Identifying self-admitted technical debt in issue tracking systems using machine learning

Li, Yikun; Soliman, Mohamed; Avgeriou, Paris

Published in:
Empirical software engineering

DOI:
10.1007/s10664-022-10128-3

Empirical Software Engineering (2022) 27:131
https://doi.org/10.1007/s10664-022-10128-3

Identifying self-admitted technical debt in issue tracking systems using machine learning

Yikun Li[1] · Mohamed Soliman[1] · Paris Avgeriou[1]

Accepted: 31 January 2022

Abstract

Technical debt is a metaphor indicating sub-optimal solutions implemented for short-term benefits by sacrificing the long-term maintainability and evolvability of software. A special type of technical debt is explicitly admitted by software engineers (e.g. using a TODO comment); this is called Self-Admitted Technical Debt or SATD. Most work on automatically identifying SATD focuses on source code comments. In addition to source code comments, issue tracking systems have shown to be another rich source of SATD, but there are no approaches specifically for automatically identifying SATD in issues. In this paper, we first create a training dataset by collecting and manually analyzing 4,200 issues (that break down to 23,180 sections of issues) from seven open-source projects (i.e., Camel, Chromium, Gerrit, Hadoop, HBase, Impala, and Thrift) using two popular issue tracking systems (i.e., Jira and Google Monorail). We then propose and optimize an approach for automatically identifying SATD in issue tracking systems using machine learning. Our findings indicate that: 1) our approach outperforms baseline approaches by a wide margin with regard to the F1-score; 2) transferring knowledge from suitable datasets can improve the predictive performance of our approach; 3) extracted SATD keywords are intuitive and potentially indicating types and indicators of SATD; 4) projects using different issue tracking systems have less common SATD keywords compared to projects using the same issue tracking system; 5) a small amount of training data is needed to achieve good accuracy.

Communicated by: Foutse Khomh, Gemma Catolino and Pasquale Salza

This article belongs to the Topical Collection: *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*

✉ Yikun Li
yikun.li@rug.nl

Mohamed Soliman
m.a.m.soliman@rug.nl

Paris Avgeriou
p.avgeriou@rug.nl

[1] Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands

Published online: 10 July 2022

Keywords Self-admitted technical debt · Technical debt identification · Issue tracking system · Deep learning · Transfer learning

1 Introduction

Technical debt (TD) is a metaphor reflecting the implementation or adoption of sub-optimal solutions to achieve short-term benefits while sacrificing the long-term maintainability and evolvability of software (Avgeriou et al. 2016).

TD is incurred either deliberately (e.g. to meet a deadline) or inadvertently (e.g. due to lack of domain knowledge) and tends to accumulate over time. If left unmanaged, the accumulation of TD can lead to critical issues in software maintainability and evolvability. To manage TD appropriately, a number of activities have been proposed, among which, *identification* is the first step (Li et al. 2015).

Most TD identification approaches in literature use static source code analysis (Alves et al. 2016). While this works well for identifying coding rule violations, it also has limitations. Most importantly, the scope of TD identification is limited to code-level issues such as code smells (Tufano et al. 2017). It does not consider other types of TD (e.g. bad architecture decisions or requirements debt (Ernst, 2012)), which are, in general, harder to detect from the source code directly.

Potdar and Shihab (2014) proposed a new approach to detect TD, focusing on the so-called *Self-Admitted Technical Debt* (SATD): technical debt that is *explicitly* admitted by software engineers in source code comments. The identification of SATD facilitates capturing TD items, which are harder to detect using source code analysis. For instance, when developers choose to partially implement a requirement in order to deliver the software faster, they are likely to note that down in source code comments, e.g. by using a TODO comment (Potdar and Shihab 2014). Another example is the use of a technology (e.g. an architecture framework), not because it is the optimal solution, but because there is a business relation with the technology provider. Such examples of TD cannot be identified by analyzing source code.

Studies on identifying SATD, focus mostly on source code comments (Sierra et al. 2019). Recently, issue tracking systems have shown to be another rich source for SATD, which acts as complementary to source code comments, i.e. it allows identifying SATD that is not admitted in source code comments (Li et al. 2020; Bellomo et al. 2016). In addition, issue tracking systems have the advantage of involving useful discussions between software engineers across the development activities (e.g. within requirements engineering, architecture design, and testing) (Merten et al. 2015; Li et al. 2020). Therefore, there is high potential of finding comprehensive SATD content within issues; in contrast, source code comments hardly ever contain such discussions. However, on the one hand, it is very time-consuming to manually identify SATD in issue tracking systems (Li et al. 2020). On the other hand, we lack approaches that automatically identify SATD in issue tracking systems.

In this paper, we aim at proposing and evaluating an approach for automatically identifying SATD in issue tracking systems. An issue usually consists of an issue summary, an issue description, and a number of comments from different developers. We call each part of an issue (i.e. summary, description or comment) as *issue section*. We first collect 4,200 issues (that correspond to 23,180 issue sections) from seven large open-source projects from two ecosystems: Apache and Google. We then manually analyze the issue sections in order to identify SATD based on our previously defined classification framework (Li et al. 2020); this results in creating the biggest SATD dataset for issue tracking systems. Finally, based on this dataset for training, we experiment with several traditional and modern machine learning techniques to automatically identify SATD in issues within two popular issue tracking systems (i.e. Jira and Google Monorail). To optimize the machine learning outcome, we also explore different word embeddings, machine learning configurations (e.g. hyperparameter tuning), as well as transfer learning.

The main contributions of this paper are the following:

1. **Contributing a rich dataset of self-admitted technical debt in issue tracking systems.** We collect 4,200 issues (that contain 3,277 SATD issue sections out of 23,180 issue sections) from seven open-source projects using two issue tracking systems. The dataset includes annotations regarding the type and indicator of each SATD issue section. We make our dataset publicly available[1] to encourage future research in this area.
2. **Comparing different machine learning techniques and optimizing the best approach to identify SATD in issue tracking systems.** We compare the F1-score of different machine learning approaches identifying SATD in issues and find out that Text CNN (Kim 2014) outperforms others. We then further investigate imbalanced data handling strategies, word embedding techniques, and hyperparameter tuning, and optimize the CNN-based approach to accurately identify SATD issue sections in issue tracking systems. Moreover, we conduct extensive experiments to evaluate the effectiveness of transferring knowledge gained from other datasets.
3. **Extracting the most informative SATD keywords and comparing keywords from different projects and sources.** We summarize and present a list of the most informative SATD keywords and we find that these keywords are intuitive and can potentially indicate types and indicators of SATD. Besides, we show that projects using different issue tracking systems have less common keywords compared to projects using the same issue tracking system. Moreover, we find source code comments and issue tracking systems have some common SATD keywords.
4. **Evaluating generalizability of our CNN-based approach.** We conduct experiments to evaluate the generalizability of our approach across projects and issue tracking systems.
5. **Exploring the amount of data necessary for training the model.** We find that only a small amount of training data is needed to achieve good accuracy.

The remainder of the paper is organized as follows. We begin by discussing some related work in Section 2. We then elaborate on the case study design in Section 3. Subsequently, we present and discuss the results in Sections 4 and 5 respectively. In Section 6, threats to validity are discussed. Finally, we present our conclusions in Section 7.

2 Related Work

In this paper, we work on an approach to identify SATD in issue tracking systems. Therefore, we divide the related work into two parts: work related to SATD in general and work related to SATD in issue tracking systems.

[1] https://github.com/yikun-li/satd-issue-tracker-data

2.1 Self-Admitted Technical Debt

Potdar and Shihab (2014) investigated source code comments that indicate technical debt items and named this phenomenon *self-admitted technical debt*. They manually analyzed 101,762 source code comments from four open-source projects (i.e., Eclipse, Chromium, Apache HTTP Server, and ArgoUML) to identify SATD comments. They found that SATD comments are widely spread in projects: 2.4% to 31.0% of files contain SATD comments. Moreover, they identified and summarized 62 keyword phrases that indicate SATD, such as *ugly*, *temporary solution*, and *this doesn't look right*. In a follow-up study, Maldonado and Shihab (2015) identified SATD by reading through 33,093 comments from five open-source project and manually classifying them into different types. They found that source code comments indicate five types of SATD: design, defect, documentation, requirement, and test debt. Besides, they observed that the majority of SATD is design debt as 42% to 84% of all identified SATD comments indicate design debt. After that, de Freitas Farias et al. (2016) investigated the effectiveness of Contextualized Vocabulary Model for identifying SATD in code comments.

Following up from Maldonado and Shihab (2015), to accurately and automatically identify SATD in source code comments, Maldonado et al. (2017) analyzed 29,473 source code comments from ten open-source projects and trained a maximum entropy classifier on the analyzed data. The results showed that design and requirement debt can be identified with the average F1-score of 0.620 and 0.402 respectively. Additionally, they found that training on a small subset of comments can achieve 80% and 90% of the best classification accuracy.

Subsequently, there was work on improving the accuracy of SATD identification in source code comments. Liu et al. (2018) proposed an approach based on text-mining to accurately and automatically detect SATD in source code comments by utilizing feature selection and combing sub-classifiers. The average F1-score achieved by their approach was improved from 0.576 to 0.737, compared to the work by Maldonado et al. (2017). Most recently, Ren et al. (2019) proposed a Convolutional Neural Network based approach for SATD identification in source code comments and showed that their approach outperformed previous methods.

Apart from SATD identification, there has been work related to measurement and repayment of SATD. Wehaibi et al. (2016) investigated the relation between SATD and software quality by analyzing five open-source projects, namely Chromium, Hadoop, Spark, Cassandra, and Tomcat. The findings indicated that SATD changes (modifications on files containing SATD comments) incur fewer defects compared to non-SATD changes, while SATD changes are more complex and difficult to perform than non-SATD changes. Kamei et al. (2016) found that about 42% to 44% of SATD sections incur positive technical debt interest. Zampetti et al. (2018) studied how SATD is resolved in five open-source projects, namely Camel, Gerrit, Hadoop, Log4j, and Tomcat. They found that between 20% and 50% of SATD comments are removed by accident (without addressing the SATD), while most of the repayment activities require complex source code changes.

2.2 SATD in Issue Tracking Systems

SATD in issue tracking systems is relatively unexplored: there are only four studies on the identification and repayment of SATD in issue trackers. Bellomo et al. (2016) explored the existence of SATD in four issue tracking systems from two government projects and two open-source projects. They analyzed 1,264 issues and annotated 109 issues as SATD issues. The results showed that technical debt is indeed declared and discussed in issue tracking systems. Subsequently, Dai and Kruchten (2017) identified 331 SATD issues from 8,149 issues by reading through the issue summaries and descriptions. They then trained a Naive Bayes classifier to automatically classify issues as SATD issues or non-SATD issues and extracted unigram keywords that indicate technical debt. The third study by Li et al. (2020) is our own previous work, where we investigated the identification and repayment of SATD in issues from two open-source projects. We annotated issues on the sentence level, instead of treating a whole issue as SATD or not, in order to have better accuracy. We then presented types of SATD, the points of time when SATD was identified and reported, and how SATD was eventually resolved. Lastly, Xavier et al. (2020) studied a sample of 286 SATD issues and found 29% of SATD in issues can be tracked to source code comments.

In this article, we analyze issues on a more fine-grained level compared to three of the related studies (Bellomo et al. 2016; Dai and Kruchten 2017; Xavier et al. 2020) as they both treated a whole issue as a single technical debt statement. In contrast, we look at issue sections (i.e., individual issue summaries, descriptions, or comments) and potentially annotate them as SATD issue sections. As compared to the third study (Li et al. 2020), we analyze about eight times more issues (4,200 issues versus 500 issues) for machine learning training. Besides, we propose a deep learning approach to accurately identify SATD in issues and compare the accuracy with other traditional machine learning methods (see RQ1). Moreover, we extract and summarize unigram to five-gram keywords, compared to only unigram keywords extracted from issue descriptions by Dai and Kruchten (2017) (see RQ2). Furthermore, we investigate the generalization of our approach (see RQ3) and explore the amount of data needed for training the machine learning model (see RQ4). RQ3 and RQ4 were not investigated before by other researchers. Finally, this is the first work discussing the differences between SATD from different sources (i.e., source code comments and issue tracking systems).

3 Study Design

We follow the guidelines for case study research proposed by Runeson et al. (2012) to design and conduct the study. The goal of the study, formulated according to the Goal-Question-Metric (van Solingen et al. 2002) template, is to "***analyze*** *issues in issue tracking systems* ***for the purpose of*** *automatically identifying self-admitted technical debt within issues* ***with respect to*** *accuracy, explainability, and generalizability* ***from the point of*** *developers* ***in the context of*** *open-source software*". This goal is refined into four research questions (RQs), which consider *accuracy* (RQ1 and RQ4), *explainability* (RQ2) and *generalizability* (RQ3). The RQs and their motivations are explained below.

- **(RQ1)** *How to accurately identify self-admitted technical debt in issue tracking systems?* This research question focuses on the *accuracy* of SATD automatic identification, and is further refined into three sub-questions:
  - **(RQ1.1)** *Which algorithms have the best accuracy to capture self-admitted technical debt in issue tracking systems?* Since we are aiming at accurately identifying SATD within issues, we need to compare the accuracy of different approaches to choose the best one.
  - **(RQ1.2)** *How to improve accuracy of machine learning model?* To optimize the accuracy of identifying SATD in issue tracking systems, we investigate word embedding refinement, imbalanced data handling strategies, and hyperparameters tuning.

  - **(RQ1.3)** *How can transfer learning improve the accuracy of identifying self-admitted technical debt in issue tracking systems?* Transfer learning focuses on using knowledge gained while solving one task to address a different but related task. Therefore, we can study the influence of leveraging external datasets (from source code comments) on our SATD detector using transfer learning.

- **(RQ2)** *Which keywords are the most informative to identify self-admitted technical debt in issue tracking systems?* By extracting keywords of technical debt statements, we can understand better how developers declare technical debt in issue tracking systems. The summarized keywords are also helpful to developers for understanding and identifying SATD within issues. Overall, understanding these keywords allow us to *explain* how the classifier works.
- **(RQ3)** *How generic is the classification approach among projects and issue tracking systems?* Different projects use different issue tracking systems (e.g. Jira and Google Monorail) and are maintained by different communities (e.g. Apache and Google). Thus, we need to evaluate how far our results can be applicable to different projects and issue tracking systems. This research question is concerned with the *generalizability* of machine learning approaches.
- **(RQ4)** *How much data is needed for training the machine learning model to accurately identify self-admitted technical debt in issues?* Intuitively, training a machine learning classifier on a bigger dataset leads to better accuracy. However, manually annotating SATD in issue tracking systems is a time-consuming task. Therefore, we ask RQ4 to determine the most suitable size for a training dataset, which can achieve the best classification *accuracy* with a minimum amount of effort.

3.1 Approach Overview

Figure 1 presents an overview of our approach. The first step is to collect issue sections from projects that use Jira and Google Monorail. Subsequently, collected issue sections are filtered to remove impertinent data. Then, issue sections are manually classified regarding their SATD. Finally, the machine learning models are trained on the manually classified dataset and executed on the whole dataset. Each of these steps is elaborated in the following sub-sections; the last step is the most complex, so it is explained in more depth.

3.2 Data Collection

To automatically identify SATD in issues, we will use supervised machine learning classifiers (Shalev-Shwartz and Ben-David 2014), which depend on training data that are

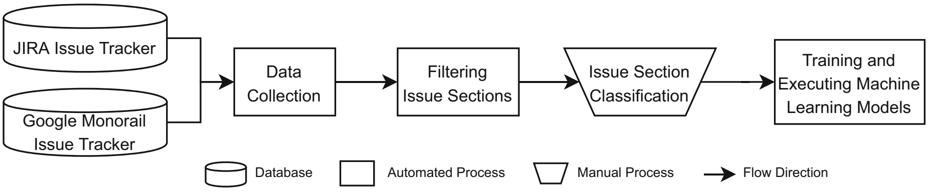

Fig. 1 The framework of our approach

Table 1 Details of chosen projects

Project	Project details				Classification details		
	Issue tracker	Languages	SLOC	# Issues	# Analyzed sections	# SATD sections	% SATD sections
Camel	Jira	Java	1,525k	14,411	2,792	377	13.5%
Chromium	Google	C++, C, and JavaScript	22,472k	1,079,511	3,435	264	6.7%
Gerrit	Google	Java	455k	12,711	2,812	195	6.9%
Hadoop	Jira	Java	3,409k	16,808	4,515	831	18.4%
HBase	Jira	Java	912k	24,342	4,936	688	13.9%
Impala	Jira	C++, Java, and Python	640k	9,733	1,934	355	18.4%
Thrift	Jira	C++, Java, and C	294k	5,196	2,756	567	20.6%
Average					3,311	468	14.1%
Total					23,180	3,277	

manually collected and classified. Thus, we first need to manually collect and analyze issues from different issue tracking systems.

Two of the most mainstream issue tracking systems are Jira and Google Monorail. Furthermore, projects, such as Camel, Chromium, Gerrit, and Hadoop, in the Apache and Google ecosystems have been commonly used in other SATD studies (Potdar and Shihab 2014; Wehaibi et al. 2016; Zampetti et al. 2018). Finally, projects in the Apache and Google ecosystems are of high quality and supported by mature communities (Smith 2018). Thus, we looked into projects in the Apache and Google ecosystems. Projects from the two ecosystems use two different mainstream issue tracking systems, namely: the Jira issue tracking system[2] and the Google Monorail issue tracking system.[3] To select projects pertinent to our study goal, we set the following criteria:

1. Both the issue tracking system and the source code repository are publicly available.
2. They have at least 200,000 source lines of code (SLOC) and 5,000 issues in the issue tracking systems. This is to ensure sufficient complexity.

There are over 400 projects in the Apache and Google ecosystems. However, training machine learning models requires manually analyzing issues to create the training dataset. Therefore, we randomly selected a sample of seven projects to be used for training and testing the machine learning models. This number of projects is similar to other SATD studies, which analyzed from four to ten projects (Potdar and Shihab 2014; Wehaibi et al. 2016; Maldonado et al. 2017). The details of the chosen projects are shown in Table 1. We analyzed the latest released versions on May 7, 2020. The number of source lines of code (SLOC) is calculated using the LOC tool.[4] It should be noted that although Chromium has significantly more issues than other projects, we found through manual inspection that the issues of Chromium are similar to the issues of other projects (and esp. Gerrit) with respect to the creation and discussion of issues. This is further supported by the fact that the same

[2] https://www.atlassian.com/software/jira

[3] https://bugs.chromium.org/

[4] https://github.com/cgag/loc

percentage (approx. 7%) of issue sections from Chromium and Gerrit were classified as SATD sections; thus the issues of Chromium and Gerrit are maintained similarly.

3.3 Filtering Issue Sections

In issue tracking systems, in addition to the comments submitted by software developers, some comments are automatically generated by bots (e.g., the Jenkins bot generates comments to report the results of the Jenkins build). Therefore, we filtered out comments that are automatically generated by bots. Specifically, we first obtain the 100 most active users by ordering the number of comments submitted per user. Then we identify the bots' usernames (such as *Hadoop QA* and *Hudson*) by checking comments submitted by these 100 most active users. After that, we removed all comments posted by bot users according to the list of bots' usernames.

Additionally, software developers sometimes attach source code in issues for various reasons. Because we focus on the SATD in issues rather than in source code, we removed source code in issues using a set of regular expressions.[5]

3.4 Issue Section Classification

Before training machine learning models to automatically identify SATD in issues, we need to inspect the collected sections within issues and manually classify them. Since software developers might discuss several types of SATD in the same issue, treating a whole issue as a single type of technical debt may be inaccurate. For example, software developers discussed both code debt and test debt in issue HADOOP-6730.[6] To accurately identify SATD in issue tracking systems, similarly to our previous study (Li et al. 2020), we treat issue summaries, descriptions, and comments as separate sections, and classify each section individually.

Since manual classification is extremely time consuming, we are not able to analyze all issues. Thus, we calculated the size of the statistically significant sample based on the total number of issues of each project with a confidence level of 95% and a confidence interval of 5%. We found that the sizes of statistically significant samples range from 358 to 384. Therefore, we randomly selected 600 issues from each project for analysis to ensure each sample size is greater than the statistically significant sample size. We then decomposed issues into sections (summaries, descriptions, and comments) for manual classification. Because each issue contains one summary, one description, and several comments, the number of decomposed sections per issue varies. This step resulted in 23,180 issue sections for analysis (see Table 1).

We used an open-source text annotation tool (Doccano[7]) to annotate sections using an existing classification framework from our previous work (Li et al. 2020). This classification framework contains several types and indicators of SATD (see Table 2), and was based on the original framework by Alves et al. (2014). Based on this classification framework, if a section matches an indicator (i.e. it indicates a certain type of SATD), we classify the section as the corresponding type. Classifying SATD into these types allows for a comparison between classifying SATD in issue tracking systems versus SATD in source code comments (see Section 5).

[5] https://github.com/yikun-li/satd-issue-tracker-data

[6] https://jira.apache.org/jira/browse/HADOOP-6730

[7] https://github.com/doccano/doccano

Table 2 Types and indicators of self-admitted technical debt

Type	Indicator	Definition	#	#	%
Architecture debt	Violation of modularity	Because shortcuts were taken, multiple modules became interdependent, while they should be independent.	46	87	2.7
	Using obsolete technology	Architecturally-significant technology has become obsolete.	41		
Build debt	Over- or under-declared dependencies	Under-declared dependencies: dependencies in upstream libraries are not declared and rely on dependencies in lower level libraries. Over-declared dependencies: unneeded dependencies are declared.	25	64	2.0
	Poor deployment practice	The quality of deployment is low that compile flags or build targets are not well organized.	39		
Code debt	Complex code	Code has accidental complexity and requires extra refactoring action to reduce this complexity.	30	1246	38.0
	Dead code	Code is no longer used and needs to be removed.	121		
	Duplicated code	Code that occurs more than once instead of as a single reusable function.	40		
	Low-quality code	Code quality is low, for example because it is unreadable, inconsistent, or violating coding conventions.	856		
	Multi-thread correctness	Thread-safe code is not correct and may potentially result in synchronization problems or efficiency problems.	40		
	Slow algorithm	A non-optimal algorithm is utilized that runs slowly.	159		
Defect debt	Uncorrected known defects	Defects are found by developers but ignored or deferred to be fixed.	25	25	0.8
Design debt	Non-optimal decisions	Non-optimal design decisions are adopted.	935	935	28.5
Documentation debt	Low-quality documentation	The documentation has been updated reflecting the changes in the system, but quality of updated documentation is low.	342	486	14.8
	Outdated documentation	A function or class is added, removed, or modified in the system, but the documentation has not been updated to reflect the change.	144		

Table 2 (continued)

Type	Indicator	Definition	#	#	%
Requirement debt	Requirements partially implemented	Requirements are implemented, but some are not fully implemented.	67	96	2.9
	Non-functional requirements not being fully satisfied	Non-functional requirements (e.g. availability, capacity, concurrency, extensibility), as described by scenarios, are not fully satisfied.	29		
Test debt	Expensive tests	Tests are expensive, resulting in slowing down testing activities. Extra refactoring actions are needed to simplify tests.	28	338	10.3
	Flaky tests	Tests fail or pass intermittently for the same configuration.	83		
	Lack of tests	A function is added, but no tests are added to cover the new function.	158		
	Low coverage	Only part of the source code is executed during testing.	69		

Subsequently, all issue sections were divided into four subsets and assigned randomly for analysis to four independent researchers, who are different from the authors of this paper. We selected the four independent researchers using convenience sampling; this resulted in an average of 5795 issue sections per independent researcher. To prepare the independent researchers for this task, we gave them a tutorial about SATD and explained to them the definitions of the different types and indicators of SATD in the classification framework (shown in Table 2). Moreover, each type of SATD was supported with examples from multiple projects in issues. This facilitated understanding each type of technical debt.

The next step for each independent researcher was to analyze and manually classify 200 issue sections (according to the classification framework in Fig. 1). These results were compared and discussed with the first author to ensure that the annotations of each independent researcher align with the definitions of the types of SATD in the classification framework. This process was repeated three times for each independent researcher to ensure a better and uniform understanding of the types of SATD.

Finally, all four independent researchers finished classifying the issue sections in his/her subset; thereafter we measured the level of agreement between the classifications of independent researchers and the first author using Cohen's kappa coefficient (Fleiss et al. 1981), which is commonly used to measure inter-rater reliability. This is useful to determine the risk of bias on the reliability of the classification. Specifically, we created four statistically significant samples corresponding to the four independent researchers with a confidence level of 95% and a confidence interval of 5%. Because each independent researcher classified on average 5795 issue sections, the size of each statistically significant sample is calculated to be 360 issue sections. Then the first author analyzed the statistically significant samples independently and the Cohen's kappa coefficient between the independent researchers and the first author was calculated. If the Cohen's kappa coefficient was below 0.7, the independent researcher discussed with the first author about discrepancies and subsequently reanalyzed all the issue sections in his/her subset. Then the first author analyzed another statistically significant sample and Cohen's kappa was calculated again. This process was repeated until the Cohen's kappa was above 0.7.

3.5 Training and Executing Machine Learning Models

3.5.1 Machine Learning Models

In order to accurately identify SATD in issues, we implement several machine learning approaches and compare their ability in identifying technical debt in issues. We describe the approaches and why we selected them below:

- **Traditional machine learning approaches (SVM, NBM, kNN, LR, RF)**: Support Vector Machine (SVM) (Sun et al. 2009), Naive Bayes Multinomial (NBM) (McCallum et al. 1998), k-Nearest Neighbor (kNN) (Tan 2006), Logistic Regression (LR) (Genkin et al. 2007), and Random Forest (RF) (Xu et al. 2012) classifiers are widely used in text classification tasks (Kowsari et al. 2019) due to their good classification accuracy. Moreover, the results of current studies on SATD identification (Maldonado et al. 2017; Huang et al. 2018; Flisar and Podgorelec 2019) show that these approaches achieve good accuracy in classifying SATD in source code comments. Thus, these approaches have potential to also achieve good accuracy when classifying SATD in issue trackers. Therefore, we train all of these classifiers using the implementation in Sklearn[8] using Bag-of-Words (BoW) with default settings and compare their accuracy.
- **Text Graph Convolutional Network (Text GCN)**: Text GCN generates a single large graph from the corpus and classifies text through classifying graph nodes using a graph neural network (Yao et al. 2019). This approach achieves promising performance in text classification tasks by outperforming numerous state-of-the-art methods (Yao et al. 2019).
- **Text Convolutional Neural Network (Text CNN)**: Text CNN is a simple one-layer CNN proposed by Kim (2014), that achieved high accuracy over the state of the art. The details of this approach are presented in more detail, as they are background knowledge for understanding some of the results in Section 4. The architecture of the model is presented in Fig. 2. The input issue section is first tokenized and converted into a matrix using an n-dimensional word embedding (see Section 3.5.4). For example, in Fig. 2, the input issue section is *'document should be updated to reflect this'*, which is represented as a 7 × 5 matrix because the issue section contains 7 words and the dimensionality of the word embedding is 5. Then the matrix is regarded as an image, and convolution operation is performed to extract the high level features. Because each row in the issue section matrix represents a word and the depth of the filter must be the same as the depth (width) of the input matrix, only the height of the filter can be adjusted, which is denoted by region size. It is important to note that multiple filters with different region sizes are applied to the issue section matrix to extract multiple features. In Fig. 2, the model adopts three filter region sizes (i.e., 1, 2, and 3) and three filters per region size. Applying the three filter region sizes 1, 2, and 3 on the input issue section produces nine feature maps with the sizes of 7, 6, and 5. For example, with a filter region size of 1, the convolution operation needs to be applied on every row (i.e. every word) of the input issue section, and thus producing a feature map with size of 7. After that, to

[8] https://scikit-learn.org

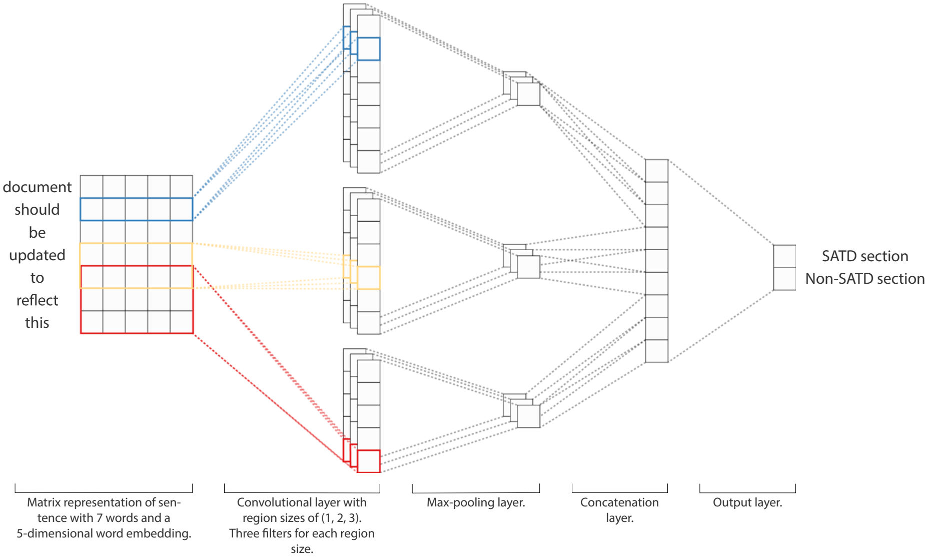
document
should
be
updated
to
reflect
this
SATD section
Non-SATD section
Matrix representation of sentence with 7 words and a 5-dimensional word embedding.
Convolutional layer with region sizes of (1, 2, 3). Three filters for each region size.
Max-pooling layer.
Concatenation layer.
Output layer.

Fig. 2 Architecture of the CNN model

make use of the information from each feature map, 1-max-pooling (which computes the maximum value of each feature map) is applied to extract a scalar from each feature. Then the output features are concatenated and flattened to form the penultimate layer. Finally, the output layer calculates the probability of the section to be a SATD section using the softmax activation function. This approach has been proven to be accurate for identifying SATD in source code comments (Ren et al. 2019); thus it also has potential for accurately identifying SATD in issues.

In this study, we mainly use Python Keras[9] and Sklearn libraries[10] to implement machine learning approaches. The machine learning models are trained on the NVIDIA Tesla V100 GPU.

3.5.2 Baseline Approaches

To compare the accuracy between the different classification approaches, we implement two baseline approaches.

- **Baseline 1 (Random)**: This is a simple baseline approach, which assumes that the SATD detection is random. This random approach classifies sections as SATD sections randomly based on the probability of a section being a SATD section. For example, if 3,277 out of 23,180 sections are SATD sections in the training set, we assume the probability of a section being a SATD section is 14.1%. Then the random approach randomly classifies any section in the test set as SATD section corresponding to the calculated probability (14.1%).

[9] https://keras.io

[10] https://scikit-learn.org

- **Baseline 2 (Keyword)**: In the work of Potdar and Shihab (2014), they identified and summarized 62 SATD keywords, such as *fixme*, *ugly*, *temporary solution*, *this isn't quite right*, and *this can be a mess*. Those keywords were used for automatically identifying SATD comments (Bavota and Russo 2016). The SATD keyword-based method classifies a section as a SATD section when the section contains one or more of these SATD keywords.

3.5.3 Strategies for Handling Imbalanced Data

As we can see in Table 1, about 15% of the sections were classified as SATD sections in our issue dataset on average, which indicates that our dataset is imbalanced. Imbalanced data always hinders the accuracy of the classifiers as the minority class tends to be overlooked (Fernández et al. 2018). In order to improve the accuracy of machine learning models, we select the following three strategies for handing imbalanced data.

- **Easy Data Augmentation (EDA)**: this technique augments text data through synonym replacement, random insertion, random swap, and random deletion (Wei and Zou 2019). To balance the dataset, we generate and add synthetic SATD sections to the training data using the EDA technique.
- **Oversampling**: This method simply replicates the minority class to re-balance the training data. We replicate the SATD sections to balance the SATD sections and non-SATD sections before training.
- **Weighted loss**: This method first calculates weights for all classes according to their occurrence. High frequency in occurrence leads to low weight value. Then the loss of each measurable element is scaled by the corresponding weight value in accordance with the class. Weighted loss penalizes harder the wrongly classified sections from minority classes (i.e. false negative and false positive errors) during training of machine learning models to resolve the imbalanced data. This strategy is widely used for training CNN models on imbalanced datasets (Phan et al. 2017; Ren et al. 2019).

3.5.4 Word Embedding

Word embeddings refer to using a set of techniques mapping words to vectors of real numbers, which has been shown to boost the performance of text classification (Joulin et al. 2017; Wieting et al. 2015). We choose word embedding technique for word representation since it is able to learn word meaning and semantics. In this study, we train CNN models on top of four different word embeddings. The first word embedding is simply initialized randomly. The second word embedding is pre-trained by Mikolov et al. (2018) on Wikipedia and news dataset. The third word embedding is pre-trained by Efstathiou et al. (2018) on Stack Overflow posts, which is specific to the software engineering domain. The last word embedding is trained by us on our collected issue data (summaries, descriptions, and comments) using the fastText technique (Mikolov et al. 2018) with default settings.

3.5.5 Evaluation Metrics

We use four statistics evaluating the accuracy of different approaches: true positive (TP) represents the number of sections correctly classified as SATD sections; false positive (FP) represents the number of sections classified as SATD sections when they are not SATD sections; true negative (TN) represents the number of sections correctly classified as not SATD sections; false negative (FN) represents the number of sections classified as not SATD sections when they are SATD sections. Consequently, we calculate **precision** ($precision = \frac{TP}{TP+FP}$), **recall** ($recall = \frac{TP}{TP+FN}$), and **F1-score** ($F1 = 2 \times \frac{precision \times recall}{precision + recall}$).

The higher evaluation metric (i.e., precision, recall, or F1-score) means the better accuracy, whereas there is a trade-off between precision and recall. In general, F1-score gives us an overall accuracy combining precision and recall.

3.5.6 Keyword Extraction

To better understand how developers declare technical debt in issues (to answer RQ2), we extract SATD keywords using the approach proposed by Ren et al. (2019). This approach is built upon the CNN approach and is able to extract n-gram keywords using the backtracking technique. More specifically, after feeding a SATD section to the CNN model, the most important features are selected according to their weights. Then the corresponding filters are located via backtracking the selected features. Finally, the n-gram keywords in the fed SATD section are located based on the filter position information. In this study, we train the model on all seven projects' issue sections. Then we summarize unigram to five-gram SATD keywords based on the extracted keywords.

4 Results

4.1 (RQ1.1) Which Algorithms Have the Best Accuracy to Capture Self-Admitted Technical Debt in Issue Tracking Systems?

To evaluate the machine learning algorithms, we first combine all the issue sections from different projects, then shuffle and split the combined dataset into ten equally-sized partitions, while keeping the number of SATD sections approximately equal in all the partitions. We then select one of the ten subsets for testing and the rest of the nine subsets for training, and repeat this process for each subset and calculate the average precision, recall, and F1-score over ten experiments. This is called stratified 10-fold cross-validation. Table 3 shows the precision, recall, and F1-score of deep learning approaches (Text CNN and Text GCN), traditional machine learning approaches (SVM, NBM, kNN, LR, and RF), and baseline approaches (keyword-based and random). The best results are highlighted in bold.

As we can see in Table 3, Text CNN with randomized word embeddings achieves the highest average F1-score of 0.597. This contrasts earlier evidence (Kim 2014), where Text CNN with two pre-trained word embeddings (Wiki-news and StackOverflow-post word embeddings) degraded the model's predictive accuracy. It is also important to note that, Text CNN with the word embeddings trained specifically for the software engineering domain (i.e., the StackOverflow-post word embeddings) is still slightly worse than the random word embeddings on the average F1-score (0.590 and 0.597 respectively). Among the traditional machine learning techniques, NBM and LR achieve decent F1-scores of 0.529 and 0.515.

Observing Table 3, we also observe that most of the approaches achieve decent precisions. While SVM and RF achieve the two highest average precisions of 0.861 and 0.730, their average recalls are relatively poor (0.295 and 0.291 respectively). Moreover, we notice that the keyword-based method achieves an average precision of 0.515; this means that the SATD sections in source code comments and issue tracking systems share some similarities and the keywords from source code comments (see Potdar and Shihab (2014) indicating SATD, such as *fixme*, *ugly*, and *temporary solution*) are useful for detecting

Table 3 Comparison of precision, recall, and F1-score between machine learning and baseline approaches

Type	Classifier	Precision	Recall	F1-score	F1-score Imp. Over Random	F1-score Imp. Over Keyword
Deep learning	Text CNN (rand)	0.685	0.530	**0.597**	**4.3**×	**13.6**×
	Text CNN (wiki)	0.677	0.463	0.549	3.9×	12.5×
	Text CNN (SO)	0.651	0.541	0.590	4.2×	13.4×
	Text GCN	0.474	0.056	0.081	0.6×	1.8×
Traditional machine learning	SVM	0.861	0.179	0.295	2.1×	6.7×
	NBM	0.520	0.539	0.529	3.8×	12.0×
	kNN	0.582	0.029	0.055	0.4×	1.2×
	LR	0.643	0.430	0.515	3.7×	11.7×
	RF	0.730	0.182	0.291	2.1×	6.6×
Baseline	Random	0.140	0.139	0.139		
	Keyword	0.515	0.023	0.044		

SATD sections in issues. However, the recall of the keyword-based method is the lowest (0.023) among all methods, which might result from the low coverage of these source code comments keywords.

> *Text CNN with default settings using random word embeddings achieves the highest F1-score (0.597) among all approaches, followed by Text CNN with default settings using pre-trained StackOverflow-post word embeddings (0.590).*

4.2 (RQ1.2) How to Improve Accuracy of Machine Learning Model?

After establishing that the Text CNN approach achieves the highest F1-score among all approaches, we can further investigate its improvement. In the following sub-sections, we investigate handling imbalanced data, refining word embeddings and tuning CNN hyperparameters.

4.2.1 Handling Imbalanced Data

As we can see in Table 1, on average only 14.1% sections were classified as SATD sections in our issue dataset, which indicates that our dataset is imbalanced. Since the Text CNN approach is not designed for imbalanced data classification tasks, firstly we look into methods for handling imbalanced data. We train the Text CNN model with different word embeddings using the aforementioned imbalanced data handling techniques discussed in Section 3.5.3.

Table 4 presents the precision, recall, and F1-score improvement of Text CNN approach with different imbalanced data handling techniques. We can see that EDA always has a negative effect on training and degrades the F1-score by 7.9% on average. The other two imbalanced data handling techniques (i.e., oversampling and weighted loss) achieve a similar improvement on three different word embeddings, while the average F1-score

Table 4 Comparison of average precision, recall, and F1-score between different imbalanced data handling strategies

Method	Word embedding	Precision (average)	Recall (average)	F1-score (average)	F1-score Imp.
Default	Random	0.685	0.530	0.597	–
	Wiki-news	0.677	0.463	0.549	–
	StackOverflow-post	0.651	0.541	0.590	–
	Average	0.671	0.511	0.578	–
EDA	Random	0.606	0.470	0.529	−11.3%
	Wiki-news	0.556	0.406	0.469	−14.5%
	StackOverflow-post	0.604	0.595	0.599	1.5%
	Average	0.588	0.490	0.532	−7.9%
Oversampling	Random	0.573	0.717	0.636	6.5%
	Wiki-news	0.591	0.592	0.591	7.6%
	StackOverflow-post	0.610	0.618	0.612	3.7%
	Average	0.591	0.642	0.613	6.0%
Weighted loss	Random	0.555	0.735	0.632	5.8%
	Wiki-news	0.583	0.617	0.599	9.1%
	StackOverflow-post	0.591	0.640	0.613	3.8%
	Average	0.576	0.664	**0.614**	**6.2%**

improvement is 6.0% and 6.2% respectively. Since oversampling replicates the sections in minority classes, the training dataset using oversampling is significantly larger than the one using weighted loss. Thus, we notice that the time spent on training using oversampling is 30.8% longer than using weighted loss. Therefore, we choose weighted loss as the imbalanced data handling technique.

4.2.2 Refining Word Embeddings

In Table 4, we notice that the average F1-score is not improved using pre-trained word embeddings (i.e., Wiki-news and StackOverflow-post) compared to using Random word embedding. Thus, we train the word embeddings on our issue dataset using fastText technique (Mikolov et al. 2018) with the dimension size of word embeddings setting to 100, 200

Table 5 Comparison of average precision, recall, and F1-score between different word vector settings using weighted loss

Word embedding	Dimensionality	Precision (average)	Recall (average)	F1-score (average)
Random	300	0.555	0.735	0.632
Wiki-news	300	0.583	0.617	0.599
StackOverflow-post	200	0.591	0.640	0.613
Issue-tracker-data	100	0.647	0.686	0.664
	200	0.662	0.680	0.670
	300	0.648	0.703	**0.673**

and 300. Because fastText is able to learn subword information during word representation training (Bojanowski et al. 2017), we use it to train word embeddings. As can be seen in Table 5, we observe that the word embeddings trained on our dataset with different dimension sizes all outperform the Random word embeddings and pre-trained word embeddings (i.e., Wiki-news and StackOverflow-post) significantly. Besides, while increasing the size of dimensions, the F1-score slightly grows. Therefore, we use the word embeddings trained on our dataset and choose 300 as the dimension size.

4.2.3 Tuning CNN Hyperparameters

We follow the guidelines provided by Zhang and Wallace (2017) to tune the hyperparameters of the CNN model. We first conduct a line-search over the single filter region size (i.e., setting the region size to (1), (3), (5), and (7)) to find the single filter region size with the best accuracy. The results are reported in Table 6. As we can see, in this study, the single region size (3) outperforms other single region sizes.

Second, we further explore the effect of multiple region sizes using regions sizes near this single best size (3) according to the suggestion from the guidelines; this is because combing multiple filters using region sizes near the best size always results in better accuracy compared to only using single best region size (Zhang and Wallace 2017). Because we cannot enumerate all combinations of region sizes, we consider the combinations of region sizes of (1,2), (1,2,3), (2,3,4), (3,4,5), (1,2,3,4), (1,3,5,7), (2,4,6,8), (1,2,3,4,5), (1,2,3,5,7), (1,3,4,5,7), (1,3,5,7,9), (1,2,3,4,5,6), and (1,2,3,4,5,6,7). From the results in Table 6, we can see that among all combinations, (1,2,3) achieves the highest F1-score, followed by (1,2,3,4,5) and (1,2,3,5,7) by small margins (0.682, 0.680 and 0.680 respectively). Thus, we choose (1,2,3) as the region size.

Table 6 Comparison of average precision, recall, and F1-score between different filter region size settings using issue-tracker-data word embeddings and weighted loss

Type	Region size	Precision (average)	Recall (average)	F1-score (average)
Single	(1)	0.552	0.782	0.646
	(3)	0.638	0.707	**0.670**
	(5)	0.631	0.686	0.657
	(7)	0.642	0.665	0.652
Multiple	(1,2)	0.643	0.715	0.676
	(1,2,3)	0.657	0.711	**0.682**
	(2,3,4)	0.655	0.706	0.677
	(3,4,5)	0.648	0.703	0.673
	(1,2,3,4)	0.663	0.703	0.679
	(1,3,5,7)	0.675	0.677	0.675
	(2,4,6,8)	0.674	0.665	0.669
	(1,2,3,4,5)	0.678	0.685	0.680
	(1,2,3,5,7)	0.681	0.682	0.680
	(1,3,4,5,7)	0.667	0.686	0.676
	(1,3,5,7,9)	0.668	0.681	0.673
	(1,2,3,4,5,6)	0.669	0.691	0.678
	(1,2,3,4,5,6,7)	0.669	0.680	0.673

Table 7 Comparison of average precision, recall, and F1-score between different filter region size settings using issue-tracker-data word embeddings and weighted loss and setting multiple region size to (1, 2, 3)

Number of feature maps	Precision (average)	Recall (average)	F1-score (average)
50	0.640	0.713	0.674
100	0.657	0.711	0.682
200	0.685	0.689	**0.686**
400	0.675	0.699	0.685
600	0.677	0.671	0.683

Third, we investigate the effect of the number of feature maps for filter region size. We keep other settings constant and only change the number of feature maps relative to the default number of features 100. We set the number of features to 50, 100, 200, 400, and 600 under the aforementioned guidelines (Zhang and Wallace 2017). The result is demonstrated in Table 7. As we can see, while increasing the number of feature maps, the average F1-score is slightly improved until 200, so we set 200 as the number of feature maps.

4.2.4 Final Results After Machine Learning Optimization

To conclude our optimization, after handling imbalanced data, refining word embeddings, and tuning hyperparameters, we compare the precision, recall, and F1-score of our customized CNN approach against the Text CNN with default settings. The results indicate that our customized approach improves the recall and F1-score over the Text CNN with the default settings by 30.0% and 14.9% respectively. More specifically, on average it does not improve precision (0.685); the recall of our approach (0.689) increases by 30.0% compared to the original approach (0.530). Overall, our customized CNN approach improves the F1-score by 14.9% from 0.597 to 0.686 on average.

After handling imbalanced data, refining word embeddings, and tuning hyperparameters, our customized CNN approach improves the F1-score over the Text CNN with default settings by 14.9% from 0.597 to 0.686.

4.3 (RQ1.3) How Can Transfer Learning Improve the Accuracy of Identifying Self-Admitted Technical Debt in Issue Tracking Systems?

To leverage the knowledge in existing SATD datasets, we follow the transfer learning guidelines on text classification provided by Semwal et al. (2018). Specifically, Semwal *et al.* conducted a series of experiments and presented the cases and settings that could lead to a positive transfer (i.e. the transfer results in improved accuracy).

In this study, we explore how the accuracy of the CNN model for SATD identification can be further improved through transfer learning. As can be seen in Fig. 2, The CNN model consists of an embedding layer, a convolutional layer, a max-pooling layer, a concatenation layer, and an output layer. The embedding layer is responsible for converting words to n-dimensional vectors, which is pre-trained on various datasets in this work. The convolutional (C) layer plays a critical role in learning various kinds of features. The weights of its kernels are trainable and transferable. The max-pooling layer is a regular layer that down-samples the input representation. The concatenation layer is used for concatenating the inputs along a specified dimension. The output (O) layer is in charge of producing the final result. The weight and biases in the output layer are trainable and transferable. Because there are no trainable parameters (i.e. the parameters are fixed during training) in the embedding layer, max-pooling layer, and concatenation layer, we explore the transfer learning settings for the convolutional layer (C) and output layer (O).

To transfer the knowledge from the source domain to aid the target domain, we initialize the parameters of the target model to be trained with the parameters trained on the transfer learning source dataset. To avoid confusion, we use the same nomenclature as the work by Semwal et al. (2018). There are three settings for parameters during transfer learning:

- ✱: Parameters are not transferred, but are randomly initialized and allowed to fine-tune.
- 🔓: Parameters are transferred and allowed to fine-tune during training.
- 🔒: Parameters are transferred and frozen, i.e., they can not learn during training.

According to the transfer learning guideline (Semwal et al. 2018), we use the fine-tuning (🔓) setting for the convolutional (C) layer, and either transferred parameters for fine-tuning (🔓)) or random initialization for fine-tuning (✱) setting for the output (O) layer, which results in two combinations of settings (i.e., C🔓 O🔓 and C🔓 O✱).

Thus, we conduct experiments with the aforementioned two transfer learning settings on our issue dataset: a) transferring and allowing both the convolutional layer and output layer to fine-tune (C🔓 O🔓); b) only transferring and allowing the fine-tuning of the convolutional layer and randomizing the parameters in the output layer for fine-tuning (C🔓 O✱). Besides, we compare the results with learning from scratch (C✱ O✱). For the source dataset selection, we choose five datasets as the transfer learning source datasets:

- *Source code comment SATD dataset (CO-SATD)* (Maldonado et al. 2017): We chose this dataset, because it contains SATD in source code comments, which is highly similar to our issue SATD dataset. This dataset contains 62,566 comments, in which 4,071 comments were annotated as SATD comments.
- *Amazon review (AMZ2)* and *Yelp review (YELP2) datasets* (Zhang et al. 2015): AMZ2 contains 2,000,000 Amazon product reviews for each polarity. YELP2 includes 289,900 business reviews for each polarity. We selected these two datasets, because their size is significantly bigger than our dataset and because they are commonly used for text classification tasks (Semwal et al. 2018).
- *Jira issues sentiment (JIRA-SEN)* and *Stack Overflow posts sentiment (SO-SEN) datasets* (Ortu et al. 2016; Calefato et al. 2018): Both datasets are relatively small datasets (only containing 926 and 2,728 samples respectively). We chose these two datasets because they are in the software engineering domain.

To evaluate the efficiency of transfer learning, we still use stratified 10-fold cross-validation. We first train our model on these transfer learning source datasets individually and then retrain the models on our issue SATD dataset in Table 8 with the transfer learning settings (C🔓 O🔓 and C🔓 O✱). The results are illustrated in Table 8. We observe that applying transfer learning with JIRA-SEN and SO-SEN outperform CO-SATD, AMAZ2, and YELP2 with respect to F1-score. This indicates that sentiment information in software engineering domain could be useful for SATD identification. Moreover, only the F1-score achieved by JIRA-SEN using the setting (C🔓 O✱) outperforms training from scratch setting (C✱ O✱), which indicates positive transfers (Perkins et al. 1992). The F1-score is slightly improved from 0.686 to 0.691 by leveraging the JIRA-SEN dataset. Our findings

Table 8 Comparison of average precision, recall, and F1-score between different transfer learning settings (C🔓 O🔓 and C🔓 O✱) and without transfer learning setting (C✱ O✱)

Source Dataset	Setting	Precision	Recall	F1-score
–	C✱ O✱	0.686	0.689	0.686
CO-SATD	C🔓 O🔓	0.674	0.671	0.672
	C🔓 O✱	0.675	0.684	0.679
AMZ2	C🔓 O🔓	0.666	0.664	0.665
	C🔓 O✱	0.681	0.664	0.672
YELP2	C🔓 O🔓	0.670	0.677	0.673
	C🔓 O✱	0.689	0.677	0.681
JIRA-SEN	C🔓 O🔓	0.676	0.696	0.684
	C🔓 O✱	0.689	0.694	**0.691**
SO-SEN	C🔓 O🔓	0.682	0.686	0.683
	C🔓 O✱	0.685	0.685	0.685

here show that transfer learning could improve the F1-score of SATD identification in issue tracking systems, but it highly depends on the source dataset selection.

Furthermore, we investigate the effectiveness of transfer learning when data is insufficient for training the target model. We report the F1-score achieved by two transfer learning settings (C🔓 O🔓 and C🔓 O✱) versus learning from scratch (C✱ O✱) on the training dataset containing 0 to 900 issue sections in Table 9. From the results, we find that when training data is extremely scarce (less than 200 issue sections for training), transferring both the convolutional layer and output layer (C🔓 O🔓) performs best by selecting CO-SATD or JIRA-SEN as the transfer learning source dataset. When there is sightly more training data (between 300 and 600 sections for training), only transferring the convolution layer (C🔓 O✱) achieves the highest F1-score by using CO-SATD as the transfer learning source dataset. When more training data is available (more than 700 sections for training),

Table 9 F1-score of different transfer learning settings with different number of training sections from 0 to 900. (✱: randomly initialize and allow to fine-tune parameters; 🔓: allow to fine-tune transferred parameters)

Source	Setting	Number of issue sections used for training									
dataset		0	100	200	300	400	500	600	700	800	900
–	C✱ O✱	0.157	0.357	0.386	0.424	0.451	0.474	0.488	0.501	0.507	0.523
CO-SATD	C🔓 O🔓	0.260	**0.395**	**0.402**	0.412	0.425	0.435	0.442	0.445	0.458	0.461
	C🔓 O✱	0.202	0.367	0.393	**0.425**	**0.469**	**0.478**	**0.493**	0.497	**0.518**	0.515
AMZ2	C🔓 O🔓	0.170	0.289	0.295	0.305	0.334	0.345	0.361	0.364	0.380	0.388
	C🔓 O✱	0.145	0.349	0.386	0.400	0.438	0.451	0.461	0.475	0.487	0.497
YELP2	C🔓 O🔓	0.051	0.288	0.290	0.300	0.323	0.329	0.352	0.358	0.378	0.389
	C🔓 O✱	0.147	0.341	0.371	0.398	0.439	0.462	0.477	0.481	0.506	0.500
JIRA-SEN	C🔓 O🔓	**0.273**	0.337	0.352	0.379	0.411	0.419	0.435	0.449	0.468	0.473
	C🔓 O✱	0.169	0.354	0.391	0.410	0.452	0.472	0.486	**0.503**	0.517	**0.531**
SO-SEN	C🔓 O🔓	0.256	0.296	0.301	0.330	0.356	0.371	0.384	0.393	0.418	0.428
	C🔓 O✱	0.089	0.350	0.389	0.410	0.458	0.477	0.484	0.498	0.515	0.517

the model without transferred parameters (C✱ O✱) likely starts to outperforms others by using CO-SATD or JIRA-SEN as the transfer learning source dataset.

> *Using Jira issue sentiment dataset (JIRA-SEN) as the source dataset for transfer learning, improves the F1-score but only slightly (from 0.686 to 0.691) with the setting C🔓 O✱. When training data is insufficient, using Jira issue sentiment dataset (JIRA-SEN) or source code comment SATD dataset (CO-SATD) as the source dataset could boost the F1-score.*

4.4 (RQ2) Which Keywords Are the Most Informative to Identify Self-Admitted Technical Debt in Issue Tracking Systems?

We first summarize a list of top SATD keywords (unigram to five-gram) according to the extracted keywords. The results are shown in Table 10. Note that keywords which are similar to the keywords in source code comments (see Potdar and Shihab 2014) are underlined. The first author manually linked the keywords to types and indicators of SATD by checking the types and indicators of issue sections that contain the keywords based on the existing classification framework from our previous work (Li et al. 2020) (see Table 2). Subsequently, the other authors checked and confirmed the correlation between keywords with types and indicators. We can see that our keywords are intuitive and potentially indicate the types and indicators of the technical debt; the types are shown in Tables 10 and 11 while the definitions of types and indicators are included in Table 2.

We also summarize the top keywords for each project. Table 12 presents the top 20 keywords on each project. We note that the keywords are highlighted in bold if the number of projects where the keywords occur is greater than or equal to 4 (i.e. it appears in over half of projects). From the table, we observe that nine keywords (e.g., leak, confusing, and unnecessary) are shared by different projects frequently. We also find two projects (i.e. Chromium and HBase), one from each issue tracking system, containing all these nine common keywords; this indicates that technical debt is admitted similarly in different issue tracking systems although each project has some unique keywords.

To gain a better understanding of how many keywords are shared by different projects, we first calculate the average number of extracted keywords from different projects and use the top 10% (i.e., 2628) of the average number of most important keywords for analysis. Then we plot a chord diagram (Gu et al. 2014) illustrating the relations (i.e. the number of common keywords) between pairs of projects.

Figure 3 shows the number of common keywords between different projects. In the figure, the absolute value (i.e., the number of common keywords) measures the strength of the relation, and links are more transparent if the relations are weak (i.e., the number of common keywords is in the 30th percentile). Moreover, the links between projects using Jira issue tracking system are colored red, and between projects using Google issue tracking system are colored yellow. The links between the two issue tracking systems are colored blue.

We can see that although relations between pairs of projects seem equally strong, according to the sum of the number of shared keywords between projects, only Gerrit and Chromium share less than 1200 keywords with other projects; this indicates that these two

Table 10 Extracted top n-gram keywords from uni- to five-gram

Unigram keyword	Bigram keyword	Trigram keyword
flaky (test)	too much	get rid of
leak (code)	not used (code)	not thread safe (requirement)
unused (code)	more readable (code)	clean up code (code)
unnecessary (code)	more efficient (code)	not done yet (requirement)
typo (code/documentation)	dead code (code)	avoid extra seek
slow (code)	infinite loop (code)	reduce duplicate code (code)
redundant (code)	too long (documentation)	no longer needed (code/documentation)
confusing (code)	not implemented (requirement)	not supported yet (requirement)
nit	less verbose (code)	documentation doesn’t match (documentation)
ugly	more robust (design)	short term solution
simplify (code)	speed up (code)	spurious error messages (code)
misleading (documentation)	missing documentation (documentation)	it’d be nice
Four-gram Keyword	Five-gram Keyword	
please add a test (test)	wastes a lot of space	
would significantly improve performance (code)	there is no unit test (test)	
makes it much easier	lead to huge memory allocation (design)	
avoid calling it twice (code)	test doesn’t add much value (test)	
takes a long time (code)	some holes in the doc (documentation)	
good to have coverage (test)	by hard coding instead of (code)	
makes it very hard	should be updated to reflect (documentation)	
patch doesn’t apply cleanly (code)	more tightly coupled than ideal (design)	
it’s not perfectly documented (documentation)	any chance of a test (test)	
need to update documentation (documentation)	should improve a bit by	
make it less brittle (design)	it’d help code readability if (code)	
documentation does not mention (documentation)	solution won’t be really satisfactory	

Google projects have the weakest connection with others. Besides, Chromium have strong relations with three projects using Jira. Moreover, we can observe that the relation between the two Google projects is also weak, while there is no weak relation between projects using Jira. This entails that there are discernible differences between SATD keywords in

Table 11 Correlation between keywords and types/indicators of SATD

Type indicator		Keyword	Example
Code	Complex Code	simplify redundant less verbose	“*That can **simplify** the logic there.*” - [HADOOP-10295]
	Dead Code	unused unnecessary not used dead code no longer needed	“*I would like to remove this as its **no longer needed**, and also its code is not complete.*” - [Camel-8174]
	Duplicated Code	reduce duplicate code	
	Low-Quality Code	typo leak confusing more readable infinite loop spurious error messages avoid calling it twice patch doesn’t apply cleanly it’d help code readability if	“*...to make their code **more readable**. I would like to see something like this in the API...*” - [HBase-1990]
	Slow Algorithm	slow more efficient speed up would significantly improve performance takes a long time	“*Rowlocks should use rentHashMap as it is much **more efficient** than Collections. synchronizedMap(HashMap)*” - [HBase-798]
Design	Non-Optimal Decision	more robust make it less brittle lead to huge memory allocation more tightly coupled than ideal	“*...didn’t tackle those pieces yet. They also seem **more tightly coupled than ideal**.*” - [HBase-12749]
Requirement	Requirement Partially Implemented	not implemented not done yet not supported yet	“***Not implemented** reached in virtual void...*” - [Chromium-43196]
Documentation	Outdated Documentation	missing documentation documentation doesn’t match some holes in the doc it’s not perfectly documented need to update documentation documentation does not mention should be updated to reflect	“*I am using this opportunity to fill in **some holes in the doc**...*” - [Impala-991]

Table 11 (continued)

Type indicator		Keyword	Example
	Low-Quality Documentation	typo confusing simplify misleading too long	*'Default searches documentation **misleading** about single-change search match behaviour in UI"* - [Gerrit-8592]
Test	Lack of Tests	please add a test there is no unit test any chance of a test	*"It looks good to me except these. **Please add a test** case for the code change..."* - [Hadoop-12155]
	Low Coverage	good to have coverage test doesn't add much value	*"this **test doesn't add much value**, does it?"* - [Gerrit-6524]
	Flaky Tests	flaky	

projects using Google Monorail (like Gerrit and Chromium) and Jira. The projects using Jira might have more common ways to declare SATD compared to projects using Google Monorail. This could be due to two potential reasons: 1) many developers in the Apache ecosystem work in multiple projects, so they use similar keywords across those projects; 2) the number of developers involved in the Google ecosystem is higher compared to the Apache ecosystem, so the variety of keywords is also higher.

In Tables 10 and 12, we can also observe that our keywords cover a few of the keywords from source code comments (see Potdar and Shihab 2014), such as 'ugly', 'inconsistent', 'bad', 'is wrong', 'get rid of', and 'clean up code', which are underlined in the table. In addition to these common keywords, there are three more common keywords not listed in the table due to space limitation: 'hacky', 'bail out', and 'crap'. Overall, we conclude that some keywords indicate SATD in both source code comments and issue tracking systems.

We find that extracted keywords are intuitive and potentially indicating types and indicators of SATD. We also observe that although different projects share a great number of SATD keywords, projects using different issue tracking systems have less common keywords compared to projects using the same issue tracking system. Source code comments and issue tracking systems have some common SATD keywords.

4.5 (RQ3) How Generic Is the Classification Approach Among Projects and Issue Tracking Systems?

In order to investigate the generalisability of our SATD detector over projects, we choose one project as the test project and the rest of the six projects as training projects using the configurations in Section 4.3. We then repeat this process for each project and calculate the average precision, recall, and F1-score over seven experiments. We call this leave-one-out cross-project validation. The results including F1-score, precision, and recall are presented in Table 13. We observe that, our approach achieves the average F1-score of 0.652, ranging between 0.561 to 0.709. In comparison with the average F1-score achieved using stratified

Table 12 Extracted top n-gram keywords per project

Camel	Chromium	Gerrit
leak	**leak**	**confusing**
typo	**flaky**	**typo**
confusing	**slow**	**flaky**
verbose	**unnecessary**	**unused**
deprecated	**simplify**	**bad**
dead	**redundant**	**slow**
slow	**typo**	truncated
unnecessary	truncated	unnecessarily
document this	**ugly**	not implemented yet
avoid	not implemented	**leak**
todo	**unused**	misleading
improve documentation	**bad**	documentation is wrong
complicated	**confusing**	coverage
remove ugly warnings	odd	complicated
thread safe	the short term	performance degradation
reuse	clean up code	documentation doesn’t
missing	too verbose	undocumented
rid of	**expensive**	**ugly**
improve exception message if failed	isn’t implemented	reword documentation
improve performance	too much	ambiguous

Hadoop	HBase	Impala	Thrift
unnecessary	**flaky**	**flaky**	**unused**
unused	**unused**	**slow**	**leak**
typo	nit	**unnecessary**	**unnecessary**
redundant	**typo**	coverage	**typo**
nit	**leak**	**confusing**	**redundant**
leak	**ugly**	**simplify**	**confusing**
slow	**redundant**	misleading	**simplify**
flaky	**unnecessary**	excessive	**flaky**
readability	**confusing**	overhead	coverage
clean up code	too much	avoid	thread-safe
complicated	**bad**	**expensive**	spurious
spurious	**slow**	improve error message	inconsistent
reuse	**expensive**	**redundant**	abstract
bad	misleading	rework	redundancy
ugly	avoid	thread-safe	**ugly**
not used	**simplify**	reduce duplicate code	outdated
cover	overhead	readability	missing
rid of	dead lock	difficult	performance regression
expensive	readability	wasted space	extra
thread-safe	rid of	verbose	unstable

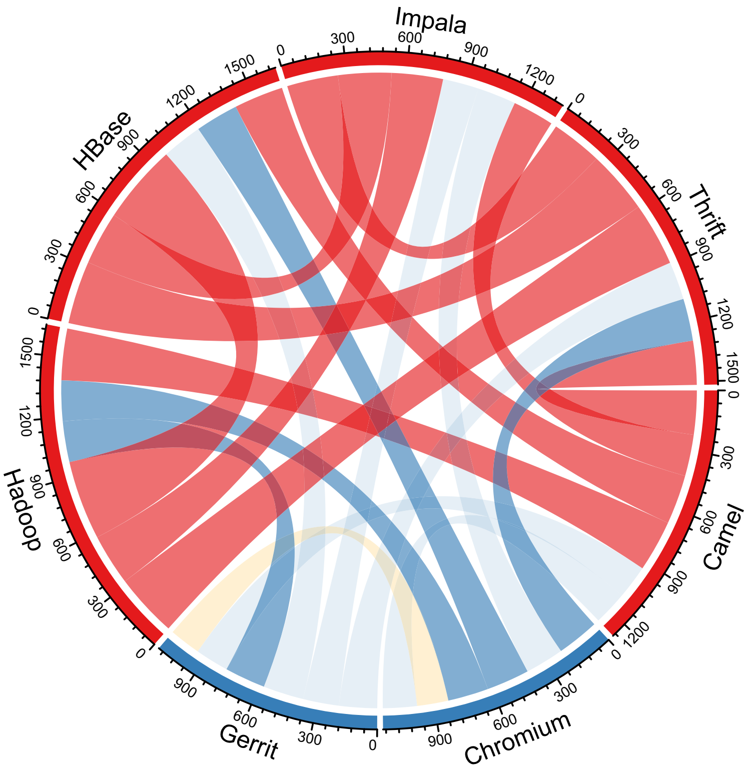

Fig. 3 Relations (the number of common keywords) between different projects

10-fold cross-validation (i.e., 0.691) in Section 4.3, the average F1-score decreases by a small margin (5.6%) from 0.691 to 0.652.

The second half of RQ3 concerns evaluating the generalizability of our CNN model (see Section 4.3) over issue tracking systems. We first choose one issue tracking system as the training system and the other one as the testing system. We then use issue sections from the projects using the training system for training the model and issue sections from the other projects using the testing system for testing. Then we switch the usage of two issue tracking systems. We call this leave-one-out cross-issue-tracker validation. Table 14

Table 13 Precision, recall, and F1-score when using six projects for training and the rest project for testing

Project	Precision	Recall	F1-score
Camel	0.719	0.647	0.681
Hadoop	0.618	0.761	0.682
HBase	0.651	0.648	0.649
Impala	0.697	0.721	0.709
Thrift	0.693	0.668	0.679
Chromium	0.659	0.556	0.603
Gerrit	0.481	0.671	0.561
Avg.	0.645	0.667	0.652

Table 14 Precision, recall, and F1-score when using projects from the same issue tracking system for training and a projects in the other issue tracking system for testing

Issue	Project	Precision		Recall		F1-score	
tracker	(target)	Result	Diff.	Result	Diff.	Result	Diff.
		Trained on projects using Google issue tracker					
Jira	Camel	0.553	−23.0%	0.538	−16.8%	0.545	−19.9%
	Hadoop	0.526	−14.8%	0.611	−19.7%	0.566	−17.0%
	HBase	0.506	−22.2%	0.610	−5.8%	0.553	−14.7%
	Impala	0.559	−19.7%	0.619	−14.1%	0.588	−17.0%
	Thrift	0.589	−15.0%	0.590	−11.6%	0.590	**−13.1%**
	Avg.	0.546	−18.9%	0.593	−13.6%	0.568	−16.3%
		Trained on projects using Jira issue tracker					
Google	Chromium	0.591	−10.3%	0.537	−3.4%	0.563	−6.6%
	Gerrit	0.488	1.4%	0.569	−15.2%	0.526	**−6.2%**
	Avg.	0.539	−4.4%	0.553	−9.3%	0.544	−6.4%

shows the results (i.e., precision, recall, and F1-score). The average precision, recall, and F1-score of projects are calculated and compared with the results obtained from leave-one-out cross-project validation (in Table 13). We find that the average F1-score achieved by the leave-one-out cross-issue-tracker validation is relatively worse compared to the leave-one-out cross-project validation: we achieve 6.4% and 16.3% decrease for models training on Jira and Google issue tracking systems respectively.

Our approach achieves the average F1-score of 0.652, ranging between 0.561 to 0.709, when applying leave-one-out cross-project validation. The average F1-score achieved by the leave-one-out cross-issue-tracker validation is declined by 6.4% and 16.3% for models training on Jira and Google issue tracking systems in comparison with leave-one-out cross-project validation.

4.6 (RQ4) How Much Data Is Needed for Training the Machine Learning Model to Accurately Identify self-admitted technical debt in Issues?

To answer this research question, we train our CNN model on datasets with a variety of sizes (number of issue sections). More specifically, we first combine all the issue sections from different projects, then shuffle and split the combined dataset into ten equally-sized partitions for 10-fold cross-validation. After that, we select one of the ten subsets for testing and the rest of the nine subsets for training. Because we want to train the model on the datasets with a variety of sizes, we create an empty training dataset, add 100 issue sections to the training dataset each time, and train our model on the created training dataset. Since we have 23,180 issue sections in total, the subset for training contains $23180 \times \frac{9}{10} \approx 20862$ issue sections. For each fold, we train the model on the dataset whose size increases from 100 to 20,862 at an interval of 100. Thus, we perform $\frac{20862}{100} + 1 \approx 209$ experiments for each fold. This process is repeated ten times for each one of the ten folds. In total, $10 \times 209 = 2090$

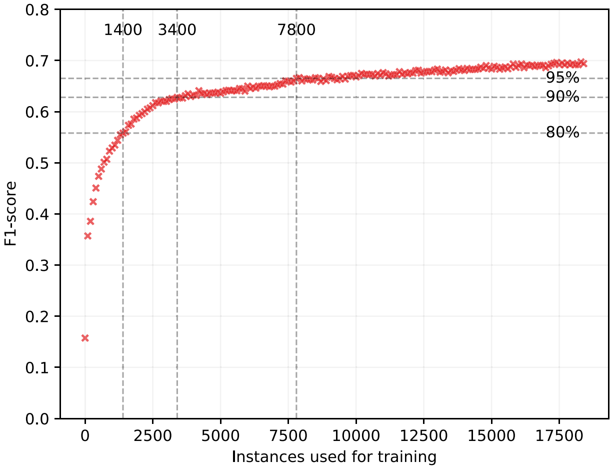

Fig. 4 F1-score achieved by incrementally adding 100 issue sections into the training dataset

experiments are carried out and the average F1-score for different sizes of training dataset over ten folds is calculated.

The result of the average F1-score achieved while increasing the size of the training dataset is shown in Fig. 4. We can observe that, while increasing the size of the training dataset from 100 to 2500 sections, the average F1-score goes up dramatically. After the 2500 sections, the average F1-score improves slowly. When the training dataset contains 20,862 sections, the highest F1-score is achieved. Besides, we find that in order to achieve 80%, 90%, or 95% of the highest average F1-score, 1400, 3400, or 7800 sections are needed respectively (namely 6.7%, 16.3%, or 37.4% of training data).

The average F1-score grows dramatically while increasing the size of the training dataset from 100 to 2500 sections. After 2500 sections, the average F1-score improves slowly. In order to achieve 80%, 90%, or 95% of the highest average F1-score, 1400, 3400, or 7800 sections are needed respectively (namely 6.7%, 16.3%, or 37.4% of training data).

5 Discussion

5.1 Differences Between Identifying SATD in Source Code Comments and Issue Tracking Systems

In recent years, a number of studies have investigated training machine learning models to automatically identify SATD in source code comments (Maldonado et al. 2017; Flisar and Podgorelec 2019; Ren et al. 2019). In contrast, our study focuses on SATD in issue tracking systems. It is important to investigate the differences between identifying SATD in these two sources, because it could help us better understand the nature of SATD and build machine learning models to more accurately identify it in these and other sources. Thus, in the following, we compare and discuss the differences between identifying SATD in code comments and issues. Based on our experience gained during the manual issue analysis, we conjecture two major reasons for the difference among issues and source code comments, as explained in the following paragraphs.

First, **the diversity of issues is much higher than source code comments.** In the work by Steidl et al. (2013), source code comments are categorized into seven types, namely *code, copyright, header, member, inline, section,* and *task comments*. The first type, *code comments*, refer to code that is commented out; this certainly does not indicate technical debt. The types *copyright, header, member, inline,* and *section comments* are descriptive comments, which have a small chance of indicating technical debt. The only type that mostly concerns SATD is *task comments*, as these are notes left by developers indicating code that needs to be implemented, refactored, or fixed.

In contrast, the scope of information stored in issue tracking systems is much broader as compared to code comments. Merten et al. (2015) categorized issue sections into 12 types, namely *issue description, request, issue management, scheduling, implementation proposal, implementation status, clarification, technical information, rationale, social interaction, spam,* and *others*. Apart from *issue management, scheduling, social interaction,* and *spam*, all other 8 types of issue sections could potentially indicate technical debt. Because the diversity of issue types is much higher than source code comments, it is harder for machine learning models to accurately capture SATD in issue types than in source code comments. Furthermore, we present a comparison between the key statistics for the two datasets in Table 15. We observe that the average length of issue sections is much longer than source code comments; this indicates that issue sections contain more information than source code comments in general. Furthermore, while there are three times more source code comments than issue sections in the two datasets, the vocabulary size of issue sections is a little higher. These statistics confirm that the diversity of issues is higher than source code comments.

Second, **some types of SATD are different in source code comments and issue tracking systems.** In the following paragraphs, we highlight two specific types of SATD, namely *defect debt* and *requirement debt*, that have key differences in code comments and issues. In the case of defect debt, only the defects that are left unresolved, can be classified as defect debt (see Table 2). For source code, *task comments* denoting that bugs need to be fixed can be directly classified as *defect debt*, since they indicate that bugs are reported to be fixed at a later stage. An example is shown below:

> *"TODO: may not work on all OSes"* - [Defect debt from JMeter code comments]

However, in issue tracking systems, most bugs are reported and resolved immediately. Thus, in order to capture defect debt, we need to first identify defects and then judge whether fixing them is postponed or not. Only if fixing the defects is deferred or ignored, they can be tagged as *defect debt*. To exemplify this distinction, we show an example of an issue section that denotes defect debt and one that does not:

> *"I do not think it is a critical bug. Deferring it to 0.14."* - [Defect debt from Hadoop issues]

Table 15 Statistics for the source code comment dataset and issue tracking system dataset

Source	Avg. Length of Issue Sections / Code Comments	# of Sections/ comments	Vocabulary size
Source code comments	10.9	62275	31728
Issue tracking systems	35.4	23180	37202

> *"...thank you for reporting the bug and contributing a patch."* - [Non-defect debt from Hadoop issues]

Similarly, *requirement debt* also manifests differently in source code comments and issues. According to the definition of requirement debt (see Table 2), it reflects partially implemented functional or non-functional requirements. In source code comments, if the requirement is not completely satisfied, developers might leave a code comment, such as:

> *"TODO support multiple signers"* - [requirement debt from JMeter code comments]

The above comment can be simply annotated as *requirement debt*, as we know that the requirement is only partially implemented. However, in issue tracking systems, things are different. As mentioned above, *implementation proposal* is a type of issue section, that expresses new requirements. But such requirements may be partially or fully implemented in the meantime; that can be inferred through the ensuing discussion in the issue. To identify *requirement debt* in issues, it is necessary to differentiate partially implemented requirements from requirements that are fully implemented or not at all. The following examples show an issue section that denotes a requirement debt and one that does not:

> *"The backend (in master) has everything in place now to support this, but the frontend still needs to be adapted."* - [Requirement debt from Gerrit issues]
>
> *"It would be good to add a script to launch Hive using Ranger authorization."* - [Non-requirement debt from Impala issues]

Although identifying SATD in issue tracking systems is harder than in source code comments, it does not require a much larger amount of data to achieve decent accuracy when training machine learning models. Compared with previous work by Maldonado et al. (2017) that identifies SATD in source code comments, their incremental training curves are similar to ours (see Fig. 4). More specifically, the F1-score increases dramatically when the training dataset is small. After that, it goes up moderately. In order to achieve 90% of the accuracy of SATD identification, our model needs 16.3% (3400) issue sections on the issue SATD dataset, while about 23.0% (11800) source code comments are required on the source code comment SATD dataset. Therefore, relatively small datasets can achieve decent accuracy on both code comments and issue sections.

5.2 Similarity Between SATD Keywords Extracted from Source Code Comments and Issue Tracking Systems

In Tables 10 and 12, some keywords are underlined to highlight common SATD keywords in both issues and code comments. We observe that there are some common SATD keywords shared between source code comments and issues, such as 'ugly', 'bad', and 'get rid of'. In order to get a deeper understanding of the differences between SATD keywords from the two sources, we extract the top 10% (i.e., 272) unigram to trigram SATD keywords (across all projects) from source code comments and issue tracking systems. Subsequently, we compare the top 272 extracted SATD keywords from the two sources.

The top unique and common keywords are presented in Table 16, while the heatmap in Fig. 5 illustrates the number of common keywords between the two sources. In Fig. 5, the symbol attached before the project names (i.e., *C* or *I*) refers to the data source (i.e., *code comments* or *issues*). We observe that there are fewer keywords shared between issues and code comments compared to keywords shared between issues or between code comments.

Table 16 Top 10 n-gram SATD keywords from issue tracking systems and source code comments

Unique keyword (issue tracking systems)	Common keyword	Unique keyword (source code comments)
performance	why	todo
clean	improve	fixme
typo	leak	hack
remove	probably	should
flaky	perhaps	workaround
unused	better	defer argument checking
slow	instead	xxx
refactor	wrong	bug
warnings	missing	not needed
confusing	deprecated	implement

More specifically, we find on average that 73.4 keywords are shared between projects in issues, 48.4 keywords are shared between projects in code comments, and only 16.8 keywords are shared between projects across the two sources. The finding indicates that there is a certain similarity between SATD in issues and code comments, but the similarity is limited.

5.3 Implications for Researchers and Practitioners

Based on the findings, we suggest the following directions for researchers:

- Our work provides a deep learning approach to automatically identify SATD from issue tracking systems. The proposed approach can enable researchers to automatically identify SATD within issues and conduct studies on the measurement, prioritization, as well as repayment of SATD in issue tracking systems on a large scale.

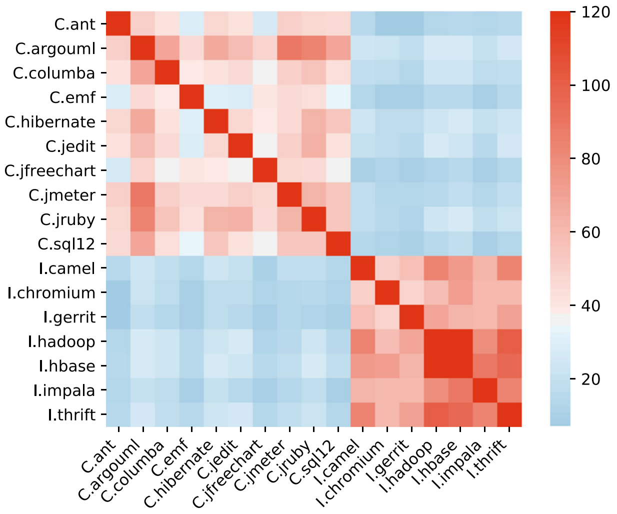

Fig. 5 Number of common keywords between different projects

- To enable further research in this area, we make our issue SATD dataset publicly available.[11] The dataset contains 23,180 issue sections, in which 3,277 issue sections are classified as SATD issue sections.
- We found that relatively small datasets can achieve decent accuracy in identifying SATD on both source code comments and issues. We thus recommend that researchers explore SATD in other sources (i.e., pull requests and commit messages) and contribute a moderate-sized dataset for automatic SATD identification in the corresponding sources.
- Our findings suggest that there is a certain similarity between SATD in issues and in source code comments, but the similarity is limited. We encourage researchers to study the differences between SATD in these and other sources, e.g. in pull requests or commit messages. This could advance the understanding of SATD in the different sources.
- Although our study experimented with the generalizability of our approach across projects and across issue tracking systems, the scope of our study is still limited. Thus, we recommend that researchers investigate the applicability of our approach to other projects (esp. industrial projects) and other issue tracking systems. If possible, we advise them to make their datasets publicly available to be used for training new SATD detectors.
- Because of the high diversity of issues and the different forms of SATD in issues, SATD identification within issues is harder than in source code comments. However, further research can potentially improve the F1-score obtained in our study (e.g., through using other machine learning techniques or trying richer datasets in the software engineering domain for transfer learning).

We also propose a number of implications for software practitioners:

- Our SATD identification approach can help software developers and especially project managers to evaluate the quality of their project. For instance, project managers can use this tool to track SATD in issue tracking systems along evolution. If the accumulated SATD reaches a threshold, then more effort may need to be spent in paying it back.
- We recommend that tool developers use our SATD identifier in their toolsets and dashboards and experiment with them in practice.
- We encourage practitioners to study carefully the SATD keywords listed in our results. This will help them to understand in practice the nature of SATD, how to better formulate it themselves and how to recognize SATD stated from others.
- Our findings can help practitioners better understand the differences between SATD in different sources, e.g., defect debt is identified differently in source code comments compared to issue tracking systems. This can also help practitioners better identify SATD in different sources.

6 Threats to Validity

6.1 Threats to Construct Validity

Construct validity reflects the correctness of operational measures for the studied subjects. We observed that only a small amount of issue sections are classified as SATD issue

[11] https://github.com/yikun-li/satd-issue-tracker-data

sections. To accurately measure the accuracy of machine learning models on SATD identification, we chose precision, recall, and F1-score as evaluation metrics. These metrics have been used in previous similar studies (Maldonado et al. 2017; Huang et al. 2018), and are well-established for this type of work.

6.2 Threats to External Validity

Threats to external validity concern the generalizability of our findings. Because we trained machine learning models on our issue SATD dataset, the data selected and analyzed might influence the generalizability of our findings. In order to partially mitigate this threat, we randomly selected seven open-source projects using two different issue tracking systems, being maintained by mature communities, and containing sufficient issue data. Besides, to ensure the collected issues are sufficiently representative of issues in each project, we calculated the size of the statistically significant sample based on the number of issues in each project. We then randomly selected issues according to the size of the statistically significant sample. In Section 4.6, we showed that the size of our dataset is sufficient for training machine learning models to identify SATD from issue tracking systems. When evaluating the predictive performance of machine learning models, we use stratified 10-fold cross-validation to mitigate the bias caused by random sampling. Moreover, in Section 4.5 we evaluated the generalizability of our approach across projects and across issue tracking systems.

However, because of the nature of open-source projects, developers tend to communicate online through tools, such as issue tracking systems and mailing lists; this facilitates new contributors to understand the details of issues and contribute to projects. In contrast, industrial projects have most developers working at the same premises; so, they tend to efficiently communicate the details of issues offline. This limits the generalizability of our results to such projects. In conclusion, our findings may be generalized to other open-source projects of similar size and complexity and of similar ecosystems.

6.3 Threats to Reliability

Reliability considers the bias that researchers may induce in data collection and analysis. Moreover, we manually classified issue sections as different types of SATD or not. To reduce this bias, the issue data was first analyzed manually by four independent researchers. Then the first author analyzed issue samples and calculated Cohen's kappa coefficient between his output and that of the independent researchers. If the agreement was good (i.e., Cohen's kappa coefficient is above 0.7), the classification is considered complete. If not, the first author discussed the classification differences between them to reach a consensus. Subsequently, they improved the classification, and Cohen's kappa coefficient was calculated again to ensure the level of agreement was good.

Furthermore, our results depend on the data analysis methods we use. In terms of machine learning approaches, we have chosen some of the most commonly approaches used by researchers and practitioners. Besides, we follow established guidelines for tuning hyper-parameters (Zhang and Wallace 2017) and exploring transfer learning (Semwal et al. 2018). Finally, and most importantly, we make our issue SATD dataset publicly available[12] in the replication package.

[12] https://github.com/yikun-li/satd-issue-tracker-data

7 Conclusion

In this work, we investigated SATD identification with respect to *accuracy*, *explainability*, and *generalizability* in issue tracking systems. We contributed a dataset including 23,180 issue sections classified as SATD sections or non-SATD sections from seven open-source projects using two issue tracking systems. Moreover, we compared different machine learning algorithms and propose a CNN-based approach to identify SATD in issues with an F1-score of 0.686. Furthermore, we explored the effectiveness of transfer learning using other datasets to improve the F1-score of SATD identification from 0.686 to 0.691. In addition, we identified a list of n-gram top SATD keywords, which are intuitive and can potentially indicate types and indicators of SATD. Besides, we observed that projects using different issue tracking systems have less common SATD keywords compared to projects using the same issue tracking system. We also evaluated the generalizability of our approach. The results show our approach achieves the average F1-score of 0.652, ranging between 0.561 to 0.709, using leave-one-out cross-project validation; when applying leave-one-out cross-issue-tracker validation, the average F1-score is dropped by 6.4% and 16.3% for models training on Jira and Google issue trackers compared to using leave-one-out cross-project validation. Finally, we investigated the amount of data needed for our approach. We showed that only a small amount of training data is needed to achieve good accuracy.

In the future, we tend to explore the differences between SATD in different sources. We also aim to use the ensemble learning technique to improve the predictive performance of machine learning models by combining different classifiers. Moreover, we plan to further analyze our dataset to identify different types (or indicators) of SATD in issue tracking systems.

Acknowledgments This work was supported by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdomproject.github.io/website).

Author Contributions –**Yikun Li** Conceptualization, Data curation, Formal Analysis, Investigation, Methodology, Software, Validation, Visualization, Writing—original draft

–**Mohamed Soliman** Conceptualization, Methodology, Project administration, Supervision, Writing—review & editing

–**Paris Avgeriou** Conceptualization, Funding acquisition, Methodology, Project administration, Supervision, Writing—review & editing

Funding This study was funded by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website).

Availability of Data and Material The dataset generated during the current study is available in the the repository, which is available at the link.[13]

Code Availability Regular expression rules for data cleaning is provided as part of the replication package. It is available at the link.[14]

Declarations

Ethics approval and consent to participate Not applicable

[13] https://drive.google.com/drive/folders/1tVWa_KVlNdPgNhAmz2JNP-cYc8RImmSH?usp=sharing

[14] https://drive.google.com/drive/folders/1tVWa_KVlNdPgNhAmz2JNP-cYc8RImmSH?usp=sharing

Consent for Publication Not applicable

Conflicts of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

Alves NS, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: 2014 Sixth international workshop on managing technical debt. IEEE, pp 1–7

Alves NS, Mendes TS, de Mendonça MG, RO Spínola, Shull F, Seaman C (2016) Identification and management of technical debt: a systematic mapping study. Inf Softw Technol 70:100–121

Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing technical debt in software engineering (Dagstuhl Seminar 16162). Dagstuhl Rep 6(4):110–138. https://doi.org/10.4230/DagRep.6.4.110

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Proceedings of the 13th international conference on mining software repositories, pp 315–326

Bellomo S, Nord RL, Ozkaya I, Popeck M (2016) Got technical debt? Surfacing elusive technical debt in issue trackers. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). IEEE, pp 327–338

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Trans Assoc Comput Linguist 5:135–146

Calefato F, Lanubile F, Maiorano F, Novielli N (2018) Sentiment polarity detection for software development. Empir Softw Eng 23(3):1352–1382

Dai K, Kruchten P (2017) Detecting technical debt through issue trackers. In: QuASoq@ APSEC, pp 59–65

de Freitas Farias MA, Santos JA, Kalinowski M, Mendonça M, Spínola RO (2016) Investigating the identification of technical debt through code comment analysis. In: International conference on enterprise information systems. Springer, pp 284–309

Efstathiou V, Chatzilenas C, Spinellis D (2018) Word embeddings for the software engineering domain. In: Proceedings of the 15th international conference on mining software repositories, pp 38–41

Ernst NA (2012) On the role of requirements in understanding and managing technical debt. In: 2012 Third international workshop on managing technical debt (MTD). IEEE, pp 61–64

Fernández A, García S, Galar M, Prati RC, Krawczyk B, Herrera F (2018) Learning from imbalanced data sets. Springer, Berlin

Fleiss JL, Levin B, Paik MC et al (1981) The measurement of interrater agreement. Statistical Methods for Rates and Proportions 2(212–236):22–23

Flisar J, Podgorelec V (2019) Identification of self-admitted technical debt using enhanced feature selection based on word embedding. IEEE Access 7:106475–106494

Genkin A, Lewis DD, Madigan D (2007) Large-scale bayesian logistic regression for text categorization. Technometrics 49(3):291–304

Gu Z, Gu L, Eils R, Schlesner M, Brors B (2014) Circlize implements and enhances circular visualization in r. Bioinformatics 30(19):2811–2812

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

Joulin A, Grave É, Bojanowski P, Mikolov T (2017) Bag of tricks for efficient text classification. In: Proceedings of the 15th conference of the European chapter of the association for computational linguistics: volume 2. Short Papers, pp 427–431

Kamei Y, Maldonado EDS, Shihab E, Ubayashi N (2016) Using analytics to quantify interest of self-admitted technical debt. In: QuASoq/TDA@ APSEC, pp 68–71

Kim Y (2014) Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on empirical methods in natural language processing (EMNLP), pp 1746–1751

Kowsari K, Jafari Meimandi K, Heidarysafa M, Mendu S, Barnes L, Brown D (2019) Text classification algorithms: a survey. Information 10(4):150

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

Li Y, Soliman M, Avgeriou P (2020) Identification and remediation of self-admitted technical debt in issue trackers. In: 2020 46th Euromicro conference on software engineering and advanced applications (SEAA), pp 495–503. https://doi.org/10.1109/SEAA51224.2020.00083

Liu Z, Huang Q, Xia X, Shihab E, Lo D, Li S (2018) Satd detector: a text-mining-based self-admitted technical debt detection tool. In: Proceedings of the 40th international conference on software engineering: companion proceedings, pp 9–12

Maldonado EDS, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 2015 IEEE 7th international workshop on managing technical debt (MTD), pp 9–15. https://doi.org/10.1109/MTD.2015.7332619

Maldonado EDS, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans Softw Eng 43(11):1044–1062. https://doi.org/10.1109/TSE.2017.2654244

McCallum A, Nigam K et al (1998) A comparison of event models for naive bayes text classification. In: AAAI-98 workshop on learning for text categorization. Citeseer, vol 752, pp 41–48

Merten T, Mager B, Hübner P, Quirchmayr T, Paech B, Bürsner S (2015) Requirements communication in issue tracking systems in four open-source projects. In: REFSQ workshops, pp 114–125

Mikolov T, Grave E, Bojanowski P, Puhrsch C, Joulin A (2018) Advances in pre-training distributed word representations. In: Proceedings of the international conference on language resources and evaluation (LREC 2018)

Ortu M, Murgia A, Destefanis G, Tourani P, Tonelli R, Marchesi M, Adams B (2016) The emotional side of software developers in jira. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). IEEE, pp 480–483

Perkins DN, Salomon G et al (1992) Transfer of learning. International Encyclopedia of Education 2:6452–6457

Phan H, Krawczyk-Becker M, Gerkmann T, Mertins A (2017) Dnn and cnn with weighted and multi-task loss functions for audio event detection. arXiv:170803211

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 2014 IEEE international conference on software maintenance and evolution. IEEE, pp 91–100

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

Runeson P, Host M, Rainer A, Regnell B (2012) Case study research in software engineering: guidelines and examples. Wiley, New York

Semwal T, Yenigalla P, Mathur G, Nair SB (2018) A practitioners' guide to transfer learning for text classification using convolutional neural networks. In: Proceedings of the 2018 SIAM international conference on data mining. SIAM, pp 513–521

Shalev-Shwartz S, Ben-David S (2014) Understanding machine learning: from theory to algorithms. Cambridge University Press, Cambridge

Sierra G, Shihab E, Kamei Y (2019) A survey of self-admitted technical debt. J Syst Softw 152:70–82. https://doi.org/10.1016/j.jss.2019.02.056. http://www.sciencedirect.com/science/article/pii/S0164121219300457

Smith T (2018) The most important players in the open source ecosystem. https://dzone.com/articles/the-most-important-players-in-the-open-source-ecos

Steidl D, Hummel B, Juergens E (2013) Quality analysis of source code comments. In: 2013 21St international conference on program comprehension (icpc). IEEE, pp 83–92

Sun A, Lim EP, Liu Y (2009) On strategies for imbalanced text classification using svm: a comparative study. Decis Support Syst 48(1):191–201

Tan S (2006) An effective refinement strategy for knn text classifier. Expert Syst Appl 30(2):290–298

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Trans Softw Eng 43(11):1063–1088

van Solingen R, Basili V, Caldiera G, Rombach HD (2002) Goal Question Metric (GQM) approach. In: Encyclopedia of software engineering. Wiley, Hoboken, pp 528–532

Wehaibi S, Shihab E, Guerrouj L (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, pp 179–188

Wei J, Zou K (2019) Eda: easy data augmentation techniques for boosting performance on text classification tasks. In: Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP), pp 6383–6389

Wieting J, Bansal M, Gimpel K, Livescu K (2015) Towards universal paraphrastic sentence embeddings. arXiv:151108198

Xavier L, Ferreira F, Brito R, Valente MT (2020) Beyond the code: mining self-admitted technical debt in issue tracker systems. arXiv:200309418

Xu B, Guo X, Ye Y, Cheng J (2012) An improved random forest classifier for text categorization. JCP 7(12):2913–2920

Yao L, Mao C, Luo Y (2019) Graph convolutional networks for text classification. In: Proceedings of the AAAI conference on artificial intelligence, vol 33, pp 7370–7377

Zampetti F, Serebrenik A, Di Penta M (2018) Was self-admitted technical debt removal a real removal? An in-depth perspective. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR). IEEE, pp 526–536

Zhang Y, Wallace BC (2017) A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. In: Proceedings of the eighth international joint conference on natural language processing (volume 1: long papers), pp 253–263

Zhang X, Zhao J, LeCun Y (2015) Character-level convolutional networks for text classification. In: Advances in neural information processing systems, pp 649–657

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.