Detecting and Explaining Self-Admitted Technical Debts with Attention-based Neural Networks

Xin Wang
School of Computer Science, Wuhan University
Wuhan, China
xinwang0920@whu.edu.cn

Jin Liu*
School of Computer Science, Wuhan University
Wuhan, China
jinliu@whu.edu.cn

Li Li
Faculty of Information Technology, Monash University
Melbourne, Australia
Li.Li@monash.edu

Xiao Chen
Faculty of Information Technology, Monash University
Melbourne, Australia
xiao.chen@monash.edu

Xiao Liu
School of Information Technology, Deakin University
Geelong, Australia
xiao.liu@deakin.edu.au

Hao Wu
School of Information Science and Engineering, Yunnan University
Kunming, China
haowu@ynu.edu.cn

ABSTRACT

Self-Admitted Technical Debt (SATD) is a sub-type of technical debt. It is introduced to represent such technical debts that are intentionally introduced by developers in the process of software development. While being able to gain short-term benefits, the introduction of SATDs often requires to be paid back later with a higher cost, e.g., introducing bugs to the software or increasing the complexity of the software.

To cope with these issues, our community has proposed various machine learning-based approaches to detect SATDs. These approaches, however, are either not generic that usually require manual feature engineering efforts or do not provide promising means to explain the predicted outcomes. To that end, we propose to the community a novel approach, namely *HATD* (**H**ybrid **A**ttention-based method for self-admitted **T**echnical **D**ebt detection), to detect and explain SATDs using attention-based neural networks. Through extensive experiments on 445,365 comments in 20 projects, we show that *HATD* is effective in detecting SATDs on both in-the-lab and in-the-wild datasets under both within-project and cross-project settings. *HATD* also outperforms the state-of-the-art approaches in detecting and explaining SATDs.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; **Software maintenance tools**.

KEYWORDS

Self-Admitted Technical Debt, Word Embedding, Attention-based Neural Networks, SATD

ACM Reference Format:
Xin Wang, Jin Liu, Li Li, Xiao Chen, Xiao Liu, and Hao Wu. 2020. Detecting and Explaining Self-Admitted Technical Debts with Attention-based Neural Networks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416583

1 INTRODUCTION

Technical debt is a metaphor, proposed by Cunningham [6], used to denote a suboptimal solution that developers take shortcuts to achieve rapid delivery during development. Some typical examples of technical debts include introducing hard-coded values into the code, making code changes while ignoring failing unit tests, copy-pasting code from other modules, etc. Those technical debts are usually introduced to a software system unintentionally by developers, or intentionally but without adequately documented. As time goes by, those undocumented technical debts, if not resolved in time, might be faded away from the developers' minds and become equivalent to unintentionally introduced technical debts [2, 45]. To mitigate this issue, developers may decide to document (via comments) their intentionally introduced technical debts. The documented debt is often known as self-admitted technical debt (SATD in short).

No matter intentionally or unintentionally, the compromises made to introduce technical debts will likely lead to negative effects on the maintenance of the software in the long run [18, 21, 23, 27, 35, 53]. Indeed, as argued by Megan Horn[1], technical debts may cause software systems to behave unexpectedly, and those surprising behaviors could be notoriously difficult to test and fix. Often, fixing one issue could introduce several new issues, resulting in high costs for resolving technical debts. As revealed in a recent study by CAST Software[2], a provider of software analysis and measurement tools, the amount of technical debts that have to be addressed after the application is deployed in production, on average, is over one million for each business application.

*Jin Liu is corresponding author.

[1] https://www.farreachinc.com/blog/far-reach/2017/10/05/the-true-cost-of-technical-debt

[2] https://www.itbusinessedge.com/slideshows/show.aspx?c=93589

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416583

Table 1: Examples of SATD comments.

Project	SATD Comments
Apache Ant	// TODO: allow user to request the system or no parent
	// What is the property supposed to be?
	// cannot remove underscores due to protected visibility >:(
JFreeChart	// do we need to update the crosshair values?
	// defer argument checking...
	// do we need to update the crosshair values?
JMeter	// Don't try restoring the URL TODO: wy not?
	// Can be null (not sure why)
	// Maybe move to vector if MT problems occur
SQuirrel	// Do we need this one.
	// is this right???
	// verify this
ArgoUML	// Why does the next part not work?
	// Shouldn't we throw an exception here?!?!
	// The following can be removed if selectAll gets fixed

Despite the fact that technical debts will introduce negative impacts to the maintenance of software, technical debt is still widespread and seems to be unavoidable in software systems[3, 12, 47, 54]. Hence, there is a strong need to invent automated approaches to detect technical debts and fix them as earlier as possible. Nevertheless, it is a challenging endeavor to handle all the types of technical debts at once [25, 38]. As the initial step towards achieving such a purpose, many state-of-the-art works start by focusing on the detection of SATDs [10, 14, 17, 58]. Unlike other technical debts, SATDs are often highlighted as comments in the code of the software [30, 41, 59]. Table 1 enumerates some SATD examples identified in popular open-source projects. For example, the comment *"May be replaced later"* in the JMeter project indicates that the current code is a temporary solution and needs to be replaced in the future.

Although explicitly highlighted, as shown in Table 1, SATDs are still largely presented in open-source software projects. Our community has hence spent various efforts to experimentally characterize SATDs. For example, Potdar et al. [38] attempt to manually explore SATDs in code comments of Java projects and eventually summarize 62 patterns that can be used to identify SATD. Huang et al. [16] have proposed a text-mining method to identify SATD and have achieved substantial improvements compared with pattern-based methods. Yan et al. [56] propose a change-level method for detecting SATDs utilizing 25 software change features.

Unfortunately, existing state-of-the-art approaches are either involving heavy manual pre-processes (such as identifying the SATD-relevant patterns or extracting features to fulfill ML-based classifiers) or lacking reliable explanations to elucidate why a SATD is flagged as such[4, 43, 50, 51, 60]. To fill this gap, we propose to the community a novel deep learning-based approach to detect and explain self-admitted technical debts in open-source software projects. We design and implement a prototype tool called *HATD*. First, it leverages positional encoder and Bi-directional Long Short-Term Memory(Bi-LSTM)[64] network to capture the sequential characteristics of SATDs from code comments. Then, it leverages diverse attention mechanisms to highlight the importance of the automatically prepared features that have contributed to the detection of SATDs. The most important features will subsequently be leveraged to explain the classification result, e.g., why a comment is (not) flagged as SATD? Through extensive experiments on 445,365 comments in 20 projects, *HATD* demonstrates its superior performance and explainability in both within-project and cross-project SATD detections over the state-of-the-art methods.

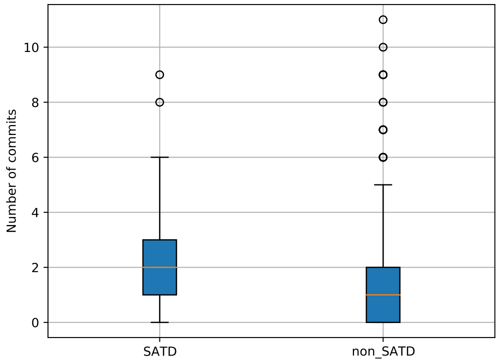

Figure 1: Distribution of changes made to SATD-involved and non-SATD-involved source code files.

The main contributions of this paper can be summarized as follows:

- We provide an overview of code comment characteristics that could be challenging for creating automated tools to detect SATDs in software projects.
- We design and implement a prototype tool called *HATD*, which leverages a hybrid attention-based method combining both single-head attention and multi-head attention mechanisms for pinpointing SATDs from code comments.
- We demonstrate the effectiveness of *HATD* by comparing with state-of-the-art methods on open-source benchmark projects, detecting SATDs in real-world and popular software projects, and providing explanations to the classification results.

