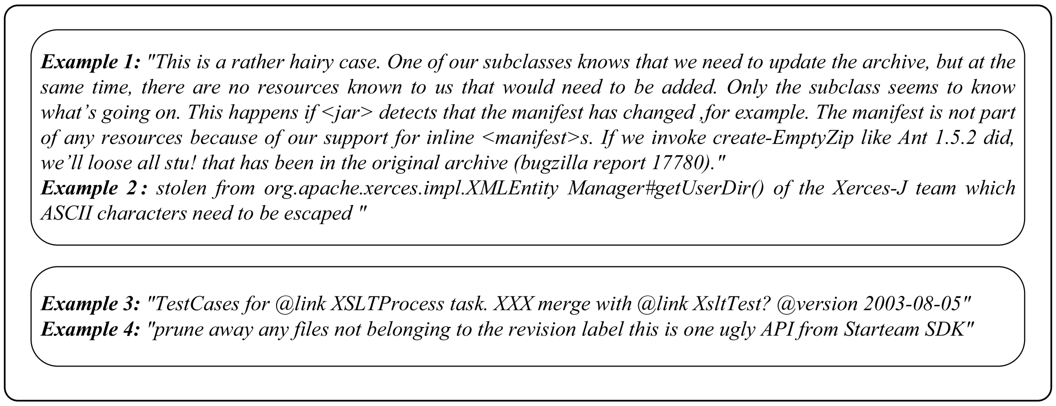
We extract the comment instances from the source code of the seven projects and separately treat each project as the test set to evaluate our model trained on the existing 10 projects. We randomly select 50 comment instances for each project and invite five programmers with at least three-year development experience. We ask them whether the comment instances are SATDs and use the following two strategies to determine the ground truth: The first one is the majority strategy that means if more than three people treat the comment instance as the SATD and this comment instance is deemed as the SATD, otherwise, non-SATD. The second is the all-agree strategy that means if all the people treat the comment instance as the SATD and this comment instance is deemed as the SATD, otherwise, non-SATD.

Results: Table 9 presents the basic statistical information and the prediction results of our model. From this table, we can see that, the average proportion of the comment instances that are predicted as SATD by our GGSATD method among the seven projects is 1.89%, which is closer to the average proportion (i.e., 1.86%) in the existing 10 projects provided by Maldonado et al. (2017) (see Table 1). This result indicates that our GGSATD method is effective when detecting the real-world SATDs.

To confirm the prediction results of our model, we randomly select 50 comment instances from each project (totally 350 comment instances), and use the majority strategy and all-agree strategy to identify SATDs manually. Table 10 presents the number of SATDs predicted by our method and two manual strategies. From this table, we can see that, the predicted result of our method is between the results of majority strategy and all-agree strategy, which means that our method is more rigorous than majority strategy. In addition, it also reduces the strong absoluteness of all-agree strategy.

Tables 11 and 12 represent the performance of our model and baseline methods for identifying SATDs with F1-score indicator in majority strategy and all-agree strategy, respectively. In terms of majority strategy, our method achieves the best average F1-score of 0.911 and obtains the average improvements by 57.02%, 8.37%, 26.13%, 20.84%, and 17.19% compared with HATD, CNN, GCN, TLGNN, and LSTM, respectively. In terms of all-agree strategy, our method obtains the best the average F1-score of 0.921 and achieves average improvements by 66.01%, 13.60%, 29.10%, 26.24%, and 28.79% compared with the five baseline methods, respectively.

In summary, our proposed GGSATD method obtains better performance than the five baseline methods in real-world projects.


Example 1: *"This is a rather hairy case. One of our subclasses knows that we need to update the archive, but at the same time, there are no resources known to us that would need to be added. Only the subclass seems to know what's going on. This happens if <jar> detects that the manifest has changed ,for example. The manifest is not part of any resources because of our support for inline <manifest>s. If we invoke create-EmptyZip like Ant 1.5.2 did, we'll loose all stu! that has been in the original archive (bugzilla report 17780)."*
Example 2 : *stolen from org.apache.xerces.impl.XMLEntity Manager#getUserDir() of the Xerces-J team which ASCII characters need to be escaped "*

Example 3: *"TestCases for @link XSLTProcess task. XXX merge with @link XsltTest? @version 2003-08-05"*
Example 4: *"prune away any files not belonging to the revision label this is one ugly API from Starteam SDK"*

Fig. 8. The error predicted instances in Apache Ant.

