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 (Hybrid Attention-based method for self-admitted Technical Debt detection), to detect and explain SATDs using attention-based neural networks. Through extensive experiments on 443,365 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

∗Jin Liu is corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

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 [7], 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 [8], 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.

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

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 early 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 conduct a simple empirical study (count the number of commits related to files that have been changed, with and without technical debts) 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 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.

#### 2.1 SATD VS. Code Changes

Intuitively, SATDs will trigger more future code changes, i.e., increasing the maintainability of the software project. 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 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.

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. 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.

---

\(^{3}\text{Given a significance level } \alpha = 0.001, \text{ if } p-value < \alpha, \text{ there is one chance in a thousand that the difference between the datasets is due to a coincidence.}\)

---

3 OUR APPROACH

We propose a Hybrid Attention-based method for self-admitted Technical Debt 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 embedding 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
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.

**Word Encoder.** Given a code comment, we assume that the comment contains $l$ words. $x_l$ represents the $i$th 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 $x_1$:

$$h_t = LSTM(x_t), t \in [1, l],$$
$$h'_t = LSTM(x_t), t \in [1, l].$$

We obtain the implicit representation of a given word $x_t$ by concatenating the forward hidden 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.

$$u_t = tanh(W_{u} [h_t, h'_t] + b_w)$$
$$a_i = softmax(u^T_i * u_w)$$
$$S^1 = \sum_{i} a_i [h_i, h'_i]$$

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 $a_i$ 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:

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

\[
x_t = (x_t + p_t)W_p + b_p \quad (7)
\]

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:

\[
Q_j = XW_q 
\]

\[
K_j = XW_k 
\]

\[
V_j = XW_v \quad (10)
\]

\[
\text{head} = \text{Softmax}\left(\frac{QJK^T}{\sqrt{d_k}}\right)V_j \quad (11)
\]

\[
S^2 = \text{Norm}(\text{Cat} (\text{head}_1,...,\text{head}_8)W^o) \quad (12)
\]

Here, \( X \in \mathbb{R}^{d \times d} \) 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 \( (h = 8 \text{ by default}) \) to expand the model’s ability to focus on different words. For each head we have \( d_q = d_k = d_v = d/h \). \( W^o \in \mathbb{R}^{hd \times d} \). 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 \( \text{sigmoid} \) function. The classification rule is defined by the following equations:

\[
\hat{Y} = \text{Sigmoid}(f_{fc}(S^1 \oplus S^2)) \quad (13)
\]

Here, \( \oplus \) 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:

\[
\text{weight}_s = \begin{cases} 
\frac{b}{a+b} & \text{if } \text{sth class is SATD} \\
\frac{a}{a+b} & \text{otherwise}
\end{cases} \quad (14)
\]

\[
\text{Loss}(\hat{Y}, T) = -\sum_s \text{weight}_s \sum_t \log(\hat{Y}_s) \quad (15)
\]

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 other baseline techniques 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.
We now present our experimental results towards answering the question of non-SATD. Based on these 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.

\[ \text{Precision} = \frac{TP}{TP + FP} \]  
\[ \text{Recall} = \frac{TP}{TP + FN} \]  
\[ F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>Release</th>
<th>Contributions</th>
<th>Comments</th>
<th>SATD</th>
<th>% of SATD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Ant</td>
<td>Java library and Command-line Tool</td>
<td>1.70</td>
<td>74</td>
<td>4,137</td>
<td>131</td>
<td>3.17</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>UML Modeling Tool</td>
<td>0.34</td>
<td>87</td>
<td>9,548</td>
<td>1,413</td>
<td>14.80</td>
</tr>
<tr>
<td>Columbia</td>
<td>E-mail Client</td>
<td>1.4</td>
<td>9</td>
<td>6,478</td>
<td>204</td>
<td>3.15</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Model Driven Architecture</td>
<td>2.41</td>
<td>30</td>
<td>4,401</td>
<td>104</td>
<td>2.36</td>
</tr>
<tr>
<td>Hibernate</td>
<td>ORM Framework</td>
<td>3.32</td>
<td>226</td>
<td>2,968</td>
<td>472</td>
<td>15.90</td>
</tr>
<tr>
<td>JEdit</td>
<td>Text Editor</td>
<td>4.2</td>
<td>57</td>
<td>10,322</td>
<td>256</td>
<td>2.48</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>Char library</td>
<td>1.019</td>
<td>19</td>
<td>4,423</td>
<td>209</td>
<td>4.73</td>
</tr>
<tr>
<td>JMeter</td>
<td>Performance Tester</td>
<td>2.10</td>
<td>33</td>
<td>8,162</td>
<td>374</td>
<td>4.58</td>
</tr>
<tr>
<td>JRuby</td>
<td>Ruby Interpreter</td>
<td>1.40</td>
<td>328</td>
<td>4,897</td>
<td>662</td>
<td>13.52</td>
</tr>
<tr>
<td>SQUILrel</td>
<td>SQL Client</td>
<td>3.03</td>
<td>46</td>
<td>7,230</td>
<td>285</td>
<td>3.94</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>91</td>
<td>6,257</td>
<td>411</td>
<td>6.37</td>
<td></td>
</tr>
</tbody>
</table>