2 MOTIVATION AND BACKGROUND

To help readers better understand our work and motivate the necessity of developing automated approaches for pinpointing SATDs, we first present in this section a simple empirical investigation to check whether SATDs impact the maintainability of software projects. Then, we enumerate some characteristics of code comments that make it challenging to automatically interpret the semantics of the comments (i.e., pinpointing self-admitted technical debts).

2.1 SATD VS. Code Changes

Intuitively, SATDs will trigger more future code changes, i.e., increasing the maintainability of the software project [39, 55]. To quantify the relationship between SATDs and code changes, we conduct a simple empirical study (count the number of commits related to files that have been changed, with and without technical debts) to check if SATDs will trigger more future code changes. To this end, we select a popular open-source project called Apache Ant to fulfill the experiments. The reason why this project is chosen is that the SATDs introduced over the development of this project are known. Indeed, as of Jan 14, 2016, the Apache Ant project has introduced in total 131 SATDs, accounting for roughly 3.2% of total comments (cf. 4,137).

In order to explore the long-term impact of SATDs on the maintenance of the project, we set a cutting point on Jan 1, 2014 and recount the SATDs introduced at that point. Among 131 SATDs, 111 of them have already been introduced into the project and have not been resolved, involving 82 Java files. In other words, at this point, 1,116 Java files are not involved with SATDs. Starting from this cutting point, we revisit all upcoming commits and count the times of changes related to each Java file. The final results are plotted in Figure 1, which illustrates the distribution of changes made to SATD-involved and non-SATD-involved Java files. It is quite clear that SATD-involved files require more changes compared to the files without involving SATDs. This difference is further confirmed by a Mann-Whitney-Wilcoxon (MWW) test, i.e., the difference is significant at a significance level[3] of 0.001. This simple empirical investigation shows that SATDs indeed negatively impact the maintenance of software projects, and hence there is a strong need to invent promising approaches to detect and resolve them [13].

2.2 Characteristics of Code Comments

Code comments are often provided by developers to remind themselves to record unsatisfactory design decisions, question design decisions of other developers, and make future changes. Since code comments are usually written in natural language by different developers with different writing styles, they may come with characteristics that are difficult to interpret [31]. In this section, we summarize some of the characteristics of code comments that make automated detection of SATDs challenging:

(1) **Domain Semantics.** Code comments often contain jargons, method names, and abstract concepts that only programmers can understand. For example, the comment *"FIXME: I use a for block to implement END node because we need a proc which captures"* in JRuby project contains a term *for*, an abstract concept *END node*.
(2) **Level of details (Comment Length).** The length of phrases in code comments can vary widely, ranging from one word like "unused", "load", to short phrases like "call workaround", "should probably error" and to sentences like "this is such a bad way of doing it, but ...". This varying length text feature in code comments makes it challenging as well to detect SATDs.
(3) **Project Uniqueness.** SATD comments on different projects will show different styles due to different developers, as empirically confirmed by Potdar et al. [38] in their manual pattern summarization work. In addition, the coding habits and vocabulary gaps among different developers aggravate this phenomenon.
(4) **Polysemy.** Polysemy is common in sentence expressions. It is often accompanied by domain knowledge in code comments. For example, code comments may contain method names. It is challenging to distinguish method names and common words, which often requires contextual information.
(5) **Abbreviation.** To facilitate memorization and writing efficiency, developers often use abbreviations in code comments. Project differences often exacerbate this phenomenon. For example, the comment *"no default in DB. If nullable, use null."* in project SQuirrel, in which *DB* is the abbreviation of *Database*.

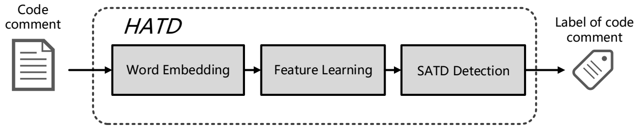

Figure 2: The overall architecture of our approach.

These characteristics exhibited frequently in SATD comments make traditional pattern-based and text-mining-based SATD detection approaches difficult to achieve high performance. Therefore, there is a strong need to mine the in-depth implicit and semantic features from the code comments so as to satisfy the different challenges in effectively detecting SATDs.

3 OUR APPROACH

We propose a **H**ybrid **A**ttention-based method for self-admitted **T**echnical **D**ebt detection named *HATD*. *HATD* analyzes code comments of different projects or application domains (Challenge 1 and 3) aiming to learn comprehensive implicit features of SATDs. It can flexibly extract and aggregate variable-length text features of SATD detections (Challenge 2). To overcome polysemy and abbreviation issues (Challenge 4 and 5), we use ELMo to obtain dynamic word embedding across contexts for SATD detections.

As depicted in Figure 2, *HATD* has three modules. The word embedding module maps the words in the comments to vectors. Different word embedding techniques can be incorporated into our approach (by default, the ELMo algorithm is implemented). In the feature learning module, we use positional encoder and Bi-LSTM to learn sequential features, then we leverage diverse attention mechanisms to further obtain potential representation of words and distinguish the importance of different words. Finally, the learned features are fed into the detection module to predict SATDs.

3.1 Word Embedding

Word embedding is a collective term for language models and representation learning techniques in natural language processing (NLP). Conceptually, it refers to embedding a high-dimensional vector whose dimension is the number of all words into a vector of a much lower dimension. The goal of word embedding module is to overcome the aforementioned challenges (cf. Section 2.2) in handling code comments. Let us consider the example of keyword *proc* from code comment *"FIXME: I use a for block to implement END node because we need a proc which captures"* in JRuby project. The *proc* here is the abbreviation of *process*, which may be seen as OOV (out of vocabulary). Because the labeled code comment is limited, and we cannot guarantee that any test word exists in training corpus. According to morphological knowledge, FastText[5, 19] solves the OOV problem to some extent by learning character-based embedding representation of subwords. Static word embedding techniques cannot deal with the phenomenon of polysemy, which is common in comments. Dynamic word representation technique such as ELMo[37] can deal with this problem to some extent by adjusting the word vector according to the context. ELMo (Embedding from Language Models)[37] is a new type of deep contextualized word representation language model, which can model the complex features of words (such as syntax and semantics) and the change of words across linguistic contexts (i.e., to polysemy). Since the

[3] Given a significance level $\alpha = 0.001$, if p-value < α, there is one chance in a thousand that the difference between the datasets is due to a coincidence.

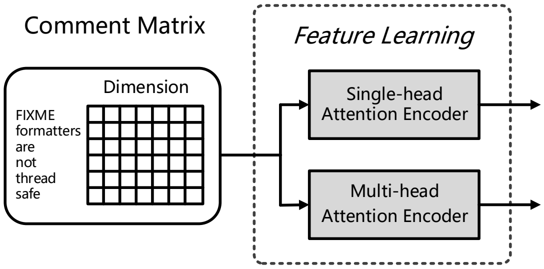

Figure 3: The working process of the feature learning module.

meaning of words is context-sensitive, ELMo's principle is to input sentences into a pre-trained language model in real time to get dynamic word vectors, which can effectively deal with polysemy. Considering that the main application scenario of ELMo is to provide dynamic word vectors, acting on downstream tasks, our model integrates ELMo to identify the SATDs. Detailed comparison of existing word embedding schemes can be found in Section 5.3.

3.2 Feature Learning

The feature learning module consists of two sub-modules, a single-head attention encoder (SAE) and a multi-head attention encoder (MAE), adopting different attention mechanisms and sequence feature extractors. The single-head attention encoder first inputs the embedding representation of the comment into a Bi-LSTM network, then we can use an attention mechanism to assign different weights to different words according to the importance of words for the purpose of interpreting detection results. In order to further encode the current word, the multi-head attention encoder allows the encoding of words considering other words information via the so-called self-attention mechanism. By combining the SAE and MAE modules, we hope to exploit the strengths of the RNN-based model[64] and the Transformer-based model[48] to improve SATD detection performance and interpret the detection results with attention mechanism. Figure 3 shows the internal structure of the feature learning module. The details of these two internal sub-modules will be given later.

3.2.1 Single-head Attention Encoder (SAE). Given a code comment, SAE module firstly learns the features of the comment from the preceding as well as the following tokens with a two-layer Bi-LSTM network. In this process, we can exploit an attention mechanism to assign different attentions to words for the purpose of interpreting detection results. Figure 4 shows the working process of SAE, which is carried out in two steps: (1) Extracting sequential features of the comment, and (2) Assigning attention weight to each word. S^1 denotes the obtained implicit feature vector of the comment.

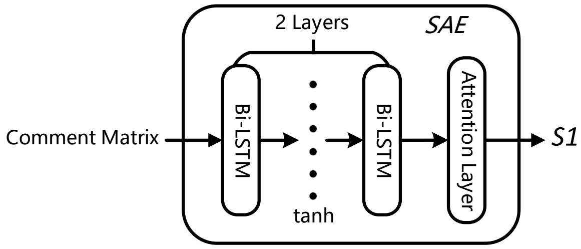

Figure 4: Workflow of the SAE module.

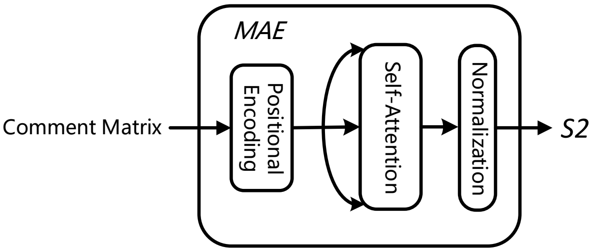

Figure 5: Workflow of the MAE module.

Word Encoder. Given a code comment, we assume that the comment contains l words. x_t represents the tth word's vector. We use a Bi-LSTM to get implicit representation of words by summarizing information from both directions for words. The Bi-LSTM contains the forward LSTM which reads the comment c from x_1 to x_l and a backward LSTM which reads from x_l to x_1:

$$h_t = \overrightarrow{LSTM}(x_t), t \in [1, l], \tag{1}$$

$$h'_t = \overleftarrow{LSTM}(x_t), t \in [1, l]. \tag{2}$$

We obtain the implicit representation of a given word x_t by concatenating the forward hidded state h_t and back hidden state h'_t, i.e., $[h_t, h'_t]$.

Assigning Attention Weights. In a code comment, each word contributes differently on SATD detection. For example, Potdar et al.[38] manually summarized 62 patterns for detecting SATDs. Considering the evolution of the project and gaps between developers, the manually summarized patterns lack generalizability and adaptability to newly generated code comments. Based on these above considerations, we exploit an attention mechanism to distinguish contributions of different words and aggregate the representation of those informative words to form a sentence vector.

$$u_t = tanh(W_w[h_t, h'_t] + b_w) \tag{3}$$

$$a^t = softmax(u_t^T * u_w) \tag{4}$$

$$S^1 = \sum_t a_t[h_t, h'_t] \tag{5}$$

We firstly feed word representation $[h_t, h'_t]$ through one-layer perception to get u_t as a hidden representation of $[h_t, h'_t]$, then we measure the importance of the word u_t with a word level context vector u_w and get a normalized importance weight a_t through a softmax function. Finally, We add all the word vectors with attention weight as the final feature representation (S^1) of the code comment.

3.2.2 Multi-head Attention Encoder(MAE). The purpose of MAE is to focus on all the words in the entire input sequence to help the model better encode the current word. Figure 5 illustrates the working process of MAE in two steps: (1) Encoding positional information of words, and (2) Encoding words considering other words information. S^2 denotes the obtained implicit feature vector of the comment.

Capturing Positional Information. To learn the order of the words in the input sequence, we firstly add a positional vector following the one proposed by Vaswani et al. [48], which follows the specific pattern that the model learns. Then we perform a linear transformation on the result. The propagation rule is defined by the following equation:

$$p_t^{(i)} = \begin{cases} sin(t/10000^{2i/d}) & if\ i = 2k, \\ cos(t/10000^{2i/d}) & if\ i = 2k+1, \end{cases} \tag{6}$$

$$x_t = (x_t + p_t)W_p + b_p \tag{7}$$

where t is the position, i is the dimension and d is the dimension of the produce output.

Assigning Self-Attention. Given a code comment, the representation of a word is closely related to the other words in the comment. To obtain further word vector representation, we introduce self-attention mechanism[48] to better encode the words. The details are given by the following equations:

$$Q_j = XW_q \tag{8}$$

$$K_j = XW_k \tag{9}$$

$$V_j = XW_v \tag{10}$$

$$head_j = Softmax(\frac{Q_j K_j^T}{\sqrt{d_k}})V_j \tag{11}$$

$$S^2 = Norm(Cat(head_1, ..., head_h)W^o) \tag{12}$$

Here, $X \in \mathbb{R}^{l \times d}$ is the word vector matrix. We compute the attention function on a set of queries simultaneously, and pack them together into a matrix Q. The keys and values are also packed together into matrices K and V [48]. $W_q \in \mathbb{R}^{d \times d_q}$,$W_k \in \mathbb{R}^{d \times d_k}$ and $W_v \in \mathbb{R}^{d \times d_v}$ are three learnable weight matrices. We use eight attention heads ($h = 8$ by default) to expand the model's ability to focus on different words. For each head we have $d_q = d_k = d_v = d/h$, $W^o \in \mathbb{R}^{hd_v \times d}$. After concatenating the embedding vectors from all the heads, we introduce layer normalization for fixing the mean and the variance of the accumulated inputs.

3.3 SATD Detection

Finally, as is shown in Figure 6, S^1 and S^2 are concatenated to form an abstract representation of the code comment, and then input into a fully connected layer f_{fc} for SATD detection. Prediction results are calculated by a *sigmoid* function. The classification rule is defined by the following equations:

$$\hat{Y} = Sigmoid(f_{fc}(S^1 \bigoplus S^2)) \tag{13}$$

Here, $\bigoplus$ denotes the concatenate operation, and $\hat{Y}$ denotes the prediction result of SATD detection.

As shown in Table 2, SATD and non-SATD comments have a very imbalanced distribution in the open-source projects [20]. To address the data imbalance problem, instead of downsampling or applying feature selection that may sacrifice the feature learning capability, we introduce a weighted cross-entropy loss as the loss function. Suppose there are a SATD comments and b non-SATD comments in the training data, then the loss function is defined as:

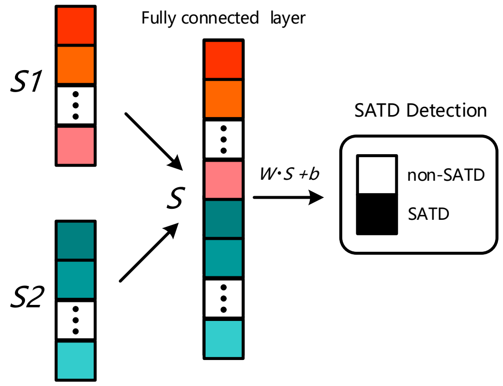

Figure 6: Workflow of module combination and SATD detection.

$$weights_s = \begin{cases} \dfrac{b}{a+b} & \text{if } s\text{th class is SATD} \\ \dfrac{a}{a+b} & \text{otherwise} \end{cases} \tag{14}$$

$$Loss(\hat{Y}, T) = -(weights_s) \sum_s T_s log(\hat{Y}_s) \tag{15}$$

Here, T and $\hat{Y}$ are the groundtruth label and predicted label of the comment, respectively. All the parameters in the model are trained via gradient descent.

4 EXPERIMENTAL SETUP

In this section, we have conducted comprehensive experiments on SATD detections aiming to answer the following research questions:

- RQ1: Is *HATD* effective in detecting self-admitted technical debts in software projects?
- RQ2: How does *HATD* compare with the state-of-the-art approaches for recognizing self-admitted technical debts?
- RQ3: How effective is the word embedding technique integrated into *HATD* compared with other baseline techniques?
- RQ4: To what extent are self-admitted technical debts existing in real-world software projects?

4.1 Dataset Description

In our experiments, we use the dataset provided by [7]. This dataset contains labeled code comments for ten open-source projects, including ApacheAnt, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and SQuirrel. The detailed descriptions of the dataset are shown in Table 2. These ten projects belong to different application areas and have different numbers of contributors, comments as well as scale and complexity. We can see that only a small ratio of SATD comments in each project.

4.2 Evaluation Methods and Metrics

In this work, we adopt three commonly used metrics, namely Precision, Recall, and F1-score (the balanced view between Precision and Recall), to evaluate the performance of our approach.

Table 2: Statistics of SATD comments in the benchmark software projects.

Project	Description	Release	Contributions	Comments	SATD	%of SATD
Apache Ant	Java library and Command-line Tool	1.7.0	74	4,137	131	3.17
ArgoUML	UML Modeling Tool	0.34	87	9,548	1,413	14.80
Columba	E-mail Client	1.4	9	6,478	204	3.15
EMF	Eclipse Model Driven Architecture	2.4.1	30	4,401	104	2.36
Hibernate	ORM Framework	3.3.2	226	2,968	472	15.90
JEdit	Text Editor	4.2	57	10,322	256	2.48
JFreeChart	Char library	1.0.19	19	4,423	209	4.73
JMeter	Performance Tester	2.10	33	8,162	374	4.58
JRuby	Ruby Interpreter	1.4.0	328	4,897	662	13.52
SQuirrel	SQL Client	3.0.3	46	7,230	285	3.94
Average			91	6,257	411	6.57

$$Precision = \frac{TP}{TP + FP} \tag{16}$$

$$Recall = \frac{TP}{TP + FN} \tag{17}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{18}$$

Here, TP (true positive) means the number of SATD comments that are detected as such; FP (false positive) means the number of non-SATD comments that are detected as SATD; and FN(false negative) means the number of SATD comments that are detected as non-SATD.

4.3 Parameters and Experimental Environment

In the Bi-LSTM component of our model, the number of layers is two, the number of hidden units is set to be 128, two branches produce outputs with 256 units, respectively. The adam optimizer with default setting (except that learning rate is configured to be 1e-4) is used for training. In order to avoid overfitting, we add a dropout layer between every two layers, with a drop probability of 0.5. We set L_2 loss weight as 5e-4. Due to the unbalanced distribution of SATD and non-SATD comments, we introduce a weighted cross-entropy loss as the loss function. We compute the mean values of the Precision, Recall and F1-score across ten experiments to estimate the performance of models. The experiments are run on a Linux system (Ubuntu 16.04 LTS) with 64GB memory and a RTX2080Ti GPU, implemented by leveraging the deep learning library PyTorch[4].

5 RESULT

We now present our experimental results towards answering the aforementioned research questions.

5.1 RQ1: Effectiveness of *HATD*

Towards evaluating the effectiveness of our approach, we apply *HATD* to study the ten projects maintained by [7], since these ten projects have been provided with ground truth. Based on these ten projects, we evaluate the performance of *HATD* from two aspects:

- **Within-project.** For each project, we perform 10-fold cross-validation to assess the classification capability of *HATD*. In particular, we first equally divide the project's comments into ten subsets. Then, we train our model with nine subsets of comments and leverage it to classify the remaining subset of comments. We repeat this experiment 10 times to ensure that each subset has been considered as a testing set once. The final performance is reported as the average score of the ten rounds of experiments.
- **Cross-projects.** In this aspect, we also conduct our experiments in ten rounds, with one project tested in each round. Specifically, for each project, we train our model based on the other nine projects and leverage it to predict all the comments of the project under testing. The classification results will be directly attached to the testing project, for the sake of simplicity, despite that the classification models are trained based on the other nine projects.

Table 3: The performance of our approach over both within-project and cross-projects SATD detections.

Project	Within-Project			Cross-Projects		
	Precision	Recall	F1	Precision	Recall	F1
Apache Ant	0.7050	0.6108	0.6545	0.6569	0.7801	0.7132
ArgoUML	0.8731	0.9469	0.9085	0.8615	0.9487	0.9030
Columba	0.9434	**0.9616**	**0.9524**	0.8906	**0.9598**	**0.9239**
EMF	0.7475	0.8586	0.7992	0.6769	0.8787	0.7647
Hibernate	0.8866	0.9536	0.9189	0.8699	0.9305	0.8992
JEdit	0.5468	0.6554	0.5962	0.7783	0.8797	0.8259
JFreeChart	0.9268	0.8747	0.9000	0.6897	0.7135	0.7014
JMeter	0.9249	0.8938	0.9091	0.8178	0.8715	0.8438
JRuby	**0.9568**	0.9388	0.9477	**0.8977**	0.9397	0.9182
SQuirrel	0.7978	0.8769	0.8355	0.7814	0.8318	0.8058
Average	0.8309	0.8571	0.8422	0.7921	0.8734	0.8299

Table 3 presents the experimental results of our approach w.r.t. Precision, Recall, and F1-score metrics. The best and worst results are respectively highlighted in bold and underline, and the difference is quite significant. This finding shows that the our approach might be sensitive to the training dataset. Indeed, our approach yields relatively poor performance for the JEdit project because many of its comments are short and sometimes meaningless (e.g., *"Inner classes"* or *"unsplit() method"*), resulting in noises or a lack of effective semantic information to be learned by our approach. In addition, the fact that some projects achieved better performance in the within-project setting (e.g., JFreeChart) while others achieved better performance in the cross-projects setting (e.g., JEdit) further confirms that the quality of the training set is important to our approach.

[4] https://pytorch.org/

Nevertheless, the experimental results show that our approach generally yields good performance for most of the projects concerning both within-project and cross-projects settings. Indeed, on average, our approach achieves a Precision, Recall, and F1-score of 83.09%, 85.71%, and 84.22% respectively for within-project SATD detection, and a Precision, Recall, and F1-score of 79.21%, 87.34%, F1-score of 82.99% respectively for cross-projects SATD detection. It is worth mentioning that our approach does not perform significant differences between within-project and cross-projects settings. This evidence suggests that our approach could be effective and practical for pinpointing SATDs in real-world software projects.

> Answer to RQ1: Our approach can achieve high-performance for both within-project and cross-projects settings and hence is effective and practical to be used to pinpoint SATDs for real-world software projects.

5.2 RQ2: Performance Comparison with the state-of-the-art Approach

Table 4: The performance of our appraoch compared to the state-of-the-art.

Projects	With-Project			Cross-Project		
	Ren’s	Ours	Gains	Ren’s	Ours	Gains
Apache Ant	0.5977	0.6545	+9.50%	0.6260	0.7132	+13.93%
ArgoUML	0.8895	0.9085	+2.14%	0.8409	0.9030	+7.38%
Columba	0.8545	0.9524	+11.46%	0.7782	0.9239	+18.72%
EMF	0.7010	0.7992	+14.01%	0.5797	0.7647	+31.91%
Hibernate	0.8465	0.9189	+8.55%	0.8035	0.8992	+11.91%
JEdit	0.5564	0.5962	+7.15%	0.5760	0.8259	+43.39%
JFreeChart	0.7759	0.9000	+15.99%	0.6504	0.7014	+7.84%
JMeter	0.8858	0.9091	+2.63%	0.7726	0.8438	+9.22%
JRuby	0.8626	0.9477	+9.87%	0.8445	0.9182	+8.73%
SQuirrel	0.7467	0.8355	+11.89%	0.7069	0.8058	+13.99%
Average	0.7717	0.8422	+9.14%	0.7179	0.8299	+15.61%

We now compare our approach with the state-of-the-art approach proposed by Ren et al.[39]. To the best of our knowledge, this is the closest work to ours since both approaches are implemented based on deep learning to detect SATDs. Their method leverages basic neural networks to accomplish its purposes while our approach brings *attention mechanism* into neural networks to achieve our objectives. We compare our approach with their method on the same dataset, i.e., the ten projects with ground truth.

Table 4 summarizes the comparison results. In within-project SATD detection, compared with Ren’s method, our method achieves performance improvements on all projects ranging from 2.14% to 15.99%. Especially, in the EMF project, which only contains 104 comments, our method gets a 14.01% improvement. This evidence suggests that our approach is more resilient to small training corpora. On average, our method obtains an F1-score of 84.22%, which is 9.14% higher than that of Ren’s method.

In cross-projects SATD detection, our method achieves even better performance improvements ranging from 7.38% to 43.39%, compared to the within-project setting. On overage, for the ten projects, our method achieves an F1-score of 82.99%, which is 15.61% higher than that of Ren’s method. It is worth highlighting that for the JEdit project, our method achieves 43.39% improvement. The performance improvement for the EMF project also exceeds 30%. These experimental results demonstrate that our approach can learn more useful information from other projects than Ren’s method. Hence, our approach is more practical to be applied to detect SATDs in real-world software projects.

Finally, despite that, we have only compared our approach with the state-of-the-art Ren’s approach, our approach should also outperform other similar ones targeting SATD extractions or classifications. Indeed, as experimentally demonstrated by Ren et al., over the same dataset, their method can outperform three baseline approaches, including a pattern-based SATD extraction approach, a traditional text-mining based SATD classification approach, and Kim’s simple CNN-based sentence classification approach. Therefore, since our approach can outperform Ren’s method in the same regard, we believe that our approach should also be able to outperform the aforementioned state-of-the-art approaches.

> Answer to RQ2: *HATD* outperforms the state-of-the-art approaches, including neural-network-based ones such as Ren’s approach and pattern-based ones, as well as traditional text-mining based approaches.

5.3 RQ3: Effectiveness of the selected word embedding technique

Word embedding, allowing to embed a high-dimensional space with the number of all words into a continuous vector space with a much lower dimension, has become one of the most important techniques in the natural language processing (NLP) community. Apart from the ELMo technique adopted in this work, the NLP community has invented various other methods to realize word embedding. Towards answering the third research question, we conduct a comparative study between ELMo and some of the other methods to justify our selection of ELMo and its effectiveness.

To the best of our knowledge, there are at least four additional popular word embedding techniques available in the community that we can compare with.

One-hot is the simplest encoding method. Each word has its own unique word vector representation. The dimension of the word vector is the length of vocabulary. Only one position in each word vector has a value of 1, and the others are 0. One-hot encoding is to assume that all words are independent of each other. It ignores the semantic similarity between words and causes severe data sparsity problems when the corpus is very large.

Word2Vec[34] is an open-source toolkit for generating word vectors launched by Google in 2013, containing two traning modes: skip-gram and continuous bag of words (CBOW). The skip-gram model is concerned with using one word to predict the surrounding words, while CBOW model is concerned with using the surrounding words to predict the central word. In order to improve model accuracy and training speed, Word2Vec usually uses negative sampling and hierarchical softmax. Word vectors generated by Word2Vec can better express the similarity and analogy relationship between words.

Table 5: The comparison results of ELMo and other state-of-the-art word embedding techniques.

Projects	**Within-Project**					**Cross-Projects**				
	ELMo	One-Hot	Word2Vec	GloVe	FastText	ELMo	One-Hot	Word2Vec	GloVe	FastText
Apache Ant	**0.6545**	0.6263	0.6242	0.5576	0.5825	**0.7132**	0.6308	0.6783	0.6226	0.6316
ArgoUML	0.9085	**0.9132**	0.9059	0.8967	0.8918	**0.9030**	0.8487	0.8440	0.8290	0.8400
Columba	**0.9524**	0.9467	0.9024	0.9424	0.9434	**0.9239**	0.8337	0.8685	0.8469	0.8718
EMF	**0.7992**	0.7826	0.7273	0.7556	0.7506	**0.7647**	0.6043	0.6435	0.6376	0.6818
Hibernate	0.9189	0.9126	0.9293	0.8871	**0.9592**	**0.8992**	0.8257	0.8331	0.8261	0.8242
JEdit	**0.5962**	0.5862	0.5714	0.5818	0.5622	**0.8259**	0.6475	0.6013	0.6118	0.6225
JFreeChart	**0.9000**	0.8837	0.8947	0.8421	0.8660	0.7014	0.6533	0.6645	0.6095	**0.7214**
JMeter	**0.9091**	0.8608	0.8718	0.8741	0.8721	**0.8438**	0.7451	0.7601	0.7537	0.7659
JRuby	0.9477	0.9000	0.9091	0.8841	**0.9524**	**0.9182**	0.8726	0.8548	0.8834	0.8626
SQuirrel	**0.8355**	0.8214	0.7719	0.7970	0.8000	**0.8058**	0.7070	0.7559	0.7299	0.6872
Average	**0.8422**	0.8234	0.8108	0.8019	0.8190	**0.8299**	0.7369	0.7504	0.7351	0.7509

GloVe (Global Vectors for Word Representation) [36] is essentially a log-bilinear model with a weighted least-squares objective. The model is inspired by the word co-occurrence probability that may encode global information for words. This just makes up for the weakness of Word2Vec just using local word co-occurrence information. Each word of GloVe involves two word vectors, one is the vector of the word itself, and the other is the context vector of the word. The final representation of the word vector is obtained by adding the two vectors.

FastText[5, 19] can be regarded as a derivative of Word2Vec, which performs word vector training based on the knowledge of language morphology. For each word entered, a word-based n-gram representation is performed, and then all n-grams are added to the original word to represent morphological information. The advantage of this method is that in english words, the morphological similarity of prefixes or suffixes can be used to establish relationships between words.

To conduct a fair comparison, we respectively reimplement the word embedding module with one of the aforementioned four techniques and keep the remaining implementation of *HATD* unchanged. We then launch the new version of *HATD* on the same dataset, with the same parameters and experimental settings (i.e., within-project and cross-projects). The word embedding size determines the ability of the model to capture the feature information of a code comment. In order to avoid the influence of different word embedding size on the experimental results, we uniformly set the embedding size as 1024 except OneHot embedding method, whose dimension is always equal to the size of the corpus.

Table 5 presents the experimental results for both within-project and cross-projects settings. In the within-project setting, due to the small size of corpus, even OneHot can perform well and even slightly better than other word embedding techniques (Word2Vec, GloVe and FastText). ELMo achieves the best performance for eight projects. This result shows that ELMo is a promising word embedding technique to be used for supporting the automated classification of SATDs. This evidence can be easily observed in the cross-projects experiments as well. Indeed, as also shown in Table 5, for nine out of the ten selected projects, ELMo achieves the best performance in supporting *HATD* pinpoint SATDs. Moreover, as suggested by the average performance for both within-project and cross-projects experiments, at least for supporting SATD classifications, ELMo performs much better than that of other word embedding techniques (i.e., One-Hot, Word2Vec, GloVe, FastText). This can be explained by the fact that ELMo will generate different vector representations for the same word depending on the context. For example, some teams use the term 'rework' to specify routine refactoring, which is not always TD. ELMO can alleviate this situation to some extent.

In the within-project SATD detection, the used training set and testing set come from the same project with similar comment characteristics, due to the same batch of developers. While in the cross-projects SATD detection, the comments of the testing set come from a new project, which is different from the training set. Overall, the performance of cross-project SATD detection is generally lower than the within-project SATD detection to varying degrees in all word embedding techniques.

> Answer to RQ3: ELMo outperforms other word embedding techniques by achieving mostly the best or second-best performance when supporting neural network-based SATD detections. Moreover, under the same context, dynamic word embedding technique seem to be more reliable than static ones.

5.4 RQ4: SATDs in real-world software projects

To evaluate the practicality of *HATD*, in this last research question, we assess to what extent can *HATD* be leveraged to detect SATDs in real-world software projects, which have no ground truth constructed beforehand. In addition, we also evaluate whether *HATD* is applicable to other programming languages, such as Python, JavaScript, etc. To this end, we resort to Github to search for hot projects, and eventually, we select ten software projects, taking different technical fields into consideration. Table 6 summarizes the ten projects, along with their metadatas, such as short descriptions, number of total comments, etc.

We then train our approach based on the ten benchmark apps (as enumerated in Table 2) and apply it to classify the comments of each of the aforementioned ten real-world software projects. The last second column of Table 6 illustrates the experimental results. In total, among 382,799 comments, *HATD* flags 5,438 of them as

Table 6: The list of our selected real-world popular software projects.

Project	Description	Release	Stars	Contributions	Comments	SATD	% of SATD
Spring Boot	A framework for creating Spring based applications.	2.3.0	47.0k	675	67,425	44	0.07
Dubbo	A high-performance, java based, open source RPC framework.	2.7.6	31.9k	297	37,516	129	0.34
OkHttp	Square's meticulous HTTP client for Java and Kotlin.	4.5.0	36.7k	199	2,728	25	0.92
Guava	Google core libraries for Java.	29.0	36.8k	229	102,705	1,398	1.36
Vue	a progressive JavaScript framework for building UI on the web.	2.6.11	162.0k	292	11,721	152	1.30
Werkzeug	The comprehensive WSGI Web Application Library.	1.0.1	5.3k	332	1,575	17	1.08
PyTorch	Tensors and Dynamic neural networks in Python.	1.4.0	37.9k	1,361	21,977	742	3.38
Django	The Web Framework for Perfectionists with Deadlines.	3.0.5	48.7k	1,877	24,074	186	0.77
scikit-learn	A Python Module for Machine Learning.	0.22.2	40.3k	1,634	19,625	288	1.47
TensorFlow	An Open Source Machine Learning Framework for Everyone.	2.2.0	143.0k	2,460	93,453	2,452	2.62
Average			59.0k	936	38,280	543	1.33

SATDs. The ratios of SATDs in all the project's comments are quite low, ranging from 0.7% to 3.38%, giving an average of 1.33%. Compared with the ground truth projects, we observe that there are fewer SATDs in these ten real-world software projects we selected, which may be because they are very popular on Github, and more contributors participate in code commits and maintenances. The average number of code contributors in these software projects is ten times that of the ground truth projects.

Manual verification of the results reported by *HATD*. We randomly select 50 samples each (100 comments in total) from the code comments marked as having technical debts and no technical debt for further analysis. We then invite five programmers to manually judge these 100 selected comments and compare their decisions with the results yielded by our attention-based deep learning model. The five programmers manually flag the 100 comments as SATD or non-SATD independently and then discuss with each other to reach consensus if inconsistent decisions are made in the first place. Among the 50 code comments marked as SATD, 12 of them are not flagged by five programmers as such. Among the 50 code comments marked as non-SATD, only five comments differ from the manual annotations. Overall, our approach yields an accuracy of 83% in detecting SATDs in popular real-world software projects. Considering that the actual projects we selected come from different technical fields and have significant domain differences, some domain characteristics will not be fully covered by our training set (based on the ten benchmark apps). With the increase of training data, we believe that the performance of our model can be further improved.

> Answer to RQ4: In a practical scenario, our approach achieves an accuracy of 83% in detecting SATDs in real-world software projects. The fact that all the selected software projects have more or less involved with SATDs shows that SATD is commonly spread in software projects. Therefore, we argue that software developers and maintainers should pay more attention to SATDs.

6 DISCUSSION

We now discuss the explainability of our approach to SATD detection and introduce potential threats to validity of this study.

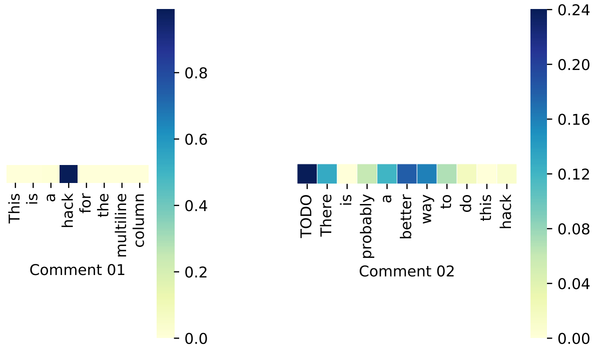

Figure 7: Visualization of weights for two code comments (Comment 01: This is a hack for the multiline column; Comment 02: TODO: There's probably a better way to do this hack).

6.1 Interpretability of *HATD*

In order to interpret the result of SATD classification, Potdar et al. [38] have manually summarized 62 frequent SATD patterns for detecting SATD comments. However, this manual process is time-consuming and labor-intensive, and difficult to summarize all SATD patterns. Ren et al. [39] exploit the backtracking mechanism to map CNN-extracted features to key h-grams and summarize key h-grams into SATD patterns. Their method, however, requires a backtracking mechanism to propagate the extracted features back into the same structure of the model.

In this work, we propose an attention-based neural network to detect and explain SATD classifications. Through the visualization of weights, we find our method can immediately provide word-level and phrase-level interpretability for each comment, regardless of project differences and time-consuming pattern summary issues.

Figure 7 shows two examples of the comments we have examined. The color shade indicates the attention weight of words. The higher the attention weight, the darker the color. For visualization purposes, we remove padding words to ensure that only meaningful words are emphasized. From Figure 7, we can see that our method highlights the word *hack* in the first comment. While in the second comment, our method highlights the word *TODO* and the phrase *a better way*. We find an interesting phenomenon that the keyword *hack* carries a significant SATD tendency in the first comment, which is different from that in the second comment. This shows that the same keywords will play different roles for SATD detection in different comments, which further exacerbates the difficulty of manually summarizing SATD patterns to identify SATDs.

Furthermore, we summarize in Table 7 the Top-10 SATD patterns for each of the ten open-source projects. Interestingly, we can observe that many projects share some common one-gram SATD patterns (underlined), such as "todo", "xxx", "fix", "fixme", "hack", "work", "perhaps", "workaround", "better" and "bug". At the same time, some SATD patterns appear in only one or two projects. For instance, “a public setter" is only used in Apache Ant project, “model element", "model/extent" and "behind addTrigger" are only used in ArgoUML project, "programming error" and "tweak" are only used in JMeter and JFreeChart projects, respectively. We observe that some patterns (bold) are not discovered by Potdar et al. and Ren et al. [38, 39], suggesting that manual summary of SATD patterns will likely miss some less evidential SATD patterns and will also encounter the problem of project differences. Our method can supplement those works by automatically identifying SATDs and highlight keywords or phrases contributing to such identifications, regardless of project differences.

6.2 Threats to Validity

Threats of internal validity are related to errors in our implementation and personal bias in data labeling. To avoid implementation errors, we have carefully checked our experimental steps and parameter settings. To avoid the personal biases of manually marking the code comments, the dataset used in our experiments has a high inter-rater agreement (Cohen's Kappa coefficient of 0.81), which has reported by Maldonado et al. [31]. In addition, we do not use the fine-grained SATD categories proposed by [7] and only consider the binary classification of code comments. In other words, design debt, requirement debt, defect debt, documentation debt, and test debt are all simply treated as SATDs.

Threats of external validity are related to both the quantity and quality of our experimental dataset and the generalizability of our experiment results and findings. To guarantee the quantity and quality of our dataset, we have taken into account ten open-source projects with different functions and fields, and a different number of comments as well as comment characteristics. In order to guarantee the generalizability of our experiment results and findings, we have crawled another ten popular software projects from Github to test our trained model. By manually verifying the results reported by our method, we conclude that our method has achieved an accuracy of 83% on 100 randomly selected comments.

7 RELATED WORK

Self-Admitted Technical Debt (SATD) is a variant of technical debt that is used to identify debt that is intentionally introduced during the software development process. The detection of SATD can better support software maintenance and ensure high software quality [9, 17, 22, 26, 28, 29, 33]. This paper aims to propose a deep learning based method to detect SATDs. To the best of our knowledge, there is very limited research on the use of deep learning technology for SATD detection. Therefore, we divide the related work into two main parts: code comment analysis, existing studies in Self-Admitted Technical Debt.

7.1 Code Comment Analysis

Code comments play an important role in software development. The significance of comments is to enable developers to easily understand and manage codes[15, 42, 49]. Code comments are usually written in natural language texts containing functional descriptions and task annotations, aiming to assist project development. Code comments will have different characteristics due to differences in developers, projects, and programming languages.

Many studies investigated the characteristics of code comments. Fluri et al. [11] investigated the level of developers adding comments or adapting comments when evolving code in three open-source systems. As a result, they found that when the relevant code changed, the comment changes were usually made in the same version. Steidl et al. [44] and Sun et al. [46] conducted a code comment quality analysis to improve the quality of code comments. More specifically, Steidl et al. provide a semi-automated approach to conduct quantitative and qualitative evaluation of comment quality. Sun et al. extended their work and provide a more accurate and comprehensive comment assessments and recommendations.

7.2 Studies in Self-Admitted Technical Debt

Currently, many researchers have focused on proposing approaches to detect and manage technical debts. Indeed, many empirical studies have applied on technical debts [1, 24, 40, 61–63]. The concept of self-admitted technical debt (SATD) is proposed by Potdar et al[38]. SATD means that the debt is intentionally introduced and reported in code comments by developers. They manually summarized 62 specific patterns from different Java projects for the detection of SATD comments. Wehaibi et al. [52] examined the relationship between SATDs and software defects and found that SATDs caused software systems to consume more resources in the future, rather than equating to software defects. According to different characteristics of SATD comments, Maldonado et al. [31] further classified SATD into five types, namely design debt, defect debt, document debt, requirement debt, and test debt.

In recent years, researchers in the field of software engineering have made significant efforts to address the issue of SATD detections [7, 8, 32, 57]. Huang et al.[16] proposed a text-mining-based method for SATD detections. In their work, they leverage feature selectors to select useful features for classifier training and detect SATD comments in the target project based on the results of the classifier votes from different source projects. Maldonado et al. [7] built a maximum entropy classifier for automatically identifying design and requirement SATD comments based on Natural Language Processing (NLP) technologies. They conducted extensive experiments in 10 open-source projects and outperformed the current state-of-the-art based on fixed keywords and phrases. Yan et al. [55] proposed a change-level method for SATD detections utilizing 25 change features. In addition, they investigated the most important features (“diffusion”) that impact SATD detections.

With the rapid development of deep learning, Ren et al. [39] exploited a deep learning based method as support of SATD detections. In their work, they utilized a convolutional neural network (CNN) to automatically learn key features in SATD comments. They introduced a weighted loss function to deal with the issue of data

Table 7: Top-10 SATD patterns in each project extracted by our approach

Apache Ant	**ArgoUML**	**Columba**	**EMF**	**Hibernate**
xxx	todo	todo	todo	todo
todo	not needed	fixme	better	better
what is	should be	function need	remove	work
a public setter	**model/extent**	work	why not this	support
fixme	**model element**	implement	**factor up into**	bug
perhaps	more work	**replace with**	**SubProgressMonitor**	workaround
hack	fixme	hack	revisit	fix
better	**behind addTrigger**	better	**GenBaseImpl**	perhaps
should we	not in uml	workaround	instead of	fixme
this comment	hack	used currently	fix	render

JEdit	**JFreeChart**	**JMeter**	**JRuby**	**SQuirrel**
hack	todo	todo	todo	todo
work	check	hack	fixme	work
todo	**defer argument checking...**	perhaps	**require pop**	data type
fix	fixme	not used	hack	render
workaround	**tweak**	work	better	bug
bug	happen	appear	**bother**	**hack to deal with**
xxx	implement this	fix	fix	follow
broken	**assert a value**	**programming error**	perhaps	workaround
stupid	**crosshair values**	**bail out**	**NOT_ALLOCATABLE_ALLOCATOR**	is this right
look bad	**NOT trigger**	bug	not very efficient	better way

imbalance. In addition, for the explainability of detection, they designed a backtracking method to extract and highlight key phrases and patterns of SATD comments to explain the prediction results.

8 CONCLUSION

The objective of this work is to automatically detect and explain self-admitted technical debts in software projects, allowing software developers to fix those issues in a timely manner so as to avoid long-time maintenance efforts. To fulfill this objective, in this work, we have provided an overview of code comment characteristics that make it challenging to automatically detect SATDs. Then, we propose to the community a hybrid attention-based method for SATD detection named *HATD*, which has been equipped with the flexibility to switch word embedding techniques based on project uniqueness and comment characteristics. After that, we experimentally demonstrate the efficiency of *HATD* by (1) outperforming state-of-the-art methods on benchmark datasets, (2) detecting SATDs in real-world software projects, and (3) demonstrating the explainability of *HATD* in highlighting the core words or phrases justifying why a given SATD is flagged as such.

ACKNOWLEDGMENT

This work was supported by the grands of the National Natural Science Foundation of China (Nos.61972290, U163620068, 61962061, 61562090).

REFERENCES

[1] Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, and Rodrigo O. Spinola. 2014. Towards an Ontology of Terms on Technical Debt. In *2014 Sixth International Workshop on Managing Technical Debt*. 1–7.

[2] Nicolli S R Alves, Thiago Souto Mendes, Manoel Mendonca, Rodrigo O Spinola, Forrest Shull, and Carolyn Seaman. 2016. Identification and management of technical debt. *Information and Software Technology* 70, 70 (2016), 100–121.

[3] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. 2015. The financial aspect of managing technical debt. *Information and Software Technology* 64, 64 (2015), 52–73.

[4] G. Bavota and B. Russo. 2016. A Large-Scale Empirical Study on Self-Admitted Technical Debt. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 315–326.

[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[6] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), 29–30.

[7] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[8] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Inf. Softw. Technol.* 121 (2020), 106270.

[9] Jernej Flisar and Vili Podgorelec. 2018. Enhanced Feature Selection Using Word Embeddings for Self-Admitted Technical Debt Identification. In *Euromicro Conference on Software Engineering and Advanced Applications*.

[10] Jernej Flisar and Vili Podgorelec. 2019. Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding. *IEEE Access* 7 (2019), 106475–106494.

[11] B. Fluri, M. Wursch, and H.C. Gall. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. 70–79.

[12] Matthieu Foucault, Xavier Blanc, Margaretanne Storey, Jeanremy Falleri, and Cedric Teyton. 2018. Gamification: a Game Changer for Managing Technical Debt? A Design Study. *arXiv: Software Engineering* (2018).

[13] Sávio Freire, Nicolli Rios, Boris Gutierrez, Darío Torres, Manoel G. Mendonça, Clemente Izurieta, Carolyn B. Seaman, and Rodrigo O. Spínola. 2020. Surveying Software Practitioners on Technical Debt Payment Practices and Reasons for not Paying off Debt Items. In *EASE*. 210–219.

[14] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2019. MAT: A simple yet strong baseline for identifying self-admitted technical debt. *arXiv: Software Engineering* (2019).

[15] Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. 2013. Automatically mining software-based, semantically-similar words from comment-code mappings. In *MSR*. 377–386.

[16] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[17] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. 2019. Self-Admitted Technical Debt Removal and Refactoring Actions:

Co-Occurrence or More?. In *2019 IEEE International Conference on Software Maintenance and Evolution,ICSME*. 186–190.

[18] Clemente Izurieta, Ipek Ozkaya, Carolyn B. Seaman, Philippe Kruchten, Robert L. Nord, Will Snipes, and Paris Avgeriou. 2016. Perspectives on Managing Technical Debt: A Transition Point and Roadmap from Dagstuhl. In *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016)*, Vol. 1771. 84–87.

[19] Armand Joulin, Edouard Grave, and Piotr Bojanowski Tomas Mikolov. 2017. Bag of Tricks for Efficient Text Classification. *EACL 2017* (2017), 427.

[20] Michael Kampffmeyer, Arnt-Borre Salberg, and Robert Jenssen. 2016. Semantic segmentation of small objects and modeling of uncertainty in urban remote sensing images using deep convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 1–9.

[21] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY - A Code-to-Code Search Engine. In *ICSE 2018*.

[22] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).

[23] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21.

[24] Philippe Kruchten, Robert L Nord, Ipek Ozkaya, and Davide Falessi. 2013. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes* 38, 5 (2013), 51–54.

[25] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2019. The Technical Debt Dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2019, Recife, Brazil, September 18, 2019*. ACM, 2–11.

[26] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).

[27] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.

[28] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD detector: a text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 9–12.

[29] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2019. Wait For It: Identifying "On-Hold" Self-Admitted Technical Debt. *arXiv: Software Engineering* (2019).

[30] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. 2017. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 238–248.

[31] Everton da S Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 9–15.

[32] Solomon Mensah, Jacky Keung, Jeffery Svajlenko, Kwabena Ebo Bennin, and Qing Mi. 2018. On the value of a prioritization scheme for resolving Self-admitted technical debt. *Journal of Systems and Software* 135 (2018), 37–54.

[33] Solomon Mensah, Jacky W. Keung, Michael Franklin Bosu, and Kwabena Ebo Bennin. 2016. Rework Effort Estimation of Self-admitted Technical Debt. In *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) Hamilton, New Zealand, December 6, 2016*, Vol. 1771. 72–75.

[34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[35] Robert L. Nord, Ipek Ozkaya, Edward J. Schwartz, Forrest Shull, and Rick Kazman. 2016. Can Knowledge of Technical Debt Help Identify Software Vulnerabilities?. In *9th Workshop on Cyber Security Experimentation and Test*.

[36] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*. 1532–1543.

[37] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of NAACL-HLT*. 2227–2237.

[38] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *ICSME*. IEEE, 91–100.

[39] XIAOXUE REN, ZHENCHANG XING, XIN XIA, DAVID LO, XINYU WANG, and JOHN GRUNDY. 2019. Neural Network Based Detection of Self-admitted Technical Debt: From Performance to Explainability. *ACM Transactions on Software Engineering and Methodology* 28, 3 (2019).

[40] Carolyn Seaman, Yuepu Guo, Nico Zazworka, Forrest Shull, Clemente Izurieta, Yuanfang Cai, and Antonio Vetrò. 2012. Using technical debt data in decision making: Potential decision approaches. In *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 45–48.

[41] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82.

[42] Giancarlo Sierra, Ahmad Tahmid, Emad Shihab, and Nikolaos Tsantalis. 2019. Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences?. In *SANER*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). 534–543.

[43] Rodrigo O. Spínola, Nico Zazworka, Antonio Vetro, Forrest Shull, and Carolyn B. Seaman. 2019. Understanding automated and human-based technical debt identification approaches-a two-phase study. *J. Braz. Comp. Soc.* 25, 1 (2019), 5:1–5:21.

[44] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*. 83–92.

[45] Ben Stopford, Ken Wallace, and John Allspaw. 2017. Technical Debt: Challenges and Perspectives. *IEEE Software* 34, 4 (2017), 79–81.

[46] Xiaobing Sun, Qiang Geng, David Lo, Yucong Duan, Xiangyue Liu, and Bin Li. 2016. Code Comment Quality Analysis and Improvement Recommendation: An Automated Approach. *International Journal of Software Engineering and Knowledge Engineering* 26, 6 (2016), 981–1000.

[47] Edith Tom, Aybuke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* 86, 6 (2013), 1498–1516.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[49] Bradley L. Vinz and Letha H. Etzkorn. 2008. Improving program comprehension by combining code understanding with comment understanding. *Knowledge Based Systems* 21, 8 (2008), 813–825.

[50] Supatsara Wattanakriengkrai, Rungroj Maipradit, Hideki Hata, Morakot Choetkiertikul, and Kenichi Matsumoto. 2018. Identifying Design and Requirement Self-Admitted Technical Debt Using N-gram IDF. In *IWESEP*.

[51] Supatsara Wattanakriengkrai, Napat Srisermphoak, Sahawat Sintoplertchaikul, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Hideaki Hata, and Kenichi Matsumoto. 2019. Automatic Classifying Self-Admitted Technical Debt Using N-Gram IDF. (2019), 316–322.

[52] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 179–188.

[53] Will, Snipes, Carolyn, Seaman, Ipek, Ozkaya, Clemente, and Izurieta. 2017. Technical Debt: A Research Roadmap Report on the Eighth Workshop on Managing Technical Debt (MTD 2016). *Software Engineering Notes Acm Sigsoft* (2017).

[54] Jifeng Xuan, Yan Hu, and Jiang He. 2012. Debt-Prone Bugs: Technical Debt in Software Maintenance. *International Journal of Advancements in Computing Technology* 4 (10 2012), 453–461.

[55] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* (2018).

[56] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2019. Automating Change-Level Self-Admitted Technical Debt Determination. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1211–1229.

[57] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. 2020. Identifying Self-Admitted Technical Debts with Jitterbug: A Two-step Approach. *CoRR* abs/2002.11049 (2020).

[58] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. 2017. Recommending when Design Technical Debt Should be Self-Admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME*. 216–226.

[59] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was Self-Admitted Technical Debt Removal a Real Removal? An In-Depth Perspective. In *MSR*.

[60] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Automatically Learning Patterns for Self-Admitted Technical Debt Removal. In *SANER*.

[61] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. 17–23.

[62] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro, Forrest Shull, and Carolyn Seaman. 2013. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. 42–47.

[63] Nico Zazworka, Antonio Vetro, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. 2014. Comparing four approaches for technical debt identification. *Software Quality Journal* 22, 3 (2014), 403–426.

[64] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. 2016. 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)*. 207–212.