2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 979-8-3503-3066-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SANER60148.2024.00087

Self-Admitted Technical Debts Identification: How Far Are We?

Hao Gu*, Shichao Zhang*, Qiao Huang†, Zhifang Liao*, Jiakun Liu‡, David Lo‡
*Central South University, Changsha, China
†Zhejiang Gongshang University, Hangzhou, China
‡Singapore Management University, Singapore, Singapore
{harry.gu, zfliao}@csu.edu.cn, 3043644408@qq.com,
qiaohuang@zjgsu.edu.cn, {jkliu, davidlo}@smu.edu.sg

***Abstract*—Self-admitted technical debt (SATD) is a kind of technical debt that is already acknowledged by the developers and needs additional work or resources to address in the future. In recent years, though many methods have been proposed to detect SATDs, these methods have mainly focused on Java-type code comments published by Maldonado et al. It is unclear whether these methods trained on Maldonado's code comments dataset can find SATD in other programming languages or other software artifacts, such as issue trackers, pull requests, and commit messages effectively. In order to answer the above confusion and investigate how far our community has progressed in the field of SATD identification, we first collect a comprehensive dataset that contains SATDs in code comments from four different programming languages (java, python, docker file, XML) and SATDs in other different artifacts (issue tracker, pull requests, commit messages) from previous papers working in the field of SATD. Then, we re-train the existing models with Maldonado's code comments dataset and test all the models on other programming languages and other artifacts. The results show that existing SATD identification methods can find SATDs in other non-Java languages, but perform poorly in identifying SATDs from three other different artifacts. In addition, in order to simultaneously identify four different artifacts of SATDs, we develop a Multi-Task Learning model utilizing BERT for SATD identification (MT-BERT-SATD). Considering four different artifacts and the SATD identification tasks, MT-BERT-SATD achieves an average F1-score of 0.712 (0.625-0.859), which is superior to existing models from 4.6% to 30.4%. Results show that MT-BERT-SATD can effectively identify SATD instances across explored programming languages and software artifacts, indicating its capability to identify SATD instances in new and unexplored programming languages and software artifacts.**

Index Terms*—multi-task learning, Self-Admitted Technical Debt, *MT-BERT-SATD

I. Introduction

Technical debt (TD) refers to various technical compromises made during software development, which result in additional work or costs in the future, thereby reducing the productivity and code quality of the development team [1]–[5]. TD admitted by developers is referred to as Self-Admitted Technical Debt (SATD). SATD poses significant challenges to software maintenance, as the introduction of these debts requires developers to spend more time and resources in repaying them in the later stages [2], [6]–[8]. Wehaibi et al. [3] found that the existence of SATD not only increases the probability of software defects but also hinders future changes to software systems. Therefore, identifying and resolving SATD in a timely manner is critical to ensuring high software quality and maintainability.

In the past, our community has proposed many approaches [6], [9]–[13] to automatically detect SATD in Java code comments [9]. However, prior studies [14]–[19] showed that the SATD can commonly exist across programming languages and software artifacts. SATD across different artifacts can be closely related (e.g., developers discuss SATD present in the source code through issue trackers). If we cannot identify the SATD in other artifacts, we cannot manage SATD as a whole. When developers remove the SATD comments in the source code but neglect the corresponding discussion of the SATD in other software artifacts, e.g., issues, there can be obsolete SATD in other artifacts. This imposes additional effort in the management of SATD. What's more, the SATD in different programming languages and different artifacts have different characteristics. For example, prior studies show the presence of a set of priori knowledge within SATD comments, such as the use of markers like "TODO" or "FIXME" [11]. However, SATD from other artifacts may lack similar priori knowledge (e.g., SATD in commit message may not marked with "FIXME"), and existing SATD detection tools focus on the SATD comments in the source code and overlook the SATD in other artifacts. This indicates that the existing approaches trained on Java projects dataset may fail. This requires researchers to identify these SATD comments from different artifacts manually when investigating new software artifacts.

Motivated by the above-mentioned examples, we would like to investigate the progress made by our community in the field of SATD identification and explore effective ways of simultaneously identifying SATD from various artifacts. We collect and make publicly available a comprehensive dataset of SATD from the datasets with manually labeled data proposed in previous SATD-related studies [9], [11], [14]–[21]. The previous datasets were focused on individual languages or artifacts, without simultaneous consideration consideration of other languages or artifacts. The comprehensive dataset

Hao Gu and Shichao Zhang are both first authors and contributed equally to this work. Zhifang Liao and Jiakun Liu are both corresponding authors.

2640-7574/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER60148.2024.00087

contains over 9K positive SATD samples from four different languages in code comments, namely Java, Python, Dockerfile, and XML. Furthermore, the dataset also contains over 6K positive SATD samples from three other kinds of artifacts, namely, issue reports, pull requests, and commit messages. Our study aims to answer the following research questions:

RQ1: Can existing SATD detection models identify SATDs across programming languages effectively?

We re-train the existing model using the corrected Maldonado dataset [22] and test these trained models and unsupervised approaches on non-Java code comments to verify their generality across programming languages. Our experiment demonstrates that existing SATD identification methods can effectively recognize SATD across programming languages (F1-scores range from 0.870 to 0.925 except pattern-based approach by relying entirely on keywords for matching).

RQ2: Can existing SATD detection models identify SATDs across software artifacts effectively?

Followed by RQ1, we evaluate these well-trained models and unsupervised learning methods on non-code comment datasets to investigate their efficacy in identifying SATD from artifacts beyond code comments, i.e. issue trackers, pull requests, and commit messages. Our results suggest that the existing SATD identification approaches exhibit poor performance in identifying SATD from other artifacts (F1-scores range from 0.023 to 0.324), highlighting the need for further research to improve SATD identification.

RQ3: Can we build a model to accurately identify SATDs across software artifacts?

Due to the varying characteristics of SATD across different artifacts [18], existing SATD identification methods have limited capacity to learn distinctive features of SATD from different artifacts using a single model architecture. In more detail, existing unsupervised learning methods, such as Pattern [6] and MAT [11], only capture features from code comments, neglecting SATD features from non-code comments. Existing supervised learning models, such as NLP [9], TM [10], and BERT [13], rely on a single input source and cannot distinguish between the different artifacts of SATD features.

To this end, we propose a multi-task learning model based on BERT, namely *MT-BERT-SATD*. *MT-BERT-SATD* employs multitask learning technique to concurrently learn the SATD features targeted at different artifacts, thereby enhancing the SATD identification performance across different artifacts. For identifying SATDs across different software artifacts (such as issue, PR, commit, and code comments), *MT-BERT-SATD* can learn the different features from them and share parameters when training models to achieve a better generalization than other single-task learning methods. Our findings demonstrate that our approach performs better than existing SATD identification methods in identifying SATDs from four distinct artifacts simultaneously. Additionally, our approach is more effective in dealing with small sample data and identifying SATD in new software artifacts than other models, demonstrating its good generalization ability.

The main contributions of this work are as follows:

- We collected and summarized the literature on SATD, especially covering studies that (1) characterize SATD in different software artifacts and (2) automate SATD detection.
- We empirically demonstrate the generalizability of existing SATD identification models across programming languages and artifacts.
- We design and develop a model, *MT-BERT-SATD*, which can effectively identify SATD from different artifacts with a good generalization ability.

Table I
SATD Samples from four different artifacts.

Issue Trackers	"This method is not specific to TaskTracker, i.e., it should work fine with LocalRunner too, right? So there ought to be a better place to put it." *From [hadoop-issue-1251]*
Pull Requests	"Is this the implementation class? If so can we use 'class' instead of type to be more explicit?" *From [incubator-heron-pull-1820]*
Commit Messages	"Removing unused dependency and changing default daemon port to 9090. Fixes #300" *From [attic-apex-malhar-commit-01b162]*
Code Comments	"// TODO: This method doesn't appear to be used." *From [jmeter-code-comment]*

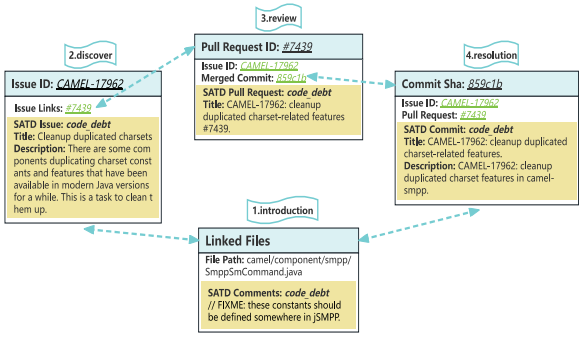

Fig. 1. An example of the four existence cycles of SATD.

II. SATD ACROSS PROGRAMMING LANGUAGES AND SOFTWARE ARTIFACTS

To understand the progress in the field of (1) empirical studies characterizing SATD in different software artifacts and (2) automated SATD detection methods, we conducted 2 literature reviews on the papers which contain the keyword "self-admitted technical debt" and are published from 2014 when the concept of SATD was first introduced until 2022. Besides, the 60 analyzed papers are mainly accepted by international first-class conferences (ESEC/FSE, ICSE, ASE, etc.) and first-class journals (TSE, TOSEM, etc.) in the direction of software engineering-system, software-programming language, as well as other important conferences and journals such as EMSE, MSR, ICSME, etc.

A. Related Work and Motivating Examples

Prior studies showed that both OSS developers and industrial developers can admit SATD in source code comments (in Java [6], build scripts [14], [16], and python [8]), commit messages [23], [24], code reviews or pull requests [19], and issue trackers [19], [25], [26]. Table I shows examples of

SATDs from these four different artifacts. When developers find or repay SATDs in some projects, they may record the finding or repayment information of SATDs in commit messages [18], [27], [28]. Also, developers may report urgent SATDs in issue trackers [29]–[31]. Apart from that, SATDs from issue trackers and code comments may be discussed in reviewing related PRs [19]. That is, SATD exists in all four artifacts, and its discovery and repayment process may continue throughout the entire cycle of SATD's existence.

Fig. 1 shows an example of a SATD instance across different software artifacts in Camel[1]. An OSS developer introduced a "FIXME" code debt in the code comments. Afterward, this SATD was discovered and recorded in the issue, and then the developer submitted a pull request to solve this SATD, which was reviewed by the manager. Finally, the repair information was recorded in the commit. This motivates us to move towards centrally managing SATD instances in the project as a whole. For example, when developers find SATD in one artifact, they can find more details of SATD in other artifacts so they can have a clear understanding of the entire SATD existence process and thus repay SATD with higher quality. In contrast, if only one artifact of SATD is singularly identified, such as source code comments, then the developer only understands part of the SATD introduction phase, but neglects the existing corresponding discussion (i.e., ignores the collective intelligence contributed by other developers). Moreover, without identifying SATDs across software artifacts, when developers repay the SATD in source code but without updating the status of the corresponding SATD issue discussion, there can be obsolete SATD instances in issue discussions. Future developers may fix the already repaired SATD instances, which wastes the effort of other developers.

B. Dataset collection

To investigate whether the existing models can detect SATD instances across programming languages and software artifacts, we collect a comprehensive dataset from previous papers studying SATDs. To ensure the reliability and authenticity of the dataset, each sub-dataset of this dataset has the following characteristics: (1) The dataset is publicly available. (2) The dataset was manually classified by the authors. (3) The manual classification process of this dataset is rigorous and reliable by following the majority rule or unifying decisions through meetings. Among the papers that are selected in our literature review, 10 papers meet the above criteria. We obtain their publicly available datasets through the links provided by the authors. Due to the code comments dataset published by Maldonado et al. (i.e., Dataset-06-Comments-Java in Table II) was corrected by Yu et al. [22] and this corrected dataset is widely used [22], [32], [33]. In this paper, Dataset-06-Comments-Java is the corrected dataset. Besides, the comprehensive dataset we proposed consists of SATDs and non-SATDs from four different artifacts, namely, code comments, issue trackers, pull requests, and commit messages. Among them, the code comment part of the dataset also contains SATDs and non-SATDs extracted from four different languages, namely Dockerfile, Python, XML, and Java. Table II shows the details of our dataset.

[1]https://github.com/apache/camel

III. Automatic Identification of SATD

To understand the progress in the field of automatic SATD detection, similar to Section II, we collected 15 articles about the automatic identification of SATD.

A. Related Work and Selected Models

Numerous studies in recent years have been devoted to the automatic identification of SATD, with a predominant focus on SATD identification within code comments. The three main techniques utilized for the identification of SATD are unsupervised learning methods [6], [11], supervised learning methods [9], [10], [13], [16], [34], and semi-supervised learning methods [22], [32]. We investigate 15 articles related to SATD detection methods and select 5 representative models with available source code [11], [13]. The five models are as follows: **Pattern** [6], **NLP (Natural Language Processing)** [9], **TM (Text Mining)** [10], **MAT (Matches task Annotation Tags)** [11], and **BERT (Bidirectional Encoder Representations from Transformers)** [13]. Where Pattern and MAT are unsupervised methods, while NLP, TM, and BERT are supervised methods. The following is a brief introduction to each method:

- **Pattern** [6] is an unsupervised SATD identification method based on pattern matching, which includes 62 frequent keywords (e.g., "yuck") or phrases (e.g., "cause for issue") commonly found in SATD comments.
- **NLP** [9] is an automated approach to identify SATD using natural language processing, namely, maximum entropy classifier [35]. During training, the model aims to maximize the conditional likelihood of the classes by taking into account feature dependencies and calculating the corresponding feature weights.
- **TM** [10] used text-mining techniques to identify SATDs in code comments. Selecting useful features by Information Gain (IG) technique [36]. IG is used to evaluate the amount of information needed to predict whether a comment contains SATD, based on the presence or absence of a particular feature. Finally, using a sub-classifier voting technique to identify SATDs in the target project.
- **MAT** [11] is a simple heuristic method, which Matches task Annotation Tags (MAT), to automatically identify SATD. Including four SATD tags: "TODO", "FIXME", "HACK", and "XXX".
- **BERT** [13] is a modern machine-learning technique to identify SATDs in code comments using BERT. It contains a pre-train and fine-tune process. Especially, a pre-trained model, namely, BERT-SO-1M, achieved the best performance. In the subsequent comparative study, we used BERT-SO-1M as the original paper's representative pre-trained model. BERT's strong performance is attributed to its pre-training as a bidirectional language model, which allows it to learn a vast

Table II
Comprehensive dataset containing four different programming languages and four different artifacts

Dataset	Sample Source	Description	Published Year	Article Source	#Samples	#SATD	Reference
Dataset-06-Comments-Java	Code Comments/java	Code comments in 10 different types of Java projects.	2017	TSE	62275	4497	[9]
Dataset-03-Comments-XML	Code Comments/XML	Code comments in maven build systems.	2021	TSE	884	513	[16]
Dataset-05-Comments-Java	Code Comments/java	Code comments in 10 different types of Java projects.	2021	TOSEM	81260	2995	[11]
Dataset-01-Comments-Dockerfile	Code Comments/dockerfile	Code comments from files in Docker.	2022	EMSE	382	50	[14]
Dataset-02-Comments-Python	Code Comments/python	Code comments in machine learning projects.	2022	FSE	856	789	[15]
Dataset-04-Comments-Java	Code Comments/java	Code comments in SQL and NO-SQL java systems.	2022	EMSE	361	256	[21]
Dataset-07-Issue	Issue Trackers	Issues in Jira and Google issue trackers.	2022	EMSE	23180	3277	[17]
Dataset-08-Issue	Issue Trackers	R package issues from rOpenSci and Bioconductor.	2022	SANER	1205	805	[20]
Dataset-10-PR	Pull Requests	Pull Requests from Spark, Kafka, and React.	2022	ESEM	2122	811	[19]
Dataset-09-PR	Pull Requests	Pull Requests from Apache echo-system.	2023	EMSE	5000	718	[18]
Dataset-11-Commits	Commit Messages	Commit Messages from Apache echo-system.	2023	EMSE	5000	747	[18]
Total	-	-	-	-	182525	15458	-

amount of contextual information and excel at various natural language processing tasks.

As of our submission, we found another article [18] published in EMSE that shares a strikingly similar research topic to ours. We believe our work is contemporaneous. However, we cannot directly replicate the work of Li et al. [18] due to two reasons: (1) To replicate the best performance as described in the literature for certain existing models with numerous parameters, we need to obtain all the corresponding parameter settings. Unfortunately, the provided replication package[2] does not include the source code required for training their model. (2) We make a fair comparison between our approach and Li et al.'s work, we need to train on the same training set and test on the same test set. However, the provided replication package[2] did not specify the specific training and testing set they used, we cannot directly compare the performance of the models based on their results.

However, we believe that our work is more complete than Li et al [18]. There are two reasons: (1) Our model achieves 80% accuracy in identifying SATD in new sources using a voting technique as mentioned in Section VIII, unlike Li et al. [18] who focused only on the four known sources. (2) Our model's datasets include diverse SATD from programming languages' code comments, obtained through extensive research, while Li et al. [18] solely relied on Maldonado et al.'s [9] code comment dataset for Java. This implies that their model failed to capture the information about SATD in code comments of different programming languages.

IV. EXPERIMENTAL SETUP

A. Research Questions

Previous approaches for identifying SATD have predominantly focused on comments in Java code released by Maldonado et al. However, SATD exists in code comments not only for Java types [14]–[16]. The non-Java programming languages have syntax rules and conventions that differ from Java. Developers require a tool to identify SATDs in code comments when developing programs in these languages. Similarly, researchers conducting empirical studies on SATDs also require a tool that can help them automatically identify SATDs in code comments. This can significantly alleviate the burden on developers and researchers in manually identifying SATDs. However, it remains unclear whether these trained models and unsupervised models can effectively detect SATD in other programming languages. To explore the generalizability of these trained models and unsupervised models in identifying SATDs in code comments of non-java types, we first need to investigate: **RQ1: Can existing SATD detection models identify SATDs across programming languages effectively?** Additionally, SATDs are not only present in source code comments [17]–[19]. For developers, identifying SATD from different artifacts can help them keep track of the latest status of SATD repayment and ensure that certain SATDs in projects are not overlooked. For researchers, an automated tool that can identify SATD from different artifacts is needed when conducting empirical studies on SATD from different artifacts. Therefore, to explore the generalizability of these trained models and unsupervised models in identifying SATDs in three other artifacts, i.e. issue trackers, pull requests, and commit messages. We need to investigate next: **RQ2: Can existing SATD detection models identify SATDs across software artifacts effectively?** Our RQ1 and RQ2 results show that previously trained models and unsupervised models can identify SATDs well in different language projects, but perform poorly in identifying SATDs from three other different artifacts. In order to simultaneously identify SATDs from four different artifacts, We want to know: **RQ3: Can we build a model to accurately identify SATDs across software artifacts?**

B. Datasets

Preprocessing: Since different developers may have different writing habits for writing SATDs, for example, they may write "HACK" as "Hack", "hack", etc., we need to apply a unified preprocessing step to the samples in the comprehensive dataset. To this end, we refer to the preprocessing procedures of Huang et al. [10] and Maldonado et al. [9]: (1) **Remove all non-alphabetic characters from the samples**, such as "//" and "**". However, we preserve exclamation and question marks during preprocessing, as they are deemed helpful for identifying SATDs by Maldonado et al. (2) **Convert all English words into lowercase**, for example, convert "HACK" and "Hack" into "hack". (3) **Remove stop words**: We use the stop word list provided by Huang et al. to remove stop words from the samples. Note that, words with a length of no more than 2 or no less than 20 are also considered stop words.

[2]https://github.com/yikun-li/satd-different-sources-data

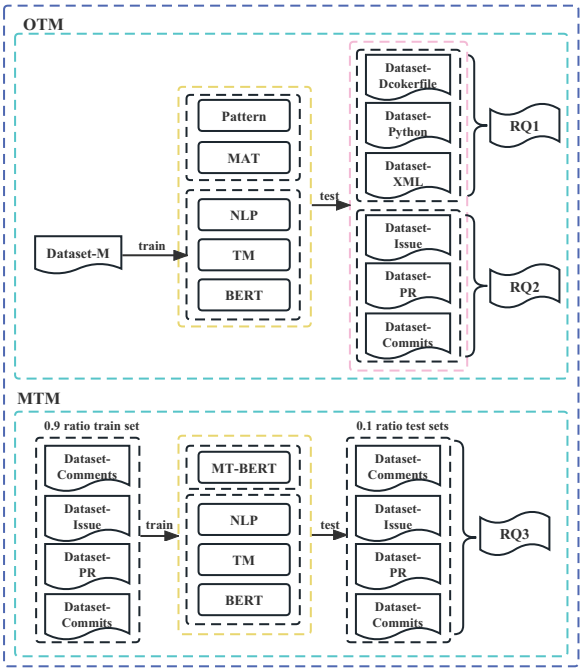

Fig. 2. OTM and MTM scenarios.

C. Experiment Scenarios

In this study, we design two prediction scenarios to investigate the effectiveness of SATD identification: One-To-Many (OTM) scenario, which is used to carry out RQ1 and RQ2 experiments, and the Many-To-Many (MTM) scenario, which is used to conduct RQ3 experiments. The experiments for OTM in RQ1 can effectively evaluate the generalization of existing models in predicting SATDs in code comments of other non-Java programming languages. The experiments for OTM in RQ2 can effectively evaluate the generalization of existing models in predicting SATDs in other non-source code comment artifacts. MTM in RQ3 evaluates the performance of our approach and existing methods in identifying SATD from four different artifacts simultaneously. Fig. 2 illustrates the two experimental scenarios.

• **OTM scenario**: In the OTM scenario, **For RQ1**, we use the corrected Maldonado dataset as the train set, namely **Dataset-M**. Samples from three non-Java language projects' code comments (i.e., **Dataset-Dockerfile**, **Dataset-Python**, **Dataset-XML**) are used as the test sets. The aim is to validate the effectiveness of existing models trained on the Maldonado dataset and unsupervised learning methods in identifying SATD in other programming languages. **For RQ2**, we also train the existing model with Dataset-M and use data from the other three artifacts (i.e., issue trackers, pull requests, and commit messages) as the test sets to verify whether the existing model can effectively identify SATD from different artifacts. Note that, because we aim to use as many test sets as possible to evaluate the existing models, we merge the non-code comments datasets in Table II based on their artifacts to validate the generalization of existing models in identifying SATDs across software artifacts. For example, Dataset-07-Issue and Dataset-08-Issue are merged into a new sub-dataset named **Dataset-Issue**, and likewise, the other non-code comments datasets are merged into **Dataset-PR** and **Dataset-Commits**, respectively.

• **MTM scenario**: Due to our aim is identifying SATDs from four artifacts simultaneously, to guarantee the input features as many as possible and the data balance across software artifacts in model training, we merge the six sub-datasets in code comments and name it **Dataset-Comments**. At this point, we have four datasets from different software artifacts, namely Dataset-Issue, Dataset-PR, Dataset-Commits, and Dataset-Comments. Then, we divide the four different artifact datasets into four train sets and four test sets with a 9:1 ratio. Here, following prior studies [17]–[19], we adopt a stratified sampling method [37] to divide the dataset into train and test sets, ensuring that the proportion of SATD and non-SATD samples is the same in both sets. The four train sets are combined to train our approach and existing supervised models, and the four test sets are used to evaluate their performance.

D. Evaluation Metrics

In this work, we use Precision, Recall, and F1-score (one type of harmonic mean between Precision and Recall) as evaluation metrics, which are widely used in the evaluation of SATD recognition models [9]–[12], [32], [34]. Here are the calculation formulas for the three evaluation metrics:

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

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

In the above formulas, TP (True Positive) represents the number of SATDs predicted by the model as SATDs, FP (False Positive) represents the number of non-SATDs predicted by the model as SATDs and FN (False Negative) represents the number of SATDs predicted by the model as non-SATDs.

V. RQ1: Can existing SATD detection models identify SATDs across programming languages effectively?

A. RQ1 OTM Experiment Description

To investigate RQ1, we first perform a re-evaluation of previous SATD detection models using the corrected Maldonado dataset [22]. More specifically, for supervised models (i.e., NLP, TM, BERT), we follow the original authors' work by retraining these supervised models in a cross-project leave-one-out method (9 projects for training, 1 project for testing) [9]. To ensure the fairness of the comparison results, for unsupervised models (i.e., Pattern, MAT), we also select 1 project at a time for testing.

For the purpose of investigating the models' ability to predict completely new data, we perform deduplication between the train set and three test sets, ensuring that each test set does not contain any data from the train set. Then, following the evaluation approach in previous studies, we average the

results of projects in Dataset-M to establish the baseline metric of these models in identifying SATDs in non-Java projects. Finally, we perform the RQ1 experiment in the OTM scenario.

B. Experiment Result Analysis

In this experiment, we record the F1-score of previous SATD identification models in identifying SATDs across programming languages. In addition, to evaluate the performance of the existing models in identifying SATD comments from non-Java projects more intuitively, we calculate the average F1-scores of Dataset-Dockerfile, Dataset-Python, and Dataset-XML and compare them with F1-score of Dataset-M. We calculate the change value (expressed in italics). Table III shows our experimental results.

Table III
F1-score of existing approaches on SATDs from different language projects.

Dataset	Approach				
	Pattern	MAT	NLP	TM	BERT
Dataset-M (baseline metric)	**0.247**	0.747	0.761	0.710	0.834
Dataset-Dockerfile	0.273	0.869	0.848	0.759	0.880
Dataset-Python	0.156	0.957	0.954	0.956	0.965
Dataset-XML	0.248	0.802	0.883	0.895	0.931
Average	0.226	**0.876**	**0.895**	**0.870**	**0.925**
Average-Change	*(-8.5%)*	*(+17.3%)*	*(+17.6%)*	*(+22.5%)*	*(+10.9%)*

Experimental results show that **four out of five models show improved F1-score in identifying SATDs in the other three non-Java datasets**, with the TM method achieving the highest improvement from 71.0% to 87.0%, representing an overall improvement of 22.5%. However, the Pattern method shows a decrease in the average F1-score from 24.7% to 22.6%, indicating an overall decline of 8.5%, this may be due to the limitations of pattern recognition methods, as Potdar et al. [6] only investigated SATDs in four projects (i.e., Eclipse, Chromium OS, ArgoUML, and Apache httpd) and manually identified 62 keyword patterns based on these projects. Therefore, Pattern can achieve good precision but the overall F1-score is low [10]. In contrast, the other four methods can effectively identify SATDs in code comments of different language projects. In fact, this is easy to understand, according to the study by Guo et al. [11], developers tend to introduce some prior knowledge when recording certain types of SATDs information, such as using keywords like “todo”, “fixme”, “hack”, “xxx”. This prior knowledge is also likely to exist in non-Java projects, which implies that the model trained on the Dataset-M can achieve promising performance in identifying SATDs in other non-Java projects as well. Existing models can capture the differences between different programming languages by training from the Java dataset. This shows the common characteristics between different programming languages and the generability of the model.

> Apart from Pattern, the other four methods achieve the average F1-score from 0.870 to 0.925 which is higher than the F1-score on Dataset-M. In summary, previous SATD identification models can identify SATDs in code comments from different languages effectively.

VI. RQ2: Can existing SATD detection models identify SATDs across software artifacts effectively?

A. RQ2 OTM Experiment Description

Following RQ1, to ensure consistency with the performance of existing models trained on Dataset-M, we compute the average of the results from projects in Dataset-M to establish the baseline metric of these models in identifying SATDs from different software artifacts. Then, we remove duplicates from these four artifacts, ensuring that Dataset-Issue, Dataset-PR, and Dataset-Commits do not contain data from Dataset-M. Finally, we perform the RQ2 experiment in the OTM scenario.

B. Experiment Result Analysis

Table IV presents the F1-score of existing SATD Identification models trained on Dataset-M and unsupervised learning methods for predicting SATDs from other three different software artifacts. The best and worst performance values of each method on different datasets are highlighted in bold and underlined, respectively.

Table IV
F1-score of existing approaches on SATDs across software artifacts.

Dataset	Approach				
	Pattern	MAT	NLP	TM	BERT
Dataset-M (baseline metric)	**0.247**	**0.747**	**0.761**	**0.710**	**0.834**
Dataset-Issue	0.031	0.024	0.242	0.291	0.244
Dataset-PR	0.030	0.032	0.369	0.394	0.385
Dataset-Commits	<u>0.008</u>	<u>0.024</u>	<u>0.145</u>	<u>0.287</u>	<u>0.164</u>
Average	0.023	0.027	0.252	0.324	0.264
Average-Change	*(-90.7%)*	*(-96.4%)*	*(-66.9%)*	*(-54.4%)*	*(-68.3%)*

The result shows that existing approaches experience a significant drop in performance when applied to find out SATDs from three other artifacts. In detail, the Pattern approach exhibits a considerable decline in its F1-score from 24.7% on Dataset-M to 2.3%, with an average drop of 90.7%. Similarly, the MAT approach drops from an F1-score of 74.7% to 2.7%, with an average drop of 96.7%. The other three supervised learning models, NLP, TM, and BERT, drop by 66.9%, 54.4%, and 68.3%, respectively.

Furthermore, we observe that unsupervised models exhibit a more pronounced performance degradation trend compared to supervised models. To elaborate, the two unsupervised approaches, namely *Pattern*, and *MAT* achieve an average F1-score of only 2.5% in identifying SATDs from the other three different software artifacts, resulting in a performance decrease of over 90% compared to the baseline metric. In contrast, the other three supervised learning approaches, namely *NLP*, *TM*, and *BERT*, achieve an average F1-score of 28%, representing a performance drop of over 60% compared to the baseline metric. This suggests that the supervised models may learn more features that are helpful in identifying SATDs across different software artifacts but with limited effectiveness.

The existing supervised models trained on Dataset-M and some unsupervised models achieve an average F1-score of no more than 32.4% (the performance of the TM) in identifying SATDs from the other three different software artifacts. In

summary, the experimental results show that these existing SATD identification approaches can not effectively identify SATDs in issue trackers, pull requests and commit messages.

There are two reasons which may lead to the results: (1) the Java dataset is not representative which does not contain the key features in SATDs across other software artifacts. (2) these existing models cannot generalize on the different software artifacts which are only trained by the Java dataset. These motivate us to use all software artifacts data to train to learn the characteristics of SATD from all software artifacts and use a better model for generalization.

> Previously trained models and unsupervised approaches demonstrate limited effectiveness in identifying SATDs across software artifacts. On average, the F1-score has dropped between -54.4% to -96.4%.

VII. RQ3: Can we build a model to accurately identify SATDs across software artifacts?

A. Multi-task-Learning

Multi-task learning is a machine learning approach that involves training a single model to perform multiple, distinct tasks simultaneously, with the aim of improving the model's overall performance across all tasks. By leveraging shared representations across different tasks, multi-task learning can lead to better generalization and faster learning, while also reducing the need for task-specific models [7], [38]. In addition, multi-task learning can also reduce the training time and hardware resource requirements, as multiple tasks can be trained simultaneously on the same model. Multi-task learning has been successfully applied in various domains such as natural language processing [38], [39], speech recognition [40], and computer vision [41], [42].

B. MT-BERT-SATD Details

For identifying SATDs across different software artifacts (such as issue, PR, commit, and code comments), multi-task learning can learn the different features from them and share parameters when training models to achieve a better generalization than other single-task learning methods. In this study, the SATD identification task for each kind of software artifact is considered as a subtask. Meanwhile, we use BERT which has achieved amazing results in text classification tasks [43] to achieve multi-task learning. Fig.3 shows the overall structure of our approach and its application scenarios. Here, we use the SATD in the collected data (issue trackers, pull requests, commit messages, and code comments) as examples. We validated the generalization ability of our model on new artifacts in Section VIII. We believe that utilizing multi-task learning will have strong generalization capabilities. To incorporate additional information beyond comments, we used SATD labels and artifact-type labels. In the encoder layer, we shared BERT parameters to capture shared information among artifacts, such as phrases indicating poor design or incomplete tests. Four classifiers enable the learning of specific information related to SATD in each artifact. This design effectively captured SATD variances and similarities information in four artifacts. Results confirmed the superiority of multi-task learning models.

In this subsection, we describe the design details of our *MT-BERT-SATD* (Multi-task Learning BERT for SATD Identification) approach.

- *Input Feature:* Given an input sample tokens vector x, $x = [x_1, x_2, ..., x_n]$, where x represents the text representation of a sample, we need to add the [CLS] and [SEP] tokens at the beginning and end of the x vector, respectively. Here, [CLS] stands for "CLaSsification" and always appears at the beginning of the input, while [SEP] stands for "SEPeration" and is used to separate sentence pairs. Since we only have one segment x, [SEP] represents the end of the input. The variable y represents the label of the sample, where $y \in \{0, 1\}$. In this context, $y = 1$ denotes that the corresponding sample belongs to "SATDs", while $y = 0$ denotes that the sample belongs to "non-SATDs". In addition, let z denote the category of the task to which a sample belongs, and $z \in \{1, 2, 3, 4\}$. As shown in Fig.3, the identification of SATDs can be divided into four tasks, where $z = 1$ represents that the sample is from issue trackers, and similarly, when $z = 2, 3$, and 4, it represents that the sample is from pull requests, commit messages, and code comments, respectively. Finally, the input feature is composed of three parts, namely x, y, and z, and can be represented as:

$$F(\phi) = (x(\phi), y^{\theta}, z^{\theta}) \tag{1}$$

- *Embedding:* The input representation x of BERT consists of the sum of three embedding vectors: Token Embeddings, Segment Embeddings, and Position Embeddings [44].
- *Encoder:* During the encoder stage, we use a framework comprising 12 Transformer Blocks, each incorporating a multi-head self-attention layer and a Multilayer Perceptron (MLP). For multi-task training, the data from all four tasks (issue trackers, pull requests, commit messages, code comments) are utilized to jointly optimize the parameters of all 12 layers, with the layer parameters being shared across different tasks.
- *Pooler:* Here, we utilize the pooler layer of BERT by taking the hidden state corresponding to the first token of each input sequence as the pooled representation of the entire sequence. We then perform a linear transformation and a non-linear transformation with a fully connected layer and a *tanh* activation function to obtain the final pooled representation, which is used for our downstream SATD identification task. Specifically, the pooler layer is implemented by the following formula:

$$output = tanh(W \times T[CLS] + b) \tag{2}$$

In this formula, $T = [T_1, T_2, T_3, ..., T_n]$, where $T[CLS]$ denotes the hidden state corresponding to the first token of the model. W is the weight matrix of the fully connected layer, b is the bias vector, and $tanh$ is the activation function.
- *Classifier:* The final classifier layers are designed for each SATD identification task. Each linear classifier is a fully

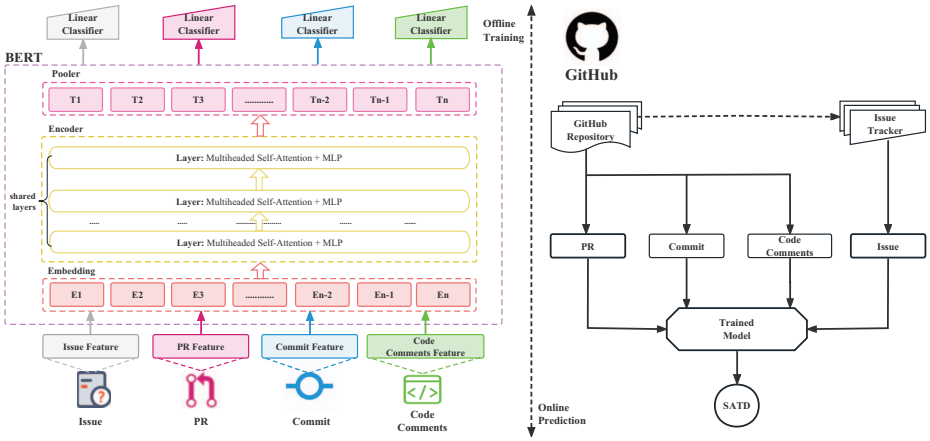

Fig. 3. ***MT-BERT-SATD*** Model Architecture: The model comprises of the Embedding, Encoder, Pooler, and four final classifier layers. Four artifacts of data serve as inputs, which are first encoded through the Embedding layer. The Encoder layer, with shared parameters, adjusts the model parameters. The output of the Encoder layer is then fed into the Pooler layer to obtain the final layer representation of the model output. Finally, the data after Pooler is input into the four sub-classifiers to perform the specific SATD classification task.

connected layer that includes a weight matrix W_i and a bias vector b_i. The pooled hidden state $T[CLS]$ from the pooler layer is used as input, and the output vector is obtained by matrix multiplication and addition of the bias. Then the $softmax$ function is applied to map the output vector to a probability distribution y, which is used for the SATD identification tasks from four different software artifacts. We also fine-tune the classifiers for the four tasks and save the trained weights.

$$y = softmax(W_i \times T[CLS] + b_i), \\ y \in (0,1), i \in \{1,2,3,4\} \tag{3}$$

During the training phase, we minimize the sum of cross-entropy loss functions for SATD identification tasks from the four different software artifacts, with the goal of achieving the best average performance. The loss weight of each task here is adjustable, allowing *MT-BERT-SATD* to pay more attention to certain tasks and thus enhance their importance. The final loss function is represented as follows:

$$\min_{\theta} w_1 \mathcal{L}_{issue}(\theta) + w_2 \mathcal{L}_{pr}(\theta) \\ + w_3 \mathcal{L}_{commit}(\theta) + w_4 \mathcal{L}_{comments}(\theta) \tag{4}$$

C. Experimental Settings

We use Transformer with 12 layers, 768 hidden sizes, and 12 attention heads. The size of the feed-forward layer is 3072. We use BERTAdam as our optimization, with an initial learning rate of 2e-5, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and L2 weight decay of 0.01. We set the learning rate warmup to 0.1. We use a dropout rate of 0.1 on all layers and utilize early stopping to avoid model overfitting. We set four linear classifiers following the pooler. We use gelu [45] as the activation function for the hidden layers. The training loss is the sum of the weighted cross-entropy losses of all tasks [46]. We fine-tune the bert-base model with a batch size of 64. We compute the mean values of precision, recall, and F1-score to evaluate our model using 10-fold cross-validation.

D. Effectiveness of MT-BERT-SATD

We perform 10-fold cross-validation to demonstrate the effectiveness of our approach. Specifically, we remove duplicate data from each training and test set to ensure that test sets do not contain any data from the train set, thus verifying the predictive ability of our approach for new data.

Table V
The performance of our approach *MT-BERT-SATD* over MTM scenario.

Dataset	Metrics		
	Precision	Recall	F1-score
Dataset-Issue	0.659	0.625	0.640
Dataset-PR	0.620	0.636	0.625
Dataset-Commit	0.787	0.675	0.724
Dataset-Comments	**0.862**	**0.855**	**0.859**
Average	0.732	0.698	0.712

Table V presents the precision, recall, and F1-score metrics of *MT-BERT-SATD* for identifying SATDs across software artifacts simultaneously. The best and worst performance values are highlighted in bold and underlined, respectively. Our approach achieves an average F1-score of 71.2% for identifying SATDs from four different software artifacts, with the highest F1-score of 85.9% obtained for SATDs from code comments. However, the performance of SATD identification for the other three artifacts is slightly lower, with an average F1-score of 66.3%. Indeed, this phenomenon can be easily comprehended since SATDs in code comments are often accompanied by apparent prior knowledge, such as the usage of keywords like "TODO" and "FIXME". These prior knowledge cues facilitate the identification of SATDs in code comments. However, the other three artifacts of SATD do not have such explicit prior knowledge. Specifically, OSS developers may report urgent SATDs or discuss SATDs in issue trackers without the above mentioned keywords [29], [30]. These discussions and reports can vary depending on the project and the urgency of the

technical debt, making the identification of SATDs in issue trackers relatively challenging. In addition, developers may discuss existing SATDs in Pull Requests, but such discussions often lack clear indicators [19], which also increases the difficulty of identifying SATDs in Pull Requests.

Overall, our approach achieves a precision of 73.2%, recall of 69.8%, and F1-score of 71.2% in identifying four artifacts simultaneously. These results demonstrate the practical effectiveness of our approach in identifying SATD.

E. Performance Comparison with other Existing SATD Identification Approaches

Table VI
Comparison of F1-score to other existing methods for recognizing SATDs.

Dataset	Approach			
	NLP	TM	BERT	MT-BERT-SATD
Dataset-Issue	0.488	0.478	0.628	**0.640**
Dataset-PR	0.492	0.536	0.617	**0.625**
Dataset-Commits	0.466	0.597	0.652	**0.724**
Dataset-Comments	0.737	0.597	0.827	**0.859**
Average	0.546	0.552	0.681	**0.712**

We now conduct experiments in the MTM scenario comparing our approach with three existing SATD identification models. Among them, BERT, used by Prenner et al [13]., is considered a powerful baseline.

Table VI shows our comparison results. Our approach outperforms NLP and TM, with an average F1-score improvement of 30.4% and 29.0% respectively, across the four different software artifacts. Compared to BERT, our approach achieves a performance improvement of 1.3% to 11.0% on the four datasets. Particularly, on the Dataset-Commit with only 747 SATD samples, our approach outperforms BERT by 11.0%. This evidence indicates that *MT-BERT-SATD* is more flexible for small training corpora. In other words, when adding a new SATD training sample artifact, our approach is more likely to achieve better performance with only a small amount of training data. Overall, our approach achieves an average F1-score of 71.2%, which represents a 4.6% improvement over the BERT proposed by Prenner et al [13]. Additionally, our approach achieves an average precision of 73.2%, which represents an improvement of 11.2% compared to BERT's 65.8%. In comparison, our approach achieves an average recall of 69.8%, which represents a decrease of 1.3% compared to BERT's 70.7%. Based on previous work, developers dislike dealing with false positives (i.e., low precision) [47], [48]. Therefore, Our method performs better than the BERT method overall.

> Our approach, *MT-BERT-SATD*, achieves an average F1-score of 71.2% in identifying SATDs across the 4 artifacts, outperforming existing NLP, TM, and BERT models. Furthermore, our approach requires only a small amount of training data to achieve good performance in detecting SATD from a new artifact.

Table VII
Cross-artifact prediction performance of our approach.

Dataset	New Metrics			Origin	Drop
	Precision	Recall	F1-score		
Dataset-Issue	0.481	0.446	0.463	0.640	-27.7%
Dataset-PR	0.588	0.571	0.579	0.625	-7.36%
Dataset-Commit	0.670	0.566	0.614	0.724	-15.2%
Dataset-Comments	0.553	0.739	0.633	0.859	-26.3%
Average	0.573	0.581	0.572	0.712	-19.7%

VIII. DISCUSSION

A. MT-BERT-SATD Predictions on New Artifacts

In the previous sections, we validate the ability of *MT-BERT-SATD* to identify SATDs simultaneously from four artifacts: issue trackers, pull requests, commit messages, and code comments. However, we want to investigate the predictive performance of our approach for a new artifact of SATDs that has not been trained before. At the same time, we also want to investigate which artifacts have a significant impact on the effectiveness of our model. Inspired by the work of Huang et al. [10], we adopt a sub-classifier voting mechanism to identify SATDs from a new artifact. In detail, for sample data in a new artifact, the four subclassifies in our model will classify it simultaneously. If at least two sub-classifiers consider the sample as SATD, our model marks it as SATD eventually, otherwise, it is marked as non-SATD. Next, we conduct a cross-artifacts prediction experiment, where three artifacts of data are used as the training set, and another artifact of data is used as the test set. Table VII shows our experimental results. The last two columns show the F1-score of our approach in MTM scenarios when the artifact is integrated, as well as the performance degradation ratio of the F1-score in cross-artifact prediction compared to the previous F1-score. The experimental results demonstrate that the absence of Dataset-Issue as an artifact leads to the highest performance degradation of 27.7% in cross-artifact prediction, followed by Dataset-Comments, which causes a decline of 26.3%. This implies that issue trackers and code comments are the two most influential artifacts on the model performance. Moreover, our approach achieves an average F1-score of 57.2% in cross-artifact prediction. It is expected that the integration of all four artifacts of data in our approach can further enhance the performance of our model in predicting new artifacts.

B. Efficacy of MT-BERT-SATD in the wild

Since we just want to evaluate the performance of our tool in the new real dataset, we select the release changelogs as the research object. This is because (1) the release changelog is one of the software artifacts with the richest textual resources, and (2) the SATD in release logs is never explored before.

To explore SATDs in release changelogs in greater detail, we select six popular projects from GitHub that belong to different technology domains in the real world. We collect the release changelogs of these six projects using the GitHub API. In order to obtain smaller granularity sample data, we split the entire release description based on individual change items which are referred to as changelog items. Table VIII

Table VIII
The list of selected real-world popular software projects.

Project	Description	Stars	Contributions	#Changelog Items	#SATD	% of SATD
Angular	A popular framework for building web applications.	87.4k	1697	1978	120	6.1
Bootstrap	A popular front-end development framework.	163k	1358	2115	310	14.7
Numpy	The fundamental package for scientific computing with Python.	23.2k	1465	1779	337	18.9
Pytorch	A popular open-source machine learning framework.	65.1k	2731	8383	765	9.1
React	A JavaScript library for building user interfaces.	206k	1615	716	203	28.4
Tensorflow	An end-to-end open source platform for machine learning.	173k	3353	9764	785	8.0
Average	-	119.6k	2036	4123	420	10.2

displays information about the six projects, including their metadata, such as brief descriptions and the total number of release changelog items, etc. The last two columns of Table VIII present our experimental results. In total, among 24,735 changelog items, *MT-BERT-SATD* marks 2,520 of them as SATD-related. Compared to the source code comments, we find that the release changelogs record more SATD, ranging from 6.1% to 28.4%, with an average of 10.2%. This may be attributed to the preference of OSS developers to summarize the SATD-related information in release changelogs for better software maintenance and upgrade management.

Manual verification of the results predicted by *MT-BERT-SATD*. We randomly pick 60 samples each (a total of 120 changelog items) from the release changelog items that are labeled as having TD and no TD for further analysis. Next, using the SATD classification criteria proposed by Li et al. [17], the first two authors independently and manually flag each of the 120 changelog items as either containing SATD or not containing SATD. Cohen's Kappa coefficient [49] is 0.80, which is considered to be a relatively high level of agreement. In case of inconsistent decisions, the first two authors discuss and reach a consensus. Finally, we compare their decisions with the results obtained from our multi-task learning model. The comparisons show that there are 14 changelog items labeled as non-SATD by authors among 60 changelog items predicted as SATD by our model and 10 changelog items labeled as SATD by authors among 60 changelog items predicted as non-SATD by our model.

Overall, our approach achieves 80%, 78%, 83%, and 80% on Accuracy, Precision, Recall, and F1 respectively in detecting SATDs from the release changelogs in real-world projects. Considering that release changelogs represent a new artifact and the fact that the six popular real-world projects we have selected come from different domains. Therefore, it is impossible for the integrated dataset collected from four known artifacts to fully cover the characteristics of SATDs in new domains. **This shows the generalization ability of our model.** We believe that the performance of our model can be further improved as training data from new artifacts are incorporated into our multi-task learning model.

IX. Threats to Validity

Threats of internal validity. Since this paper aims to explore the real progress of SATD identification in our community, it is essential to select some of the most representative studies. To this end, the approaches investigated in this paper are all selected from top international conferences and journals, which have been widely cited and discussed in our community. In addition, although code for two supervised SATD detection approaches, namely CNN [12] and HATD [34], have not been made publicly available, as reported by Prenner et al. [13], the BERT model performance is similar to that of HATD and significantly higher than that of the CNN model. Therefore, we believe that this threat has been minimized.

Threats of external validity. In this work, we utilize multi-task learning to train a sub-classifier for each of the four known artifacts and investigate how to utilize *MT-BERT-SATD* for identifying SATDs in a new artifact. Specifically, we employ a sub-classifier voting mechanism to identify SATDs in release changelogs. Upon manual verification of the results reported by our approach, we determine that it has attained an accuracy of 80% on 120 randomly chosen changelogs. Our results show that *MT-BERT-SATD* is effective in identifying SATDs in new artifacts beyond the four original artifacts.

X. Conclusion and Future Work

In this paper, we investigate the extent of progress made by our community in the field of SATD identification. We find that existing SATD models can effectively identify SATD in code comments of non-Java projects. However, they are ineffective in simultaneously identifying SATD in the other three new artifacts (i.e., issue trackers, pull requests, and commit messages). Further attention is needed for SATD identification. Additionally, to identify SATD from the four different software artifacts, we propose a multi-task learning model based on BERT, namely *MT-BERT-SATD*. Our experimental results demonstrate that *MT-BERT-SATD* achieves an average F1-score of 71.2% when identifying SATD from all four artifacts, outperforming existing SATD identification methods from 4.6% to 30.4%. Furthermore, we demonstrate that well-trained *MT-BERT-SATD* remains effective in identifying SATD from new artifacts [26] other than the four known ones. To our best knowledge, this means that *MT-BERT-SATD* is the first open-source model capable of identifying SATD from all artifacts. In the future, we will collect SATD from other artifacts to train our model to enhance the generalizability of our model, even though the quantity may be limited.

DATA AVAILABILITY. Datasets and the source code can be available at https://github.com/zscszndxdxs/2023-MT-BERT-SATD.

References

[1] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[2] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *2015 IEEE 7Th international workshop on managing technical debt (MTD)*, IEEE, 2015, pp. 9–15.

[3] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 179–188.

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

[5] Y. Miyake, S. Amasaki, H. Aman, and T. Yokogawa, "A replicated study on relationship between code quality and method comments," *Applied computing and information technology*, pp. 17–30, 2017.

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

[7] Y. Zhang and Q. Yang, "An overview of multi-task learning," *National Science Review*, vol. 5, no. 1, pp. 30–43, 2018.

[8] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks," *Empirical Software Engineering*, vol. 26, pp. 1–36, 2021.

[9] E. da Silva Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[10] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, pp. 418–451, 2018.

[11] Z. Guo, S. Liu, J. Liu, *et al.*, "How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–56, 2021.

[12] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM transactions on software engineering and methodology (TOSEM)*, vol. 28, no. 3, pp. 1–45, 2019.

[13] J. A. Prenner and R. Robbes, "Making the most of small software engineering datasets with modern machine learning," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5050–5067, 2021.

[14] H. Azuma, S. Matsumoto, Y. Kamei, and S. Kusumoto, "An empirical study on self-admitted technical debt in dockerfiles," *Empirical Software Engineering*, vol. 27, no. 2, p. 49, 2022.

[15] D. OBrien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, "23 shades of self-admitted technical debt: An empirical study on machine learning software," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 734–746.

[16] T. Xiao, D. Wang, S. McIntosh, *et al.*, "Characterizing and mitigating self-admitted technical debt in build systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4214–4228, 2021.

[17] Y. Li, M. Soliman, and P. Avgeriou, "Identifying self-admitted technical debt in issue tracking systems using machine learning," *Empirical Software Engineering*, vol. 27, no. 6, p. 131, 2022.

[18] Y. Li, M. Soliman, and P. Avgeriou, "Automatic identification of self-admitted technical debt from four different sources," *Empirical Software Engineering*, vol. 28, no. 3, pp. 1–38, 2023.

[19] S. Karmakar, Z. Codabux, and M. Vidoni, "An experience report on technical debt in pull requests: Challenges and lessons learned," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022, pp. 295–300.

[20] J. Y. Khan and G. Uddin, "Automatic detection and analysis of technical debts in peer-review documentation of r packages," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2022, pp. 765–776.

[21] B. A. Muse, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "Fixme: Synchronize with database! an empirical study of data access self-admitted technical debt," *Empirical Software Engineering*, vol. 27, no. 6, p. 130, 2022.

[22] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, "Identifying self-admitted technical debts with jitterbug: A two-step approach," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1676–1691, 2020.

[23] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? an in-depth perspective," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 526–536.

[24] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, "Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?" In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2019, pp. 186–190.

[25] Y. Li, M. Soliman, and P. Avgeriou, "Identification and remediation of self-admitted technical debt in issue trackers," in *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*, IEEE, 2020, pp. 495–503.

[26] F. Zampetti, G. Fucci, A. Serebrenik, and M. Di Penta, "Self-admitted technical debt practices: A comparison between industry and open-source," *Empirical Software Engineering*, vol. 26, pp. 1–32, 2021.

[27] Y. Li, M. Soliman, P. Avgeriou, and L. Somers, "Self-admitted technical debt in the embedded systems industry: An exploratory case study," *IEEE Transactions on Software Engineering*, 2022.

[28] A. Peruma, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "Refactoring debt: Myth or reality? an exploratory study on the relationship between technical debt and refactoring," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 127–131.

[29] S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt? surfacing elusive technical debt in issue trackers," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 327–338.

[30] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente, "Beyond the code: Mining self-admitted technical debt in issue tracker systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 137–146.

[31] L. Xavier, J. E. Montandon, F. Ferreira, R. Brito, and M. T. Valente, "On the documentation of self-admitted technical debt in issues," *Empirical Software Engineering*, vol. 27, no. 7, p. 163, 2022.

[32] H. Tu and T. Menzies, "Debtfree: Minimizing labeling cost in self-admitted technical debt identification using semi-supervised learning," *Empirical Software Engineering*, vol. 27, no. 4, p. 80, 2022.

[33] Z. Guo, S. Liu, T. Tan, Y. Li, and L. C. and YM Zhou and BW Xu, "Self-admitted technical debt research: Problem, progress, and challenges," *Journal of Software*, vol. 33, no. 1, pp. 26–54, 2021.

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

[35] C. Manning and D. Klein, "Optimization, maxent models, and conditional estimation without magic," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Tutorials-Volume 5*, 2003, pp. 8–8.

[36] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[37] G. W. Imbens and T. Lancaster, "Efficient estimation and stratified sampling," *Journal of Econometrics*, vol. 74, no. 2, pp. 289–318, 1996.

[38] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[39] K. Hashimoto, C. Xiong, Y. Tsuruoka, and R. Socher, "A joint many-task model: Growing a neural network for multiple nlp tasks," *arXiv preprint arXiv:1611.01587*, 2016.

[40] M. L. Seltzer and J. Droppo, "Multi-task learning in deep neural networks for improved phoneme recognition," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 2013, pp. 6965–6969.

[41] S. Liu, E. Johns, and A. J. Davison, "End-to-end multi-task learning with attention," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 1871–1880.

[42] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7482–7491.

[43] D. Croce, G. Castellucci, and R. Basili, "Gan-bert: Generative adversarial learning for robust text classification with a bunch of labeled examples," in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 2114–2119.

[44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[45] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with gaussian error linear units," *CoRR, abs/1606.08415*, vol. 3, 2016.
[46] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.
[47] A. Bessey, K. Block, B. Chelf, *et al.*, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[48] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 50–60.
[49] M. L. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.