Table 11
F1-score values of our model and five baselines using majority strategy in 7 extension projects.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Kafka	**0.950**	0.594	0.836	0.721	0.688	0.836
Logback	**0.889**	0.621	0.847	0.689	**0.889**	0.647
Mybatis3	**0.865**	0.592	0.817	0.688	0.847	0.851
React	**0.859**	0.681	0.781	0.667	0.709	0.822
Spring	**0.952**	0.458	0.873	0.790	0.844	0.830
Tomcat	**0.911**	0.464	0.883	0.814	0.653	0.752
Zookeeper	**0.954**	0.653	0.851	0.690	0.650	0.706
Average	0.911	0.580	0.841	0.723	0.754	0.778

Table 12
F1-score values of our model and five baselines using all strategy in 7 extension projects.

Project	GGSATD	HATD	CNN	GCN	TLGNN	LSTM
Kafka	**0.920**	0.569	0.773	0.653	0.610	0.750
Logback	**0.923**	0.573	0.864	0.548	**0.923**	0.691
Mybatis3	**0.956**	0.514	0.752	0.813	0.777	0.625
React	**0.838**	0.726	0.729	0.653	0.681	0.709
Spring	**0.975**	0.488	0.867	0.740	0.802	0.792
Tomcat	**0.883**	0.438	0.830	0.870	0.650	0.671
Zookeeper	**0.950**	0.577	0.859	0.714	0.661	0.767
Average	0.921	0.555	0.810	0.713	0.729	0.715

6. Discussion

In this section, we first discuss to what extent the different parameter settings impact our proposed GGSATD method. We mainly focus on the different learning rates and dimensions of the word embedding. In this part of the experiments, we only modify the parameters that need to be discussed while ensuring that other parameters remain unchanged, and observe the impact of those parameters on our method. Then we discuss how some specific patterns proposed in Yu et al. (2020) impact our method. Finally, we conduct an error analysis of our experiment results.

6.1. The impact of different parameter settings

The impact of different learning rates. We empirically choose four different learning rates from {0.001, 0.002, 0.005, 0.01} and conduct experiments with each setting in within-project and cross-project scenarios, respectively. Table 13 presents the results on each learning rate (i.e., LR) in terms of three indicators. From this table, we can see that, our model with learning rate 0.001 obtains better performance with three indicators in both scenarios except for the Precision indicator in the cross-project scenario. Thus, lower learning rate will possibly be more suitable for our method. The reason why we do not choose the learning rate that is less than 0.001 is that the model with very small learning rate may result in the slower decline of the model loss, which is time-consuming.

The impact of different word embedding dimensions. We empirically choose three word embedding dimensions from {100, 200, 300} and conduct experiments with each setting in within-project and cross-project scenarios, respectively. Table 13 presents the results on each word embedding dimension (i.e., DIM) in terms of three indicators. From this table, we can see that, our model with different dimensions only has slight differences with three indicators in the two scenarios. However, our model with dimension 300 seems to be the best choice for detecting SATDs.

6.2. The impact of patterns

A previous study (Yu et al., 2020) suggested that comment instances containing some specific keywords (i.e., patterns), such as "*fixme*", "*todo*", "*workaround*", and "*hack*", are almost related to SATDs and these comment instances are always treated as SATDs (Yu et al., 2020). To explore how such patterns impact our model, we select JEdit as the example to conduct the experiment because it has worse performance in two scenarios. We extract the comment instances containing such patterns and treat them as SATDs. Then, we employ our GGSATD method on the remaining data to determine which comment instances are SATDs. After the experiment, we find that, in within-project scenario, when we discard the comment instances with these patterns, we can obtain the F1-score value of 0.781. However, the F1-score value can achieve 0.816 if these comment instances are treated SATDs directly when calculating the indicator. Meanwhile, the same trend can be found in the cross-project scenario, i.e., the F1-score value increases from 0.543 to 0.703. We try to explain such phenomenon as follows. Recall that the proportion of SATDs in JEdit is 1.5%, which is class imbalanced. When the comment instances with such patterns are removed, the data becomes more imbalanced. As a result, the performance is declined because our model does not consider the class imbalance issue and we will treat this issue as our future work. On the other hand, compared with the results in Table 2, we can find that, such patterns just bring slight impact on the performance of our GGSATD method, which implies that our method is insensitive to such patterns.

6.3. Error analysis

In this section, we conduct the error analysis of our method. We select four comment instances from Apache Ant as examples as shown in Fig. 8. From this figure, we can see that, *Examples*

Table 13
Average results for different parameter settings.

Parameters		Within			Cross		
		Precision	Recall	F1-score	Precision	Recall	F1-score
LR	0.001	**0.916**	**0.884**	**0.895**	0.879	**0.849**	**0.862**
	0.002	0.891	0.821	0.848	0.887	0.804	0.835
	0.005	0.907	0.836	0.864	0.891	0.796	0.829
	0.01	0.904	0.824	0.855	**0.895**	0.763	0.797
DIM	100	0.902	0.862	0.880	**0.895**	0.806	0.838
	200	0.900	0.859	0.879	0.875	0.822	0.842
	300	**0.916**	**0.884**	**0.895**	0.879	**0.849**	**0.862**

1 and *2* are truly SATDs but our method predicts them as non-SATDs because the semantic description of these two comment instances are unobvious. In *Example 1*, the phrase "*hairy case*" is a SATD-prone expression and appears only once in the whole dataset. The phrase "*need to be escaped*" in *Example 2* does not clearly indicate whether the operation is completed and it brings confused meanings. Thus, our method cannot predict them better. *Examples 3* and *4* are truly non-SATDs but our method predicts them as SATDs. The expressions "*XXX*", "*version*" and "*test*" in *Example 3*, and "*Prune away*" and "*ugly*" in *Example 4* are always used in the SATD-prone comment instances. Thus, our method predicts them as SATDs.

7. Threats to validity

Threats to internal validity are associated with the potential errors during our experimental implementation. To relieve these threats, we carefully modify the source code of GGNN provided by the original authors to cater to our task. In addition, all the comparative baseline methods are reproduced based on the Python and TensorFlow libraries, and the first two authors both check the implement details for reducing potential errors. Threats to external validity are associated with the generalization of our proposed GGSATD method. In this work, we conduct experiments on a publicly available dataset released by the recent study (Maldonado et al., 2017) which includes 10 open source projects with a total of 62,566 comments. Since all these projects are developed in Java programming language, we ought to collect comment data from projects developed with other languages to generalize our experiment results. Threats to construct validity are associated with the reasonability of the used performance indicators. In this work, we use three widely-used performance indicators, i.e., Precision, Recall, and F1-score, to evaluate the performance of our proposed GGSATD method.

8. Conclusion

In this paper, we propose a novel method, called GGSATD, to automatically identify the self-admitted technical debts. More specifically, GGSATD first constructs the graph for each comment instance and the Glove technique is used to initial the word embedding vectors. Then, the gated graph neural network is utilized to iteratively update node representation. Finally, the representation layer incorporating multi-layer perceptrons and pooling mechanisms are employed to obtain the graph level representation. The experiments on 10 open-source projects demonstrate that our proposed GGSATD method obtains promising performance compared with five baseline methods in both within-project and cross-project scenarios. In addition, the experimental results on seven real-world projects show the effectiveness of our GGSATD method.

In the future, we plan to collect more data to enrich our experiments and will take class imbalance issue into account because the existing comment data are internally imbalanced.

CRediT authorship contribution statement

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

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grants (No. 61972290), the National Natural Science Foundation of China (No. 62102054), and the Open Foundation of Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education of China (CPSDSC202004).

References

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2015. The financial aspect of managing technical debt: A systematic literature review. Inf. Softw. Technol. 64, 52–73.
Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
Bavota, G., Russo, B., 2016. A large-scale empirical study on self-admitted technical debt. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 315–326.
Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y., 2014. On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.
Christlein, V., Spranger, L., Seuret, M., Nicolaou, A., Král, P., Maier, A., 2019. Deep generalized max pooling. In: 2019 International Conference on Document Analysis and Recognition (ICDAR). IEEE, pp. 1090–1096.
Cohen, J., 1968. Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit. Psychol. Bull. 70 (4), 213.
Cunningham, W., 1992. The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger 4 (2), 29–30.
De Freitas Farias, M.A., de Mendonça Neto, M.G., da Silva, A.B., Spínola, R.O., 2015. A contextualized vocabulary model for identifying technical debt on code comments. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD). IEEE, pp. 25–32.
Efron, B., 1983. Estimating the error rate of a prediction rule: improvement on cross-validation. J. Am. Statist. Assoc. 78 (382), 316–331.
Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., Yin, D., 2019. Graph neural networks for social recommendation. In: The World Wide Web Conference, pp. 417–426.
Flisar, J., Podgorelec, V., 2019. Identification of self-admitted technical debt using enhanced feature selection based on word embedding. IEEE Access 7, 106475–106494.
Foucault, M., Blanc, X., Storey, M.-A., Falleri, J.-R., Teyton, C., 2018. Gamification: a game changer for managing technical debt? a design study. arXiv preprint arXiv:1802.02693.
Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E., 2020. Message passing neural networks. In: Machine Learning Meets Quantum Physics. Springer, pp. 199–214.