4 https://pytorch.org/

5.3 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 L2 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.

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Ant</td>
<td>0.7059</td>
<td>0.6469</td>
<td>0.6545</td>
<td>0.7064</td>
<td>0.6801</td>
<td>0.7132</td>
<td>0.7059</td>
<td>0.6469</td>
<td>0.6545</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>0.8731</td>
<td>0.9469</td>
<td>0.9085</td>
<td>0.8615</td>
<td>0.9487</td>
<td>0.9030</td>
<td>0.8731</td>
<td>0.9469</td>
<td>0.9085</td>
</tr>
<tr>
<td>Columbia</td>
<td>0.9434</td>
<td>0.9616</td>
<td>0.9524</td>
<td>0.8906</td>
<td>0.9598</td>
<td>0.9239</td>
<td>0.9434</td>
<td>0.9616</td>
<td>0.9524</td>
</tr>
<tr>
<td>EMF</td>
<td>0.7475</td>
<td>0.8586</td>
<td>0.7992</td>
<td>0.6769</td>
<td>0.8787</td>
<td>0.7647</td>
<td>0.7475</td>
<td>0.8586</td>
<td>0.7992</td>
</tr>
<tr>
<td>Hibernate</td>
<td>0.8666</td>
<td>0.9536</td>
<td>0.9189</td>
<td>0.8699</td>
<td>0.9305</td>
<td>0.8992</td>
<td>0.8666</td>
<td>0.9536</td>
<td>0.9189</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.5468</td>
<td>0.6554</td>
<td>0.5962</td>
<td>0.7783</td>
<td>0.8797</td>
<td>0.8259</td>
<td>0.5468</td>
<td>0.6554</td>
<td>0.5962</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>0.9268</td>
<td>0.8747</td>
<td>0.9000</td>
<td>0.6897</td>
<td>0.7135</td>
<td>0.7014</td>
<td>0.9268</td>
<td>0.8747</td>
<td>0.9000</td>
</tr>
<tr>
<td>JMeter</td>
<td>0.9249</td>
<td>0.8938</td>
<td>0.9091</td>
<td>0.8178</td>
<td>0.8715</td>
<td>0.8438</td>
<td>0.9249</td>
<td>0.8938</td>
<td>0.9091</td>
</tr>
<tr>
<td>JRuby</td>
<td>0.9568</td>
<td>0.9588</td>
<td>0.9477</td>
<td>0.8977</td>
<td>0.9397</td>
<td>0.9182</td>
<td>0.9568</td>
<td>0.9588</td>
<td>0.9477</td>
</tr>
<tr>
<td>SQUILrel</td>
<td>0.7978</td>
<td>0.8769</td>
<td>0.8355</td>
<td>0.7814</td>
<td>0.8318</td>
<td>0.8058</td>
<td>0.7978</td>
<td>0.8769</td>
<td>0.8355</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.8309</td>
<td>0.8571</td>
<td>0.8422</td>
<td>0.7921</td>
<td>0.8734</td>
<td>0.8299</td>
<td>0.8309</td>
<td>0.8571</td>
<td>0.8422</td>
</tr>
</tbody>
</table>

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.
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 approach compared to the state-of-the-art.