He, K., Zhu, M., 2019. Text classification using gated and transposed attention networks. In: 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–7.

Hu, J., Liao, J., Liu, L., Ma, W., 2020. RCapsNet: A recurrent capsule network for text classification. In: 2020 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–8.

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

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

Islam, M.R., Muthiah, S., Ramakrishnan, N., 2019. NActSeer: Predicting user actions in social network using graph augmented neural network. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, pp. 1793–1802.

Izurieta, C., Ozkaya, I., Seaman, C., Snipes, W., 2017. Technical debt: A research roadmap report on the eighth workshop on managing technical debt. ACM SIGSOFT Softw. Eng. Notes 42 (1), 28–31.

Koncel-Kedziorski, R., Bekal, D., Luan, Y., Lapata, M., Hajishirzi, H., 2019. Text generation from knowledge graphs with graph transformers. arXiv preprint arXiv:1904.02342.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. Ieee Softw. 29 (6), 18–21.

Li, Z., Avgeriou, P., Liang, P., 2015a. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220.

Li, Z., Liang, P., Avgeriou, P., 2015b. Architectural technical debt identification based on architecture decisions and change scenarios. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture. IEEE, pp. 65–74.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015c. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Lim, E., Taksande, N., Seaman, C., 2012. A balancing act: What software practitioners have to say about technical debt. IEEE Softw. 29 (6), 22–27.

Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., Li, S., SATD detector: A text-mining-based self-admitted technical debt detection tool.(2018). In: Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE), Vol. 3, pp. 9–12.

Ma, Q., Yuan, C., Zhou, W., Hu, S., 2021. Label-specific dual graph neural network for multi-label text classification. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 3855–3864.

Maldonado, E.d.S., Shihab, E., 2015. Detecting and quantifying different types of self-admitted technical debt. In: Proceedings of the 7th IEEE International Workshop on Managing Technical Debt. IEEE, pp. 9–15.

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

Marinescu, R., 2004. Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. IEEE, pp. 350–359.

Mensah, S., Keung, J., Bosu, M.F., Bennin, K.E., 2016. Rework effort estimation of self-admitted technical debt.

Mittal, V., Gangodkar, D., Pant, B., 2021. Deep graph-long short-term memory: A deep learning based approach for text classification. Wirel. Pers. Commun. 1–15.

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

Pal, A., Selvakumar, M., Sankarasubbu, M., 2020. Magnet: Multi-label text classification using attention-based graph neural network. In: ICAART (2). pp. 494–505.

Park, N., Kan, A., Dong, X.L., Zhao, T., Faloutsos, C., 2019. Estimating node importance in knowledge graphs using graph neural networks. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 596–606.

Peng, H., Li, J., He, Y., Liu, Y., Bao, M., Wang, L., Song, Y., Yang, Q., 2018. Large-scale hierarchical text classification with recursively regularized deep graph-cnn. In: Proceedings of The 27th World Wide Web Conference, pp. 1063–1072.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 19th Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543.

Point, A.T., from Dagstuhl, R., Perspectives on Managing Technical Debt.

Potdar, A., Shihab, E., 2014. An exploratory study on self-admitted technical debt. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 91–100.

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

Sachan, D.S., Zaheer, M., Salakhutdinov, R., (2019). Revisiting lstm networks for semi-supervised text classification via mixed objective function. In: Proceedings of The AAAI Conference on Artificial Intelligence, Vol. 33 (01) pp. 6940–6948.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2008. The graph neural network model. IEEE Trans. Neural Netw. 20 (1), 61–80.

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

Silva, F., 1997. Bridging long time lags by weight guessing and "long short term memory". Spatiotemporal Models Biol. Artif. Syst. 37, 65.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2016. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516.

Valdivia Garcia, H., Shihab, E., 2014. Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 72–81.

Wang, X., Liu, J., Li, L., Chen, X., Liu, X., Wu, H., 2020. 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 (ASE). IEEE, pp. 871–882.

Wang, J., Ma, A., Chang, Y., Gong, J., Jiang, Y., Qi, R., Wang, C., Fu, H., Ma, Q., Xu, D., 2021. ScGNN is a novel graph neural network framework for single-cell RNA-seq analyses. Nature Commun. 12 (1), 1–11.

Wattanakriengkrai, S., Maipradit, R., Hata, H., Choetkiertikul, M., Sunetnanta, T., Matsumoto, K., 2018. 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, pp. 7–12.

Wehaibi, S., Shihab, E., Guerrouj, L., 2016. Examining the impact of self-admitted technical debt on software quality. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). 1, IEEE, pp. 179–188.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y., 2020. A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. 32 (1), 4–24.

Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., Weinberger, K., 2019a. Simplifying graph convolutional networks. In: International Conference on Machine Learning. PMLR, pp. 6861–6871.

Wu, S., Tang, Y., Zhu, Y., Wang, L., Xie, X., Tan, T., 2019b. Session-based recommendation with graph neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33 (01) pp. 346–353.

Xia, X., Lo, D., Shihab, E., Wang, X., Yang, X., 2015. Elblocker: Predicting blocking bugs with ensemble imbalance learning. Inf. Softw. Technol. 61, 93–106.

Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J., Tang, Y., 2019. LDFR: Learning deep feature representation for software defect prediction. J. Syst. Softw. 158, 110402.

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

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

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

Yang, Q., Ji, H., Lu, H., Zhang, Z., 2021. Prediction of liquid chromatographic retention time with graph neural networks to assist in small molecule identification. Anal. Chem. 93 (4), 2200–2206.

Yao, L., Mao, C., Luo, Y., 2019. Graph convolutional networks for text classification. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence, Vol. 33 (01) pp. 7370–7377.

Yin, R., Li, K., Zhang, G., Lu, J., 2019. A deeper graph neural network for recommender systems. Knowl.-Based Syst. 185, 105020.

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

Yu, D., Wang, H., Chen, P., Wei, Z., 2014. Mixed pooling for convolutional neural networks. In: International Conference on Rough Sets and Knowledge Technology. Springer, pp. 364–375.

Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., Di Penta, M., 2017. Recommending when design technical debt should be self-admitted. In: Proceedings of The 33th 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 216–226.

Zazworka, N., Spínola, R.O., Vetro', A., Shull, F., Seaman, C., 2013. A case study on effectively identifying technical debt. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, pp. 42–47.

Zhang, Y., Yu, X., Cui, Z., Wu, S., Wen, Z., Wang, L., 2020. Every document owns its structure: Inductive text classification via graph neural networks. arXiv preprint arXiv:2004.13826.

Zhao, K., Liu, J., Xu, Z., Li, L., Yan, M., Yu, J., Zhou, Y., 2021a. Predicting crash fault residence via simplified deep forest based on a reduced feature set. arXiv preprint arXiv:2104.01768.

Zhao, K., Xu, Z., Yan, M., Zhang, T., Yang, D., Li, W., 2021b. A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models. Inf. Softw. Technol. 106652.

Zhao, K., Xu, Z., Zhang, T., Tang, Y., Yan, M., 2021c. Simplified deep forest model based just-in-time defect prediction for android mobile apps. IEEE Trans. Reliab..

Jiaojiao Yu is currently a Ph.D. student at the School of Computer Science, Wuhan University. Her research interest includes software engineering, deep learning, and natural language processing.

Kunsong Zhao is currently a master student at the School of Computer Science, Wuhan University. He received the B.S. degree at School of Computer Science and Information Engineering, Hubei University in 2019. His research interest includes software engineering, deep learning, and natural language processing.

Jin Liu received his Ph.D. degree from Wuhan University, China, in 2005. He is now a full professor in School of Computer Science, Wuhan University. His main research interests include machine learning and data mining. He has published more than 60 papers in well-known conferences and journals.

Xiao Liu received his Ph.D. degree in computer science and software engineering from the Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia, in 2011. He was an associate professor at the Software Engineering Institute, East China Normal University, Shanghai, China during 2013 to 2015. He is currently an associate professor with the School of Information Technology, Deakin University, Melbourne. His research interests include workflow systems, cloud and edge computing, big data analytics, and human-centric software engineering.

Zhou Xu is an assistant professor in the School of Big Data and Software Engineering at Chongqing University, China. He received two Ph.D. degrees from Wuhan University (Wuhan, China) and The Hong Kong Polytechnic University (Hong Kong, China) in 2019 and 2021, respectively. His research interests include software defect prediction, empirical software engineering, feature engineering, and data mining.

Xin Wang received the M.S. degree in computer technology from Yunnan University in 2019. He is currently a Ph.D. candidate in computer science at Wuhan University. He has co-authored more than 10 papers, and published in top venues such as ASE, ICWS, WWWJ. His current research interests mainly include service computing, software engineering and recommender systems.