<table>
<thead>
<tr>
<th>Projects</th>
<th>With-Project</th>
<th>Cross-Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ren s</td>
<td>Ours</td>
</tr>
<tr>
<td>Apache Ant</td>
<td>0.5977</td>
<td>0.6345</td>
</tr>
<tr>
<td>ArgouML</td>
<td>0.8895</td>
<td>0.9085</td>
</tr>
<tr>
<td>Columbus</td>
<td>0.8545</td>
<td>0.9524</td>
</tr>
<tr>
<td>EMF</td>
<td>0.7010</td>
<td>0.7992</td>
</tr>
<tr>
<td>Hibernate</td>
<td>0.8465</td>
<td>0.9189</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.5364</td>
<td>0.5962</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>0.7759</td>
<td>0.9000</td>
</tr>
<tr>
<td>JMeter</td>
<td>0.8858</td>
<td>0.9091</td>
</tr>
<tr>
<td>JRuby</td>
<td>0.8626</td>
<td>0.9477</td>
</tr>
<tr>
<td>JQuirrel</td>
<td>0.7467</td>
<td>0.8355</td>
</tr>
<tr>
<td>Average</td>
<td>0.7717</td>
<td>0.8422</td>
</tr>
</tbody>
</table>

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 average, 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 training 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.
We then launch the new version of the word embedding module with one of the aforementioned four techniques, 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 sizes 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.

### 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...
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.

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

<table>
<thead>
<tr>
<th>Apache Ant</th>
<th>ArgoUML</th>
<th>Columbia</th>
<th>EMF</th>
<th>Hibernate</th>
</tr>
</thead>
<tbody>
<tr>
<td>todo</td>
<td>todo</td>
<td>todo</td>
<td>todo</td>
<td>todo</td>
</tr>
<tr>
<td>not needed</td>
<td>fixme</td>
<td>function need</td>
<td>remove</td>
<td>better</td>
</tr>
<tr>
<td>should be</td>
<td>should be</td>
<td>function need</td>
<td>why not this</td>
<td>support</td>
</tr>
<tr>
<td>a public setter</td>
<td>model/extent</td>
<td>work</td>
<td>implement</td>
<td>factor up into</td>
</tr>
<tr>
<td>model element</td>
<td>model element</td>
<td>implement</td>
<td>SubProgressMonitor</td>
<td>workaround</td>
</tr>
<tr>
<td>more work</td>
<td>replace with</td>
<td>SubProgressMonitor</td>
<td>workaround</td>
<td>fix</td>
</tr>
<tr>
<td>fixme</td>
<td>todo</td>
<td>todo</td>
<td>todo</td>
<td>todo</td>
</tr>
<tr>
<td>behind addTrigger</td>
<td>behind addTrigger</td>
<td>behind addTrigger</td>
<td>behind addTrigger</td>
<td>behind addTrigger</td>
</tr>
<tr>
<td>not in uml</td>
<td>workaround</td>
<td>workaround</td>
<td>fix</td>
<td>renderer</td>
</tr>
<tr>
<td>this comment</td>
<td>hack</td>
<td>todo</td>
<td>todo</td>
<td>todo</td>
</tr>
<tr>
<td>todo</td>
<td>work</td>
<td>check</td>
<td>todo</td>
<td>todo</td>
</tr>
<tr>
<td>defer argument checking...</td>
<td>defer argument checking...</td>
<td>defer argument checking...</td>
<td>defer argument checking...</td>
<td>defer argument checking...</td>
</tr>
<tr>
<td>fix</td>
<td>fixme</td>
<td>not used</td>
<td>hack</td>
<td>require pop</td>
</tr>
<tr>
<td>workaround</td>
<td>workaround</td>
<td>workaround</td>
<td>better</td>
<td>better</td>
</tr>
<tr>
<td>bug</td>
<td>happen</td>
<td>appear</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>xxxx</td>
<td>implement this</td>
<td>fix</td>
<td>hack</td>
<td>better</td>
</tr>
<tr>
<td>broken</td>
<td>assert a value</td>
<td>programming error</td>
<td>not used</td>
<td>data type</td>
</tr>
<tr>
<td>stupid</td>
<td>crosshair values</td>
<td>bail out</td>
<td>require pop</td>
<td>fix</td>
</tr>
<tr>
<td>look bad</td>
<td>NOT trigger</td>
<td>bug</td>
<td>todo</td>
<td>todo</td>
</tr>
</tbody>
</table>

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 grants of the National Natural Science Foundation of China (Nos.61972290, U163620068, 61962061, 61562090).

REFERENCES


[36] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe:
[22] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein.
[38] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted