electronics

MDPI

Article

Detecting Self-Admitted Technical Debts via Prompt-Based Method in Issue-Tracking Systems

Jiaojiao Yu [1,2], Hao Tian [1,2,3], Ruiheng Li [1,2,*], Qiankun Zuo [1,2] and Yi Di [1,2]

[1] Hubei Key Laboratory of Digital Finance Innovation, Wuhan 430205, China; jojoyu@whu.edu.cn (J.Y.); th@hbue.edu.cn (H.T.); qk.zuo@hbue.edu.cn (Q.Z.); diyi8710@hbue.edu.cn (Y.D.)
[2] School of Information Engineering, Hubei University of Economics, Wuhan 430205, China
[3] Hubei Internet Finance Information Engineering Technology Research Center, Hubei University of Economics, Wuhan 430205, China
* Correspondence: liruiheng@cug.edu.cn

Abstract: Self-admitted technical debts (SATDs) refer to a solution in software development that selects suboptimal solutions to meet the current requirements and are intentionally introduced and documented by developers. SATDs in issue-tracking systems are a complement to those within source code comments. The effective identification of SATDs is crucial for software quality assurance and maintenance. Current studies focus on whether issue sections contain debt, but overlook specific SATD types. Meanwhile, they lack solutions for the challenge that SATD features are hard to learn due to the scarcity of instances containing SATDs. To address these problems, we propose a novel method, which is a weighted prompt tuning to identify SATDs, called WPTD. Specifically, WPTD employs a weighted prompt tuning to adapt the model with few-shot samples for insufficient training data. Moreover, to improve the performance of the model, WPTD constructs an SATD verbalizer by extracting keywords through mutual information and refining it with prior contextual information. Furthermore, it also improves SATD representation by extracting weights using the chi-square method and integrating them into the text. Finally, to reduce bias, WPTD computes the average score of results as final predicted distributions. We conduct comprehensive experiments on seven projects and the results show that our method significantly outperforms baseline approaches. In addition, we summarize the project-specific keywords, which can help developers better understand SATDs.

Keywords: technical debt; SATDs; prompt tuning; issue-tracking systems

check for updates

Citation: Yu, J.; Tian, H.; Li, R.; Zuo, Q.; Di, Y. Detecting Self-Admitted Technical Debts via Prompt-Based Method in Issue-Tracking Systems. *Electronics* **2024**, *13*, 4700. https://doi.org/10.3390/electronics13234700

Academic Editors: Ireneusz Miciuła, Łukasz Radliński, Aleksander Jarzębowicz and Włodzimierz Wysocki

Received: 29 October 2024
Revised: 25 November 2024
Accepted: 25 November 2024
Published: 28 November 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Technical debt is a metaphor for the software development process in which developers choose a sub-optimal solution to meet current objectives [1]. In general, developers will introduce technical debts either unintentionally, due to lack of skills and experience, or intentionally, due to very urgent delivery time and tight budgets [2,3]. However, the accumulation of technical debts will negatively impact on the software quality and its long-term maintenance [4,5]. Therefore, reasonable management of technical debts is effective for developing high-quality software products [6,7]. To better manage and relieve technical debts, researchers have explored many solutions, such as debt categories [8], root cause identification of debts [8], debt elimination [9,10], etc. Among these studies, the most crucial step is how to identify technical debts [11].

In the recent past, a variety of techniques have been put forward by researchers for the classification of technical debt within software development projects. Some researchers have concentrated their efforts on detecting technical debt through the analysis of source code [12]. However, code-level technical debts narrow the possible debt categories, which means it can merely include code-level debts but cannot contain other types of debts, such as Requirement debts [13]. Other researchers focused on detecting debts through

documentation, called Self-Admitted Technical Debts (SATDs) [14]. SATDs are complementary to code-level debts, which have various forms of documentation to record. Previous research [14] has shown that SATDs based on source code comment is currently the main research object. However, SATD based on issue-tracking systems [15] is a refinement of code comment.

Current research on identifying SATDs is primarily focused on extracting features from code comments [16,17]. Code comments are used to annotate the logic, parameters, and returns of a function, which can help developers enhance the readability of the source code [18]. Developers can also record the corresponding debt in the code comments. However, this method still faces the following challenges. First, these records need to be written by developers themselves, who can easily forget to document such information during the software development process. Second, when these debts are updated, it is challenging to track their current status. As an alternative, issue-tracking systems aim to document software issues during the development process [6]. Such issue-tracking systems can assist developers in tracking and discussing SATDs during software development. Issue-tracking systems are essential tools for software development teams, enabling them to systematically track issues, allocate tasks, and record defects. These systems are crucial for managing and resolving issues more efficiently, which in turn improves the quality and efficiency of project delivery. Issues are used to document a variety of occurrences during software development that require resolution, such as software defects, functionality enhancements, and technical tasks. Each issue in the system is composed of several sections, including descriptions, comments, and summaries. These sections are vital for complementing code comments and recording SATDs [11,15].

1.1. Motivation

We show a set of examples that include different issue sections and SATD types in Figure 1. The issue ID is CHROMIUM-11934, which shows an example of a Requirement debt included in a comment, the developer indicates that the current issue is inside his to-do list, but the task is ignored and will be processed. An individual from the development team finds the current issue and the developer can update the progress of the current issue by commenting on the recorded issue and marking the SATD record. The description of issue ID CHROMIUM-22066 records that extensive tests have been introduced to test the debt. The description of issue ID GERRIT-1091 records a Documentation debt for outdated documentation due to documentation not being updated instantly. The summary of the issue ID HADOOP-10485 records a code debt for dead code. From Figure 1, we observe that issue sections can track and record the status of SATDs to enable developers to better understand and track SATDs.

Code debt refers to code that is difficult to understand and maintain due to excessive complexity, lack of modularity, or poor coding practices, as well as code that contains errors, has not been thoroughly tested, or does not comply with coding standards, both of which contribute to technical debt and impact the long-term health of a software project. Design debt usually also refers to code sub-optimal choices and is also a type of code debt, so these two types of debt have not been studied separately in previous studies [11,14]. Therefore, we will continue with the previous studies [11,14] by combining these two types of study. Documentation debt is substandard documentation quality, as evidenced by incomplete, inaccurate, or poorly written documentation, as well as out-of-date documentation that does not reflect the current state of the software, which hampers the maintainability and understandability of the software and leads to potential confusion and errors. Requirement debt is when the requirements specified in a software development project are not fully satisfied or are only partially satisfied, resulting in the functionality or quality of the final product being compromised. Test debt occurs when the testing process is suboptimal, such as when only a portion of the code base is tested or when testing is inadequate, which can lead to undiscovered defects and reduced software reliability. From the above, we observe that the issue-tracking system can record SATDs during the software development and can

track SATD changes. It also records the understanding of SATDs by different developers, so it is important research for SATDs in the issue-tracking system.

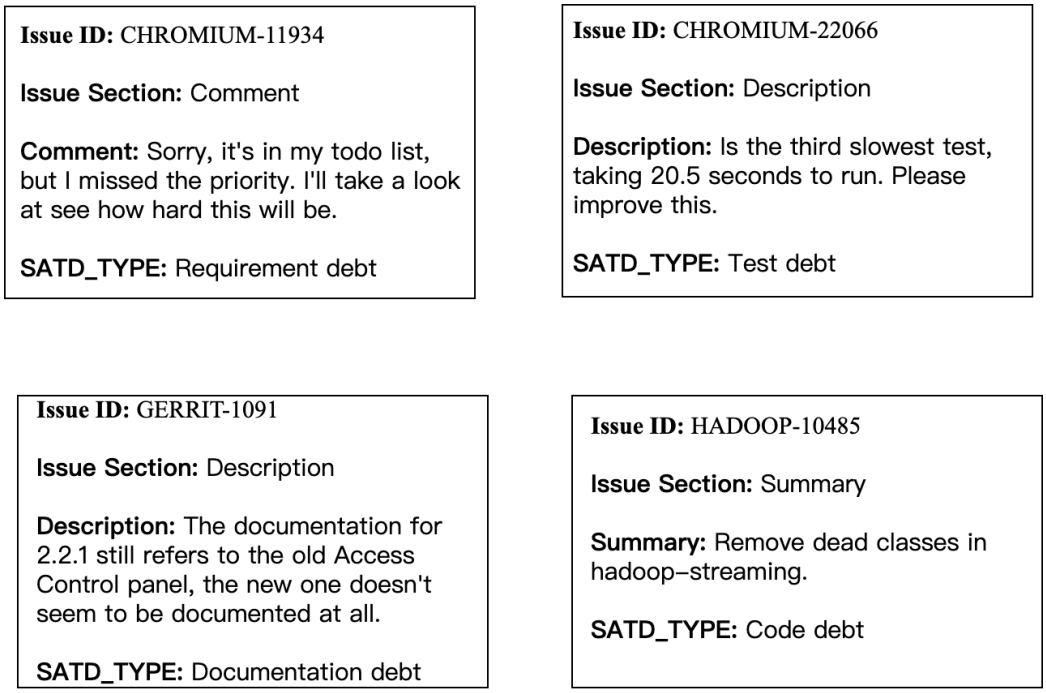


Figure 1. An example of issue sections with different SATD types.

However, it still has the following three limitations. ***Limitation 1***: Existing studies [11,19] neglected to identify specific types of SATDs for the issue sections in the issue-tracking systems. ***Limitation 2***: Previous studies have shown that only 1.86% of code comments contained SATDs [14] and 14.1% of issue-tracking systems included SATDs [11]. The scarcity number of SATDs makes feature representation learning difficult. Existing studies [11,19] did not consider the training of models with little data for efficiently identifying specific types of SATDs. ***Limitation 3***: Existing studies [11,20] depended on the summarized keywords corresponding to specific types of SATDs across multiple projects, but there is a lack of extraction and analysis of each type of SATDs within each project. The significant differences between distinct projects will lead to a considerable variation in the representation of SATDs within each project [21]. In addition, different types of SATDs are always described in distinct ways [5]. Thus, keyword analysis for different projects can provide more insights to improve the generalization of designed detection models.

1.2. Our Work

We propose WPTD, a weighted prompt-tuning method to identify specific types of SATDs in the issue-tracking systems. To enable the model to deal with the identification of specific SATDs, WPTD first constructs and refines the SATD verbalizer, which can improve the predictive performance for specific types of SATDs (*for solving L1*). More specifically, WPTD employs mutual information (MI) feature extraction to construct an SATD verbalizer and prior contextual information to refine the SATD verbalizer, enhancing the association between the features of specific types of SATD and their labels. Then, WPTD utilizes the Chi-Square method to extract the SATD features in the corpus and adds the weight of feature representations into the tokenizer, which increase the weight of the SATD features so as to enhance the SATD feature representations. To improve the model's ability to learn features with a few available samples, WPTD applies the prompt-based method to train the model, which uses the context provided by the prompts to adapt that model, thus avoiding the need to use a large amount of data to train the model for the task adaptation (*for solving L2*). Finally, WPTD calculates the mean distribution probability of each type of

SATD followed by a softmax function to detect which type an SATD belongs to. To facilitate the analysis of different perspectives of the technical debts, WPTD adopts the MI algorithm to extract keywords contained in specific types of SATDs in each project. As a result, we can analyze the differences and relationships of the keywords across both different projects and different types of SATDs (*for solving L3*).

We have conducted extensive experiments on a benchmark dataset encompassing seven projects that utilize two prevalent issue-tracking systems, namely Jira and Google Monorail. The objective of these experiments was to assess the efficacy of our approach in identifying specific types of SATDs. We relied on three key performance indicators: Precision, Recall, and the F1-score. Our method demonstrated superior results when compared to established baseline methods. Specifically, it achieved higher values for Precision, Recall, and F1-score, which are 0.5544, 0.6346, and 0.5509, respectively. These figures indicate that our method is more accurate in detecting SATD instances within the dataset, offering a more reliable tool for software maintenance and quality assurance processes.

1.3. Contributions

The major contributions of this paper are summarized as follows:

- We propose a method for automatically detecting specific types of SATD based on a weighted prompt, named WPTD. Specially, this is the first work focusing on applying a few available samples to train the model for detecting SATDs.
- We summarize a set of key terms that represent various categories of SATDs within each project, which helps researchers gain a deeper understanding of technical debt.
- We conduct extensive experiments to evaluate the performance of our method and compare with baseline methods on a public dataset. The demonstration shows that our method consistently exceeds the performance of these baseline methods.

1.4. Paper Organization

The remainder of this paper is organized as follows. Section 2 provides the related work. In Section 3, we present a comprehensive description of our proposed model. The experimental design and a detailed analysis of the results are covered in Section 4. Section 5 is dedicated to discussing the implications of various experimental conditions, potential threats to the validity of our findings, and the broader implications of our work. We come to a conclusion in Section 6.

2. Background and Related Work

2.1. SATD in Issue-Tracking System

The issue-tracking system is used to help teams manage and resolve these issues more efficiently, thereby improving the quality and efficiency of project delivery. In software development, issues are frequently utilized to track software defects, functionality enhancements, and technical tasks. SATDs take place during the entire software lifetime, so issues or issue sections have the potential to contain debt in issue-tracking systems. Moreover, an issue may contain multiple SATDs. The process of tracking and resolving issues produces intermediate results that may lead to other debts, and these are documented in detail in the issue section.

Hence, we focus on studies that detect SATDs in issue sections. Issue sections can demonstrate the evolution of debt at a fine-grained level, and developers can track records in issue sections to understand the source of debt that is created. Detecting SATDs in issue sections can help developers to better understand the debt and provide a basis for subsequent elimination efforts.

There are some studies on SATDs in issue-tracking systems. Bellomo et al. [6] checked 1264 issues manually in the four issue-tracking systems for open-source projects and found 109 instances of technical debt. Their research shows that in issue-tracking systems, developers discuss and explicitly state technical debt. Kenton and Toutanova [22] extracted a method using a Naive–Bayes classifier to automatically classify whether issue summaries

and descriptions contain SATDs or not. Their experiments show that their method obtained an outstanding performance and commonly used words can be used as indicators of technical debt. Xavier et al. [23] studied SATDs in issue-tracking systems by collecting 286 SATDs. Their experiments showed that only 29% of the SATD instances could be traced to source code, and there is still substantial potential for the design of new tools to support the management of technical debt. Li et al. [15] used a manual approach to identify multi-types of SATDs in issues on two open-source projects. Their experiments indicate that the repayment of SATDs is mainly carried out by those who discovered or created the SATD. Li et al. [11] was the first to propose a machine learning approach to identify SATDs in issue sections. They collected and manually analyzed 4200 issues containing 23,180 sections, and then used a machine learning approach to identify SATDs. Their experiments showed that their approach was superior to the baseline approach in the task of identifying SATDs.

2.2. Prompt Tuning

Prompt tuning refers to the process of fine-tuning a large-scale language model to better adapt it to a specific task [24]. It is used in many tasks of natural language processing, including classification [25,26], natural language understanding [27], and relation extraction [28].

There are some studies focusing on prompt tuning [29,30]. Brown et al. [31] proposed the prompt tuning first and the large language model GPT-3, which made use of an extremely large number of parameters and training datasets. Their experiments showed that a sufficiently large language model with one-shot or few-shot samples could equal or even surpass the start-of-the-art (SOTA) performance of fine-tuned models on some tasks. Schick and Schütze [32] proposed a prompt method that transforms the task into a "pinch-and-snip" problem containing the task, which was further improved by combining unlabeled data and gradient optimization. Their experiments showed that their method achieved the performance of GPT-3 even with a small-scale language model. Gu et al. [33] proposed the PPT prompt tuning framework, which transformed similar classification tasks into a united task format, followed by soft prompts for pre-training. Their experiments showed that their method was effective on large-scale pre-training models.

Prompt tuning improves the model performance by optimizing templates, parameter tuning, and verbalizers [31].

3. Proposed Method

In this section, we present a WPTD method, which consists of four steps. The first step is an SATD verbalizer construction, the second step is an SATD verbalizer refinement, the third step is weighted prompt tuning, and the fourth step is model training. Below, we provide a detailed description of these four steps.

3.1. Overview

Figure 2 shows an overview of our method, including the following steps. In order to effectively extend the diversity of labels, we utilize the mutual information (MI) method to extract the features of each SATD type, which constructs the SATD verbalizer. Then, we refine the SATD verbalizer by contextual prior knowledge [34] to remove the low-frequency invalid words. Next, to improve the performance of the model, we obtain the weight of SATD features which employ the Chi-square (CHI) method to extract and obtain the predicted distribution of mask position on the SATD verbalizer by prompt tuning. Finally, we compute the average score of the SATD verbalizer in each type and map the predicted distribution into labels. Followed by employing a softmax function, we can accurately identify specific types of SATDs within issue-tracking systems.

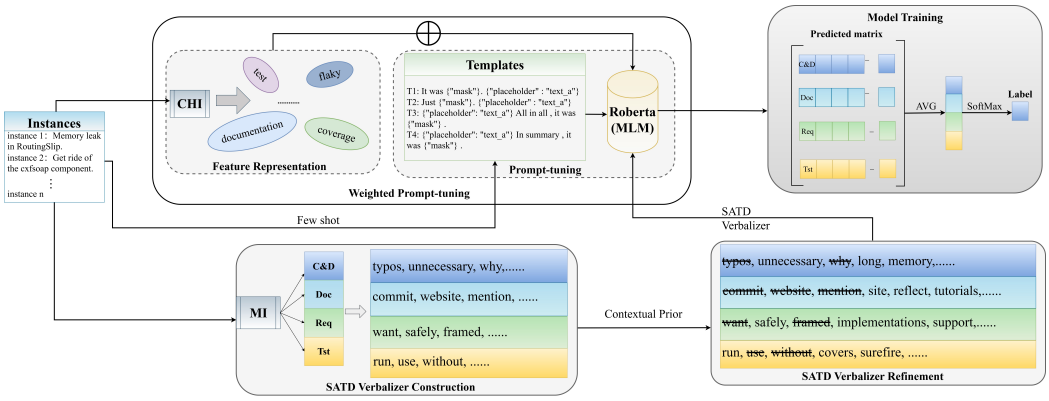


Figure 2. Overview of our method.

3.2. SATD Verbalizer Construction

Prompt tuning transforms the text classification task into a masked language modeling (MLM) task [35]. It wraps the input data with templates and utilizes the MLM model to infer the masked words by the contextual information that is not masked. The labels are transformed into a label-word space that is a verbalizer [34,36], and the MLM focuses on the word prediction distribution for the words in the verbalizer at the masked positions. However, there is no uniqueness in context-based masking position prediction, and multiple words can express the intended meaning. Therefore, the predicted words for the masked position should have broad coverage and low subjective bias [36]. It can attempt to enhance the performance of predicting distribution for the masked position by effectively increasing the diversity of words at the masked position. Therefore, we extract keywords from the corpus to construct an SATD verbalizer for improving the performance of predicted distribution in mask position. Additionally, each SATD instance is treated as an independent sample when extracting SATD keywords, and each non-repeated word is a feature. Thus, the corpus constitutes a high dimensional feature representation [37], leading to increased computational complexity and sparse feature distributions [38,39]. To effectively extract keywords from the corpus, feature-selection methods are widely applied in software engineering tasks [18], which can effectively reduce the dimensionality and minimize the noise.

In our method, we employ the MI feature selection [40,41] to select the relevant features for each type as the keywords for constructing the SATD verbalizer. As MI does not require any assumptions about the properties of the relationships between keywords and categories, it is well-adapted to the identification of features and categories for text classification [42].

Figure 3 shows the mutual information of features and categories, where $W = \{w_1, w_2, w_3, ..., w_o, ...w_t, ...w_n\}$ denotes the word set which is composed of unduplicated words of the corpus and contains n words. $C = \{c_1, c_2, c_3, c_4\}$ denotes the categories of SATD. $p(W)$ represents the marginal probability distribution of words in the corpus. $p(C)$ represents the marginal probability distribution of categories. $p(W, C)$ denotes the joint probability distribution function of the word set and categories. From this figure, we can see that the mutual information $I(W;C)$ is the overlap area between $p(W)$ and $p(C)$. The formula for computing $I(W;C)$ between features and labels are defined as follows:

$$I(W;C) = \sum_{c \in C} \sum_{w \in W} p(W,C) log(\frac{p(W,C)}{p(W)p(C)}) \quad (1)$$

We calculate the mutual information between each word and the category. The scores of the mutual information are ranked for each category, and those features that have the highest scores are chosen as the keywords to construct the SATD verbalizer. In our method, we extracted the top 20 keywords to construct the SATD verbalizer. Thus, we evolve label set C into the SATD verbalizer (SV) by each type of SATD keyword.

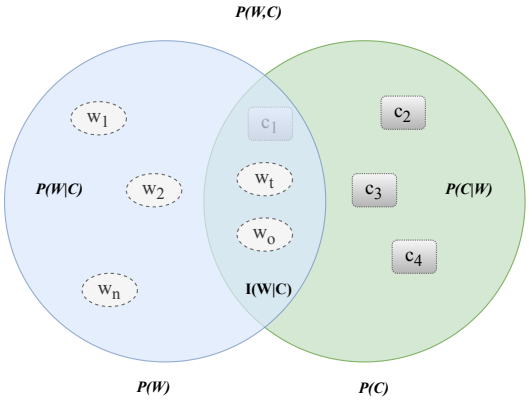


Figure 3. The mutual information of features and categories.

3.3. SATD Verbalizer Refinement

With the previous step, we constructed the SATD verbalizer for diverse words. These words are obtained based on calculating the mutual information between categories and instances. Therefore, they contain some low-frequency words and cannot adapt to the Pre-Trained Language Model (PLM) because they are not trained by the PLM. To solve this problem, we use contextual prior knowledge [34] to filter these words so that the words follow the distribution of the context and improve the predictive distribution of the model over the words.

We take the SATD instances input prompt tuning and use the model to adapt the distribution of instances for contextual prior knowledge. Based on previous studies [30,43], we select a manual template [26] to render SATD instances, which contains a wealth of expert knowledge. The template module shown in Figure 2 contains detailed information about the template. Each instance is rendered by these four sub-templates in the template. These templates replace the mechanism in the MLM model that randomly masks positions. By masking the specific position of the input, prompt tuning guides the model to focus on the specific position, thereby enhancing its ability to identify the specific types of SATD [36].

After rendering the instances using the template, we obtain texts containing mask positions. We chose Roberta [44] as the PLM and use its MLM model to calculate the predicted probability of each label word in the SV at the mask position. By using the probability expectation of the entire distribution of instances, we can obtain the prior distribution of label words at mask positions; the formula is as follows:

$$a = T(A) \tag{2}$$

$$P_Z(l) = \mathbb{E}_{a \sim Z} P_{mask} \tag{3}$$

where A is an SATD instance, A is wrapped by T and the result is a, Z denotes the distribution of instances in the corpus, l is the label words, and $l \in SV$, p_{mask} is the mask position in a.

In particular, in our method, we chose a small subset of instances $\tilde{S}$ as the corpus of the contextual prior, and we do not use the original labels. Thus, the contextual prior distribution for this small portion of text can be expressed as follows:

$$P_Z(l) \doteq \frac{1}{|\tilde{S}|} \sum_{a \in \tilde{S}} P_{mask} \tag{4}$$

The predicted distribution of each word in the SATD verbalizer is computed by the mask position of subset instances. For better word selection, we set a distribution threshold

(cutoff). The word is removed from the SATD verbalizer when the probability distribution of the word is less than the threshold. The formula is as follows:

$$\tilde{SV}_i = \begin{cases} SV_j & \text{if } p_D(SV_j) < \text{cutoff} \\ continue; & \text{if } p_D(SV_j) \geq \text{cutoff} \end{cases} \tag{5}$$

where $\tilde{SV}$ is the SATD verbalizer refinement, i denotes the i-th words in $\tilde{SV}$, j denotes the j-th words in SV.

3.4. Weighted Prompt Tuning

After the above, we determined the template, SATD verbalizer, and the required PLM (Roberta). Next, we need to use the information that is confirmed to complete the process of weighted prompt tuning.

Feature representation. To enhance the SATD feature representations, we utilize the CHI feature-selection method for SATD feature extraction on the corpus, which can examine the independence of words W and classes C; the formula is as follows:

$$CHI(w_j, c_i) = \frac{N(FJ - GH)^2}{(F+H)(G+J)(F+G)(H+J)} \tag{6}$$

where N is the total number of corpora, F represents the number of instances containing w_j and belong to c_i, G is the number of instances containing w_j but not to c_i, H refers to the number of instances that do not contain w_j but belong to category c_i, and J refers to the number of instances that do not contain w_j and not belong to c_i. The final CHI value of word w_j is calculated as follows:

$$CHI(w_j) = \sum_i^4 P(c_i) CHI(w_j, c_i) \tag{7}$$

where $P(c_i)$ denotes the probability that an instance belonging to category c_i occurs in all documents and the total number of classes is 4. It is worth noting that we chose the top 10% words as the key features of SATD, as the set $\mathbb{F}$.

Next, we transform representation $\mathbb{F}$ into vectors by CountVectorizer (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html (accessed on 27 January 2024)). CountVectorizer extracts unique words and sorts them alphabetically. For each instance, it generates an n-dimensional vector, where n is the number of unique words in the corpus. Specifically, we input a set of keywords to the conversion vector; each word is treated as a separate document. We convert $\mathbb{F}$ into a vector by using CountVectorizer, as the feature representation $\overrightarrow{\mathbb{F}}$.

We take the instance "Remove dead classes in hadoop-streaming" in Figure 1 as an example of feature representation. The instance contains six words. The six words are, respectively, targeted to four categories to compute the value of CHI according to Equation (6), and then compute the CHI of the words against the six words according to Equation (7). Finally, the top words are selected as the key features and converted into vectors by CountVectorizer.

Prompt tuning. It is costly to retrain the pre-trained model, so we employ fine-tuning to train the model, thus saving computing resources and time costs. Moreover, from the Table 1, we can observe that the proportion of documents containing SATD is very small. Thus, we utilize weighted prompt tuning to train the model, which is particularly effective for adapting to scenarios with limited data. Additionally, prompt tuning makes the model focus on the predicted distribution at mask-specified positions, potentially improving the performance of the model without extensive parameter adjustments.

Table 1. Details of dataset.

Projects	System	#Issue Sections	Req		C&D				Tst		Doc	
					Desgin		Code					
			#	%	#	%	#	%	#	%	#	%
Camel	Jira	2792	8	0.29%	111	3.98%	89	3.19%	41	1.47%	103	3.69%
Chromium	Google	3435	27	0.79%	105	3.06%	83	2.42%	29	0.84%	8	0.23%
Gerrit	Google	2812	12	0.43%	89	3.17%	40	1.42%	10	0.36%	37	1.32%
Hadoop	Jira	4515	130	2.88%	156	3.46%	397	8.79%	79	1.75%	16	0.35%
Hbase	Jira	4936	8	0.16%	193	3.91%	255	5.17%	90	1.82%	113	2.29%
Impala	Jira	1934	5	0.26%	144	7.45%	118	6.10%	41	2.12%	36	1.86%
Thrift	Jira	2756	20	0.73%	137	4.97%	265	9.62%	48	1.74%	59	2.14%
Average	-	3311	30	0.79%	134	4.03%	178	5.38%	48	1.44%	53	1.70%

Prompt tuning can adapt the model with a small amount of data. Therefore, we randomly selected some samples from each of the four SATD categories as training data for the model. The training data are rendered by the template. We denote the X as an input instance, which contains a type SATD; we render it by four sub-templates as follows:

$$X_1 = \text{[CLS] It was [MASK]. } X.$$

$$X_2 = \text{[CLS] Just [MASK]! } X.$$

$$X_3 = \text{[CLS] } X\text{. All in all, it was [MASK].}$$

$$X_4 = \text{[CLS] } X\text{. In summary, it was [MASK].}$$

where [CLS] is a special token used to denote the start of a sequence. Next, we need to use the Roberta to tokenize the rendered text. To enhance the representation of SATD features in the corpus, we add the weights of the feature vector $\overrightarrow{\mathbb{F}}$ obtained in the previous step, i.e., Equations (6) and (7). Finally, we utilize the MLM of Roberta to predict the probability of the label word in $S\tilde{V}$ at the mask position for each template. We take the instance "Remove dead classes in hadoop-streaming" in Figure 1 as an example of prompt tuning. We render the instances by X_1, X_2, X_3, and X_4, we replace X with "Remove dead classes in hadoop-streaming", and then tokenize the converted X_1–X_4 by Roberta. We obtain the feature representation in the previous step, and if the feature representation is in an instance, we add the corresponding weight to the instance tokenizer. Finally, we predict four type labels for mask positions on X_1–X_4.

3.5. Model Training

After obtaining the probability distribution of label words in $S\tilde{V}$, we need to determine the final classification predicted by the mask position. In our method, we employ average weighting to determine the final label probability. Simultaneously, there exists an inherent bias towards the weights of words in the SATD verbalizer when selecting some samples as the training set. It can result in significant variability in the performance of the model, potentially compromising the generalizability of the model. Hence, we assume that each label word in each category contributes with equal weight to the predicted label. We utilize the average of predicted scores across each label word as the final prediction score. We compute the prediction scores of the label words for each category at the mask position as the following formula:

$$\hat{l_v} = \frac{\sum_{v \in S\tilde{V}} P_{mask}([MASK] = v | x_p)}{|v_l|} \tag{8}$$

where $v \in S\tilde{V}$, v denotes the label words in $S\tilde{V}$, v_l is mapped into the specific label. We select the highest-predicted probability in four sub-templates, map the probability of label

words $\hat{l_w}$ into the probability of labels, and finally the cross-entropy loss function is utilized to calculate the final prediction score with the following formula:

$$Loss(\tilde{p}_{\hat{l}}, \tilde{p}_t) = -\sum_{i=1}^{R} \tilde{p}_t^i log(\tilde{p}_{\hat{l}}^i) \quad (9)$$

where R is the total number of corpus, $\tilde{p}_l$ is the predictive category distribution, and $\tilde{p}_t$ is the true category distribution.

When the model is fully trained, we render the input corpus by template in the inference step, put the rendered data into the trained model, and classify the samples by calculating the average scores of the SATD verbalizer and mapping them into labels. In this paper, we chose to optimize the verbalizer to boost the performance of WPTD. Specifically, we extract keywords from the corpus to construct the verbalizer for SATD. Additionally, we utilize contextual prior knowledge to refine the SATD verbalizer by selecting keywords that align with the textual distribution. The SATD verbalizer enhances the diversity of predicted words in the masked positions and improves the predicted word distribution at mask positions. As a result, the ability of the model to detect specific types of SATD is effectively enhanced.

4. Experimental Setup and Results

4.1. Experimental Setup

Dataset: To assess the performance of WPTD, we conducted experiments on a benchmark dataset [11], which analyzed the issues of seven open-source projects on Jira and Google Monorail, and included various issue sections such as the summary, description, and comments. They extracted 23,180 issue sections from 4200 issues. The dataset contains seven projects including Camel, Chromium, Gerrit, Chromium, HBase, Impala, and Thrift. Camel (https://github.com/apache/camel (accessed on 27 January 2024)) is an open-source framework for integrating data systems. Chromium (https://github.com/chromium/chromium (accessed on 27 January 2024)) is a web browser. Gerrit (https://github.com/GerritCodeReview/gerrit (accessed on 27 January 2024)) is an open-source code review software. Hadoop (https://github.com/apache/hadoop (accessed on 27 January 2024)) is a distributed systems framework. HBase (https://hbase.apache.org/ (accessed on 27 January 2024)) is a distributed and column-oriented open-source database. Impala (https://impala.apache.org/ (accessed on 27 January 2024)) is a query system. Thrift (https://github.com/apache/thrift (accessed on 27 January 2024)) is a software framework for remote procedure calls. The dataset (https://github.com/yikun-li/satd-issue-tracker-data (accessed on 27 January 2024)) exhibits a high degree of coherence, as indicated by a Cohen's kappa coefficient that exceeds 0.7. In particular, we focus on SATDs that are included in all datasets, and categories that have a minimum number of training data that are satisfied in most projects [45]. Meanwhile, it was found in previous studies that usually Code and Design debts are categorized into one type of debt [19,46], so this study focuses on the following four types of debts, namely Code/Design debt, Requirement debt, Documentation debt, and Test debt.

Table 1 shows the details of the dataset, including the name of projects (Projects), the source of issue-tracking system (System), the number of SATD issue sections in each project (#Issue sections), the number of specific-type SATD in each project, i.e., Requirement debt (Req), design and code debts (C&D), Test debt (Tst), and Documentation debt (Doc). # denotes the count of the specific types of SATD and % denotes the proportion of the specific SATD types. From Table 1, we can find that issue sections containing SATDs are maintained at a low number, and the percentage of SATDs in each category is extremely diffuse.

Performance Indicators: In our method, we utilize Precision, Recall, and F1-score as the key indicators to assess the effectiveness of our method. These three indicators are well-recognized and frequently used in software engineering for evaluating [47–51], and popularly employed in identifying SATD in source code comments [18,20,21,45] as well.

Setup: In this paper, we conduct our experiments on the deep learning library Pytorch (https://pytorch.org/ (accessed on 27 January 2024)) and openPrompt (https://github.com/thunlp/OpenPrompt/ (accessed on 27 January 2024)), run on a Linux system (Ubuntu 16.04 LTS) with 64 GB memory and RTX2080Ti GPU. For the PLM, we use RoBERTa-base for all experiments. We set the learning rates as 1×10^{-4}, the batch size as 10, and epoch as 50. In refining the label words, we set the thresholds (cutoff) as 0.5, the size of the unlabeled support set $|\tilde{S}|$ as 200. Since we chose a few-shot samples as training input data, to ensure the validity of the experiment, we repeat it 10 times to randomly select samples as input samples.

4.2. RQ1: Does Our Method Perform Better in Detecting Specific Types of SATD in Issue-Tracking Systems?

Motivation: This RQ explores whether our method outperforms the existing methods for identifying specific types of SATD in issue-tracking systems.

Methods: To answer this question, we selected five benchmark methods for comparative experiments, including MultiCNN (Li et al. [19]), CNN (Li et al. [11]), NBM (Huang et al. [16]), BiLSTM (Yu et al. [17]), and BERT (Kenton et al. [52]).

- MultiCNN: Li et al. [19] proposed a multitask-CNN method for identifying SATDs, which is the state-of-the-art method for detecting SATD tasks.
- CNN: Li et al. [11] employed CNN to identify SATDs or non-SATDs in issue-tracking systems, which utilized fasttext [53] to train the word embeddings and weighted loss to balance the data.
- NBM: Huang et al. [16] proposed an automated method for detecting SATDs in source code comments, which used the Information Gain (IG) to select useful features, built a classifier for each project by NBM, and employed a voting mechanism to predict whether a code comment contains SATDs or not.
- BiLSTM: Yu et al. [17] proposed an improved BiLSTM [54] with the attention mechanism for identifying SATDs in source code comments.
- BERT: Aiken et al. [52] proposed a BERT-based method for identifying SATD features and employed data augmentation through duplication to address the challenge of data imbalance.

Results: Table 2 presents a comparative performance of our method versus the baseline methods across the metrics of Precision, Recall, and F1-score. The table shows that the average Precision of our method is 0.5544, and achieves improvements by 140.21%, 55.83%, 3.57%, 86.13%, and 58.03% compared with MultiCNN, CNN, NBM, BiLSTM, and BERT, respectively. Moreover, in terms of the average Precision of each project, our method demonstrated superior performance in four out of the seven projects when compared to the five baseline methods. The average Recall value of our method is 0.6346 and improvements by 152.23%, 149.84%, 38.95%, 130.01%, and 153.84% compared with MultiCNN, CNN, NBM, BiLSTM, and BERT, respectively. In addition, in terms of the average Recall of each project, our method demonstrated superior performance to the five baseline methods in seven projects. The average F1-score of our method is 0.5509 and achieves improvements by 160.23%, 159.41%, 16.28%, 126.06%, and 167.75% compared with MultiCNN, CNN, NBM, BiLSTM, and BERT, respectively. In particular, in terms of the average F1-score of each project, our method demonstrated superior performance in five out of the seven projects when compared to the five baseline methods.

Table 2 presents a comparative performance of our method versus the baseline methods across the metrics of Precision, Recall, and F1-score. The table shows that the average Precision of our method is 0.5544 and achieves improvements by 140.21%, 55.83%, 3.57%, 86.13%, and 58.03% compared with MultiCNN, CNN, NBM, BiLSTM, and BERT, respectively. Moreover, in terms of average Precision of each project, our method demonstrated superior performance in four out of the seven projects when compared to the five baseline methods.

Table 2. The performance of WPTD and 5 baseline methods on 7 projects.

Projects	Precision						Recall						F1-Score					
	MultiCNN	CNN	NBM	BiLSTM	BERT	WPTD	MultiCNN	CNN	NBM	BiLSTM	BERT	WPTD	MultiCNN	CNN	NBM	BiLSTM	BERT	WPTD
Camel	0.2425	0.2841	0.5524	0.3505	0.2841	0.6342	0.2535	0.2500	0.4702	0.2835	0.2500	0.7099	0.1976	0.1812	0.4940	0.2413	0.1812	0.6045
Chromium	0.4370	0.3740	0.5070	0.4357	0.3725	0.5417	0.2586	0.2500	0.4071	0.2519	0.2500	0.6306	0.2306	0.2140	0.3793	0.2273	0.2135	0.5232
Gerrit	0.1715	0.3431	0.3663	0.2980	0.3431	0.6008	0.2500	0.2500	0.3753	0.2798	0.2500	0.6448	0.2035	0.2035	0.3693	0.2586	0.2035	0.5954
Hadoop	0.2134	0.3554	0.5838	0.3028	0.3554	0.4928	0.2490	0.2500	0.4907	0.2668	0.2500	0.6009	0.2153	0.2077	0.5214	0.2420	0.2077	0.5106
Hbase	0.1700	0.3455	0.5635	0.1706	0.3399	0.4891	0.2500	0.2655	0.4458	0.2561	0.2500	0.5568	0.2023	0.2359	0.4867	0.2156	0.2023	0.4726
Impala	0.1910	0.3808	0.6733	0.2119	0.3808	0.6180	0.2500	0.2500	0.5708	0.3321	0.2500	0.6932	0.2165	0.2162	0.6063	0.2826	0.2162	0.6265
Thrift	0.1900	0.4076	0.5009	0.3155	0.3800	0.5039	0.2500	0.2624	0.4369	0.2613	0.2500	0.6059	0.2159	0.2281	0.4594	0.2385	0.2159	0.5233
Average	0.2308	0.3558	0.5353	0.2979	0.3508	0.5544	0.2516	0.2540	0.4567	0.2759	0.2500	0.6346	0.2117	0.2124	0.4738	0.2437	0.2058	0.5509

Table 3 shows each type of performance of our method and comparative methods for each type of performance across the average of Precision, Recall, and F1-scores. The table shows that the averages of Recall and F1-score will be lower than the other methods when our method detects the Code and Design debt. In Table 1, it can be observed that the number of Code and Design debt is the most prevalent, which means the method can obtain more of the features of Code and Design debt. In other words, the method only obtains fewer features than the other three types of SATD. Thus, these methods can obtain good performance on Code and Design debt. However, these methods can obtain a worse performance on other SATD types. It is worth noting that our approach can obtain good performance on this low number of texts. Moreover, we have benchmarked WPTD against the state-of-the-art (SOTA) [19] method to detect SATD in issue-tracking systems, which are presented to be more effective than other mainstream SATD classification methods. Therefore, we can infer that our method surpasses other prevalent SATD classification methods in issue tracking systems.

Table 3. The performance of WPTD and 5 baseline methods on each SATD type.

Methods	Precision				Recall				F1-Score			
	C&D	Req	Doc	Tst	C&D	Req	Doc	Tst	C&D	Req	Doc	Tst
MultiCNN	0.7029	0.0000	0.0771	0.1429	0.9914	0.0000	0.0086	0.0043	0.8214	0.0000	0.0157	0.0100
CNN	0.7033	0.0165	0.7033	0.0000	0.9887	0.0073	0.0073	0.0127	0.8200	0.0000	0.0101	0.0194
NBM	0.7893	0.2655	0.5537	0.5327	0.9007	0.1535	0.4612	0.3113	0.8425	0.1786	0.4818	0.3921
BiLSTM	0.7152	0.1905	0.0000	0.2857	0.9644	0.0339	0.0827	0.0227	0.8164	0.0548	0.0636	0.0400
BERT	0.7016	0.0000	0.7016	0.0000	1.0000	0.0000	0.0000	0.0000	0.8230	0.0000	0.0000	0.0000
WPTD	0.8682	0.2718	0.5091	0.5683	0.7013	0.4383	0.5801	0.8187	0.7708	0.2665	0.5037	0.6624

Answer to RQ1: Our proposed method performs better than the SOTA method in terms of three indicators for detecting multi-types of SATD in issue-tracking systems.

4.3. RQ2: Does Our Method Using Information-Extracted Features Outperform Other Methods in Constructing the SATD Verbalizer?

Motivation: Our method employs the MI method to extract features as keywords to construct the SATD verbalizer, which enhances the diversity of words in the mask position. This RQ investigates whether the MI method can better construct the SATD verbalizer than other methods.

Methods: To answer this question, we have selected three feature-extraction methods for comparative experiments, which are employed to construct the SATD verbalizer, including CHI [55,56], TF_IDF [57,58], and DF [59,60].

- CHI [55,56]: CHI is a hypothesis testing statistics. It assumes that the keyword and the category are directly irrelevant. If the value of the CHI distribution calculated deviates more from the threshold, the original hypothesis is not supported. In other words, the keyword categories have a high degree of relevance.
- TF_IDF [58]: The TF_IDF is used to estimate the degree to which a word is essential to a particular document in the training set. The TF represents the frequency of occurrence

of a word in the text, which is also known as word frequency, and the IDF is called Inverse Document Frequency (IDF). The value of TF_IDF is high which means the word is important to the document.

- DF [60]: The Document Frequency (DF) of a word is the number of documents in which the word is expected to appear. If a word has a low DF, it can be removed from the original feature space. DF not only can reduce the dimensionality of the feature space but also has the potential to improve the accuracy of the classification.

Results: Table 4 represents a comparative performance of our method versus the three comparative methods across the indicators of Precision, Recall, and F1-score, where Average denotes the average value of each method in seven projects. The table reveals that our method outperforms the CHI, DF, and TF_IDF methods in terms of Precision, with mean improvements of 8.04%, 9.81%, and 15.23%, respectively. Our method demonstrated superior performance in six out of the seven projects when compared to the three baseline methods. Our method outperforms the CHI, DF, and TF_IDF methods in terms of Recall, with mean improvements of 8.68%, 5.51%, and 13.80%, respectively. Our method demonstrated superior performance in four out of the seven projects when compared to the three baseline methods. Our method outperforms the CHI, DF, and TF_IDF methods in terms of F1-score, with mean improvements of 8.61%, 5.59%, and 14.49%, respectively. Our method demonstrated superior performance in four out of the seven projects when compared to the three baseline methods.

Table 4. The performance by different feature-extraction methods in terms of Precision, Recall, and F1-score. (The bold part indicates obtaining the highest performance).

Projects	Precision				Recall				F1-Score			
	CHI	**DF**	**TF_IDF**	**MI**	**CHI**	**DF**	**TF_IDF**	**MI**	**CHI**	**DF**	**TF_IDF**	**MI**
Camel	0.5972	0.4822	0.5011	**0.6342**	0.6430	0.5609	0.6070	**0.7099**	0.5886	0.4682	0.4777	**0.6045**
Chromium	0.5083	**0.5461**	0.5150	0.5417	0.5538	**0.6534**	0.5773	0.6306	0.5001	**0.5610**	0.5081	0.5232
Gerrit	0.4994	0.5631	0.4898	**0.6008**	0.5835	0.6241	0.5864	**0.6448**	0.5083	0.5802	0.5235	**0.5954**
Hadoop	0.4655	0.4111	0.3654	**0.4928**	0.5568	0.4729	0.4227	**0.6009**	0.4619	0.4196	0.3421	**0.5106**
Hbase	0.4778	0.4762	0.4244	**0.4891**	0.5425	0.5816	0.5044	0.5568	0.4668	**0.4962**	0.4364	0.4726
Impala	0.5538	0.5627	0.6155	**0.6180**	0.6748	0.7347	**0.6937**	0.6932	0.5552	0.6053	**0.6447**	0.6265
Thrift	0.4900	0.4928	0.4568	**0.5039**	0.5330	0.5826	0.5122	**0.6059**	0.4698	0.5217	0.4359	**0.5233**
Average	0.5131	0.5049	0.4811	**0.5544**	0.5839	0.6014	0.5577	**0.6346**	0.5072	0.5217	0.4812	**0.5509**

Answer to RQ2: MI can extract the features to specific types of SATD, which can extend labels to diversify the words in mask position.The extended labels have a positive impact on the performance of multi-type SATD classification tasks in issue-tracking systems.

4.4. RQ3: Does Our Method Outperform Its Variant Methods?

Motivation: This RQ investigates whether each module in our method is effective in improving the performance of our method.

Methods: To answer this question, we investigated the effectiveness of our method from different perspectives. In particular, we analyzed the effect of each of the four modules used in WPTD, namely the base pre-training model, the SATD verbalizer module, the SATD verbalizer optimization module, and the global feature weights module. Therefore, we examine the effects of our method from different perspectives. We set up four variants of our method as a comparison experiment, i.e., WPTD_br, No_v, No_f, and No_w. WPTD_br employs the PLM Bert to replace the PLM Roberta, No_v removes the SATD verbalizer construction module, No_f removes the SATD verbalizer refinement module, No_w removes the global feature weights module.

Results: Tables 5–7 represent the performance of our method and four variants in terms of Precision, Recall, and F1-score, respectively. From Table 5, we can observe that, in terms

of Precision, the mean of our method obtains better performance than the four variants, and can be improved by 13.51%, 4.67%, 7.91%, and 9.44% compared with WPTD_br, No_v, No_f, and No_w, respectively. Five projects out of seven projects obtained better performance in Precision compared with the four variant methods. From Table 6, we can see that, in terms of Recall, the mean of our method obtains better performance than the four variants, and achieves improvements by 16.11%, 9.85%, 7.17%, and 4.61%. Four projects out of seven projects obtained better performance in Recall compared with the four variant methods. From the Table 7, we can see that, in terms of F1-score, the mean of our method obtains better performance than the four variants, and achieves improvements by 21.86%, 5.62%, 7.74%, and 7.88%. Six projects out of seven projects obtained better performance in the F1-score compared with the four variant methods.

Table 5. The Precision of our method and 4 variant methods. (The bold part indicates obtaining the highest Precision).

Projects	Precision				
	WPTD_br	**No_v**	**No_f**	**No_w**	**WPTD**
Camel	0.5614	0.5943	0.5759	0.5603	**0.6342**
Chromium	0.4183	0.5684	0.4879	0.4962	**0.5417**
Gerrit	0.4353	**0.6055**	0.5291	0.5510	0.6008
Hadoop	0.4553	0.4012	0.4453	0.4576	**0.4928**
Hbase	0.4830	0.4457	0.4502	0.4306	**0.4891**
Impala	0.5846	**0.6233**	0.6155	0.5905	0.6180
Thrift	0.4810	0.4693	0.4923	0.4600	**0.5039**
Average	0.4884	0.5297	0.5138	0.5066	**0.5544**

Table 6. The Recall of our method and 4 variant methods. (The bold part indicates obtaining the highest Recall).

Projects	Recall				
	WPTD_br	**No_v**	**No_f**	**No_w**	**WPTD**
Camel	0.6669	0.6268	0.6631	0.6653	**0.7099**
Chromium	0.5139	0.6197	0.5412	0.6165	**0.6306**
Gerrit	0.5554	**0.6816**	0.6369	0.6249	0.6448
Hadoop	0.5007	0.4629	0.5494	0.4772	**0.6009**
Hbase	0.5088	**0.5599**	0.5309	0.5348	0.5568
Impala	0.5875	0.6141	0.6717	**0.7417**	0.6932
Thrift	0.4927	0.4789	0.5514	0.5861	**0.6059**
Average	0.5466	0.5777	0.5921	0.6066	**0.6346**

Table 7. The F1-score of our method and 4 variant methods. (The bold part indicates obtaining the highest F1-score).

Projects	F1-Score				
	WPTD_br	**No_v**	**No_f**	**No_w**	**WPTD**
Camel	0.5293	0.5931	0.6001	0.5740	**0.6045**
Chromium	0.4227	0.5008	0.4724	0.4773	**0.5232**
Gerrit	0.3791	**0.6266**	0.5381	0.5495	0.5954
Hadoop	0.4308	0.3808	0.4468	0.4253	**0.5106**
Hbase	0.4523	0.4695	0.4551	0.4437	**0.4726**
Impala	0.5407	0.6139	0.5688	0.6222	**0.6265**
Thrift	0.4071	0.4664	0.4980	0.4825	**0.5233**
Average	0.4517	0.5216	0.5113	0.5106	**0.5509**

Compared with WPTD_br, the PLM Roberta more effectively provides tokenizers and MLM models to detect specific types of SATDs in issue-tracking systems. Compared with No_v, the extended labels expand the diversity of words predicted by the mask and boost the predictive performance of our method. The label-refinement module employs the contextualized prior to remove the low-frequency words, the results show that the label-refinement module can cut off the noise words and boost the performance of our method compared with No_f. The global SATD feature weights module enhances the feature representations in detecting specific types of SATD. Thus, this module has a positive impact on our method compared with No_w.

Answer to RQ3: Our method performs better than its variants, and all modules employed in our method have positive impact on its performance.

4.5. RQ4: What Are the Important Features of Each Type of SATD in Issue-Tracking Systems?

Motivation: Developers utilize diverse words to articulate various kinds of SATDs in issue-tracking systems, with the significance of these terms exhibiting considerable variation. This RQ aims to discuss the relationship of keywords between different projects and types so as to further analyze the characteristics of SATD description in the issue-tracking systems.

Methods: To answer this question, we chose the mutual information (MI) method of feature selection and extracted the top 10 keywords for each category in each project for analysis and comparison.

Results: Tables 8–11 present the top 10 keywords of each type. From those tables, we find that the majority of features are similar in each project in each category. For example, in Table 8, the keywords "typo" and "wiki" are contained in seven projects. There are six projects containing the feature word "update" and five projects containing the feature word "performance". This phenomenon demonstrates that if an issue section contains these keywords, its category can be easily recognized. In particular, not all keywords appear on all projects, we only show the top 10 keywords. On some projects, keywords are thus weighted and ranked differently. There is a very small percentage of keywords that are project-specific keywords.

In terms of types, there are some feature words that exist in more than one category. For example, Tables 8 and 9 both contain "typo", "docs", "wiki", and "update", which implies that these keywords are relevant to Code and Design and Documentation debts, but none of the keywords are present in all four tables simultaneously. Anyway, there is a high-ranking keyword in each category. From these tables, we observe that "performance" is a feature word specific to Code and Design, "documentation" is a feature word specific to Documentation debt, "implemented" and "safe" are the specific keywords to Requirement debt, "run" and "test" are the specific keywords to Test debt. The keywords can help developers detect specific types of SATD in issue-tracking systems.

Table 8. Top 10 keywords of Code and Design debt.

Camel	Chromium	Gerrit	Hadoop	Hbase	Impala	Thrift
flaky	typo	typo	wiki	typo	typo	typo
typo	flaky	wiki	doc	wiki	wiki	wiki
docs	docs	unit	coverage	unit	unit	unit
coverage	wiki	update	update	update	update	update
javadoc	update	coverage	javadoc	coverage	javadoc	links
unit	unit	javadoc	performance	links	links	javadoc
links	coverage	links	unit	performance	doc	coverage
wiki	links	performance	implemented	page	performance	documented
document	javadoc	page	remove	implemented	todo	broken
performance	page	remove	use	readme	remove	mention

Table 9. Top 10 keywords of Documentation debt.

Camel	Chromium	Gerrit	Hadoop	Hbase	Impala	Thrift
documentation	documentation	documentation	documentation	documentation	documentation	documentation
typo	typo	typo	docs	docs	typo	typo
docs	docs	docs	wiki	typo	docs	docs
javadoc	wiki	wiki	typo	wiki	wiki	wiki
links	javadoc	javadoc	javadoc	update	javadoc	javadoc
readme	update	update	update	document	update	update
tutorial	page	links	page	links	links	links
guide	links	page	tutorial	page	page	test
update	section	section	website	updated	test	section
section	readme	website	tutorials	readme	website	mention

Table 10. Top 10 keywords of Requirement debt.

Camel	Chromium	Gerrit	Hadoop	Hbase	Impala	Thrift
implemented	safe	safe	implemented	implemented	implemented	implemented
safe	implemented	implemented	todo	safe	safe	safe
todo	thread	thread	safe	thread	todo	todo
safety	todo	todo	thread	todo	thread	keyprovider
thread	keyprovider	keyprovider	reached	reached	keyprovider	yet
keyprovider	android	reached	yet	keyprovider	yet	thread
yet	violation	void	safety	yet	reached	reached
reached	sshd	yet	void	void	void	void
void	discourage	violation	mac	mac	mac	mac
mac	unsafe	rect	android	android	discourage	sshd

Table 11. Top 10 keywords of Test debt.

Camel	Chromium	Gerrit	Hadoop	Hbase	Impala	Thrift
test	test	test	test	test	test	test
flaky	flaky	flaky	flaky	unit	flaky	flaky
coverage	unit	unit	coverage	flaky	unit	unit
unit	coverage	coverage	unit	coverage	coverage	coverage
run	run	run	run	run	run	run
lack	havent	layout	jenkins	havent	lack	testcase
surefire	lack	havent	fail	commented	testcase	layout
automated	add	lack	suite	add	fail	regression
testcase	automated	automated	chance	layout	layout	lack
jenkins	testcase	testcase	integration	complain	complain	surefire

With these keywords, developers can easily determine whether an issue section is an SATD or not, and furthermore, developers can determine which category of SATDs an issue section contains by using the special keywords. We also found that some features can only distinguish whether an issue section contains SATDs or not, but are not effective in determining the category of SATDs.

Answer to RQ4: The features of SATD for different types in different projects help the developers to understand SATDs better.

4.6. RQ5: What Are the Relationships and Differences Between SATDs in Source Code Comments and Issue-Tracking Systems?

Motivation: There are many studies focusing on the identification of SATDs in source code comments. However, we focus on detecting SATDs on issue sections in issue-tracking systems. Investigating the commonalities and differences between the two different sources will help developers better understand SATDs and provide a basis for the identification of SATDs from other sources. Thus, this RQ investigates the relationship and difference between SATDs on source code comments and SATDs in issue-tracking systems.

Methods: To answer this question, we analyzed the differences and relationships of the SATDs in two sources from different perspectives.

Results: We explore the differences in SATDs between the two different sources from the following three perspectives.

The different distribution of debt categories: For example, the percentage of Documentation debt and Test debt are 0.14% and 0.09% in the source code comment dataset, respectively. However, they are 1.77% and 1.44% in issue-tracking systems, respectively. The frequency of these two types of SATDs is notably higher in issue-tracking systems than in source code comments, which indicates that issue-tracking systems tend to record and track these two types of SATDs more than in source code comments. It also shows that the tendencies of the types of SATDs recorded in the two different sources of debt are different. Moreover, these two types of SATDs are not considered in most of the studies based on source code comments due to the low percentage [18,45]. However, we pay attention to these two types of debts and find the uniqueness of these two types of debts.

Describe the different granularity of SATDs: We take the Requirement debt as an example to explain the details. The Requirement debt represents requirements that are not completed by the developer which are defined by [46] in source code comments. Figure 4 shows a source code comment that contains a Requirement debt in the ArgoUML project. From this sentence, we can find that the comment just indicates to the developer that the module needs to be completed, but there is no indication of the status of the work. For example, whether the work has been undertaken or not, and how much of the work has been completed. Next, we take a set of issues in issue-tracking systems as an example. Figure 5 shows the issue CHROMIUM-64779 that the sections contain Requirement debt in comment, description, and summary. In the above two examples, we can observe that the Requirement debt recorded in the code comments tends to be summarized and defined. The issue-tracking system is a more detailed description of the Requirement debt after discussion and analysis by the developers. Therefore, the granularity of the Requirement debt is different in the two systems and also causes their feature variability.

" TODO Needs completing stub implementation only! tfm"

- [Requirement debt of source code comments in ArgoUML]

Figure 4. An example of a source code comment containing a Requirement debt.

Comment: *"It is not possible to implement a cross-platform solution because the delay is unacceptable. So (assuming we want an animation here), I think the current approach is correct -- it just needs to be implemented on os x and linux."*

Description: *"It's possible that we don't want to do anything here, but if so, NOTIMPLEMENTED seems wrong (it shows up in everyone's logs with each app launch). \#if defined(OS_WIN) rect); \#else NOTIMPLEMENTED(); \#endif"*

Summary: *"64779: NOTIMPLEMENTED() in for non-Windows"*

- [Requirement debt of issue tracking systems in CHROMIUM-64779]

Figure 5. An example of an issue containing the Requirement debts.

The different keywords: We summarize the top 10 keywords for each type of SATD between two sources. Table 12 shows the result of MI feature extraction by our method in two scenarios. In particular, since we have not lemmatized the document, lexical differences are not considered in the keywords, i.e., words with different lexical properties from the same root are treated as the same. From this table, we can observe that the word "*document*" is the common keyword for Code and Design debt and Documentation debt in two scenarios, the words "*todo*" and "*implement*" are common keywords of Requirement debt in two scenarios, and the word "*test*" is a common keyword of Test debt in two scenarios. It indicates that although the procedure of steps to record SATDs is different in the two systems, there still exist a few keywords that can be common due to the definition of SATD categories. However, the other keywords are different.

From the above analyses, we can conclude that the SATDs contained in the source code comments and SATDs contained in the issue-tracking system have similarities, but the similarities are limited.

Table 12. Top 10 keywords of each type of SATD in issue-tracking systems and source comments.

C&D		Doc		Req		Tst	
Issue	**Comment**	**Issue**	**Comment**	**Issue**	**Comment**	**Issue**	**Comment**
documentation	todo	documentation	document	implemented	needed	test	test
test	needed	typo	function	safe	todo	flaky	assert
flaky	test	docs	param	thread	should	unit	supported
typo	should	wiki	needs	todo	class	coverage	yet
docs	class	javadoc	string	yet	implement	run	though
wiki	implement	update	spec	safety	used	remove	fails
unit	document	test	javadocs	keyprovider	remove	cover	even
performance	instance	links	knowledge	reached	so	use	asserttrue
update	used	page	please	void	from	add	lead
coverage	yet	readme	object	mac	instance	performance	jan

Answer to RQ5: There is some relation between the SATDs in the source code comments and the SATDs in the issue-tracking system, but the relevance is limited.

5. Discussion

5.1. The Impact of Different Settings

In this section, we discuss the impact of choosing a few shots on the performance of our method and extending the influence of the number of features in the label set on model performance. We only change the parameters to be investigated, ensuring that other parameters remain unchanged.

Few-shot sample: We empirically set up 10 different shots from 1-shot to 10-shot samples and conduct experiments with each set. Some types of SATD have a few numbers of instances and contain a range from 5 instances to 16 instances in Table 1. In order to efficiently obtain features for each type, we chose a range from 1-shot to 10-shot. Table 13 shows the F1-score value of our method in different few-shot settings. From this table, we can see that a low number of samples provides fewer features and shows a negative performance for the model. Thus, rich samples can expand the features and introduce extra noise to the dataset. There are some samples with too few SATDs of some species that can also result in an unbalanced distribution due to the lack of samples. These problems also cause a decrease in model performance. Therefore, we selected eight shots, which enabled our method to achieve better performance.

Table 13. The F1-score in different few-shot samples. (The bold part indicates obtaining the highest F1-score).

Projects	1-Shot	2-Shot	3-Shot	4-Shot	5-Shot	6-Shot	7-Shot	8-Shot	9-Shot	10-Shot
Camel	0.2265	0.2161	0.2912	0.2646	0.3282	0.4535	0.4831	**0.6045**	0.5084	0.5126
Chromium	0.3317	0.4090	0.3669	0.4603	0.4695	0.5031	0.4889	**0.5232**	0.4092	0.5003
Gerrit	0.3790	0.4670	0.4558	0.4680	0.4909	0.5038	0.5063	**0.5954**	0.5064	0.5102
Hadoop	0.2311	0.3471	0.2918	0.3477	0.3213	0.3624	0.4013	**0.5106**	0.4377	0.4108
Hbase	0.4020	0.4098	0.4299	0.4506	0.4329	0.4341	0.4278	0.4726	0.4677	**0.5065**
Impala	0.4470	0.5407	0.4749	0.5297	0.5171	0.5248	0.4696	**0.6265**	0.5572	0.5967
Thrift	0.3164	0.3607	0.4251	0.4620	0.4847	0.4882	0.5090	**0.5233**	0.5183	0.4594
Average	0.3334	0.3929	0.3908	0.4261	0.4349	0.4671	0.4694	**0.5509**	0.4864	0.4995

The number of features: We empirically set up 10 different numbers of features (NF) {10, 20, 30, 40, 50, 60, 70, 80, 90, and 100} and conduct experiments with each set to investigate the impact that the number of features has on our method.

Figure 6 shows the results of each setup on the three indicators, where the vertical axis displays the performance values obtained for the three indicators, and the horizontal axis indicates the quantity of selected features. It can be observed from this figure that our method obtains optimal performance in all three indicators in which the number of selected features is 20. If NF is too small or too large, it can have a negative impact on our method. Too few features do not enhance our model performance, and too many features have too much noise, which negatively affects our model performance. Therefore, we chose the top 20 features as the feature words for expanding the label.

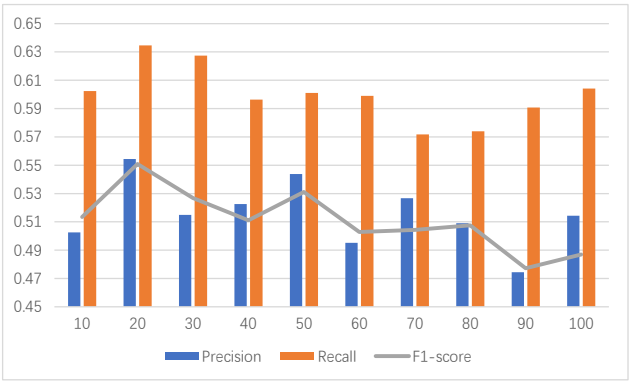


Figure 6. Bar diagram of the performance obtained on our method for different number of features.

5.2. Implications

Our study provides insights for the follow-up study of multi-type SATDs in issue-tracking systems.

Implications to researchers can be provided as follows: We provide a deep learning method that can identify specific types of SATDs by issue-tracking systems automatically. WPTD can assist researchers in the automatic identification of specific types of SATD. Our method employs the few-shot sampling. In the real world, researchers can train our model by inputting a few samples when they find new types or sources of debt. The trained model can be used to automatically identify large amounts of unprocessed text and reduce manual classification. Our results show that there are some similarities between problem-tracking systems and SATDs in code comments, but the similarities are limited. This conclusion can assist researchers in their study of the elimination of technical debt from different sources, and the impact of debt on software in different systems.

Implications for developers. Our approach to identifying SATDs in issue-tracking systems at a more fine-grained level will help developers better understand specific types of SATDs, providing a foundation for future SATD-elimination efforts. The lists of keywords we have developed for each category of SATDs will help developers understand the properties of SATDs, how to better describe SATDs, and how to accurately identify and locate SATD descriptions by others.

5.3. Threats to Validity

Threats to internal validity: Threats to internal validity stem from coding errors in the experiments and human biases in the dataset. WPTD and baseline methods utilize Pytorch and third-party libraries (e.g., sklearn, openPrompt, etc.). We based ours on the code provided by previous studies [34,36] and carefully modified the code to satisfy our needs.To minimize label errors in the dataset due to human bias, we used the highly consistent dataset [11] (with Cohen's kappa coefficient above 0.7).

Threats to external validity: Threats of external validity center on the generalizability of WPTD. We conducted the experiments on a publicly available and reliable dataset

that includes 3000 issue sections from seven projects in two issue-tracking systems, with different contributors, different numbers of issue sections, and different product functions. This is a public dataset that can aid researchers in replicating our results. In addition, we recognize the importance of validating our method on other data sources, which we leave for future work.

Threats to construct validity: Threats to construct validity are associated with the performance indicators we have chosen and their applicability. In this study, we utilized recognized indicators to assess the performance of our model, including Precision, Recall, and F1-score. We adopt the Scott–Knott ESD test for analysis of the method, enhancing the credibility of our evaluation. Furthermore, it concerns the applicability of our model. Our work is designed to ascertain which SATD category an issue section belongs to and to identify SATD types within issue-tracking systems, meeting the need to locate different types of debt for developers. The results demonstrate the superiority of WPTD and indicate that it is highly suitable for such tasks.

6. Conclusions

In this work, we proposed a novel method named WPTD which can detect the specific type of SATD in an issue-tracking system. Initially, WPTD employs the MI feature extraction to choose keywords that construct the SATD verbalizer. Next, the SATD verbalizer is refined by using the prior contextual information, effectively eliminating low-frequency words. Further, WPTD deploys the template to render the instance and utilizes the MLM model of the PLM to adapt the model and predict the probability of the mask position in the SATD verbalizer. Finally, it calculates the average score of the SATD verbalizer in each type of SATD. The highest score in the template is taken as the predicted probability. We conducted experiments using three performance indicators on the dataset, and the results have shown that WPTD outperformed all baseline methods. Looking ahead, we are concerned with collecting and analyzing more SATDs from a variety of sources, not limited to comments, codes, and other forms of data. We also expect to focus on studying the impact of technical debt in conjunction with joint issue-tracking systems, submit systems, code reviews, and other multidimensional data.

Author Contributions: Conceptualization, method, and writing, J.Y.; validation, H.T.; software, R.L.; review and editing, Q.Z. and Y.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China (Grant Nos. 42104072, and 52201363), Research Plan Project of Hubei Provincial Department of Education (Q20232208).

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Cunningham, W. The WyCash portfolio management system. *ACM Sigplan Oops Messenger* **1992**, *4*, 29–30. [CrossRef]
2. Codabux, Z.; Williams, B. Managing technical debt: An industrial case study. In Proceedings of the 4th International Workshop on Managing Technical Debt (MTD), San Francisco, CA, USA, 20 May 2013; pp. 8–15.
3. Xuan, J.; Hu, Y.; Jiang, H. Debt-prone bugs: Technical debt in software maintenance. *arXiv* **2017**, arXiv:1704.04766.
4. Ampatzoglou, A.; Ampatzoglou, A.; Chatzigeorgiou, A.; Avgeriou, P. The financial aspect of managing technical debt: A systematic literature review. *Inf. Softw. Technol.* **2015**, *64*, 52–73. [CrossRef]
5. Wehaibi, S.; Shihab, E.; Guerrouj, L. Examining the impact of self-admitted technical debt on software quality. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 1, pp. 179–188.
6. Bellomo, S.; Nord, R.L.; Ozkaya, I.; Popeck, M. Got technical debt? Surfacing elusive technical debt in issue trackers. In Proceedings of the 13th International Conference on Mining Software Repositories, Austin, TX, USA, 14–22 May 2016; pp. 327–338.

7. Brown, N.; Cai, Y.; Guo, Y.; Kazman, R.; Kim, M.; Kruchten, P.; Lim, E.; MacCormack, A.; Nord, R.; Ozkaya, I.; et al. Managing technical debt in software-reliant systems. In Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, Santa Fe, NM, USA, 7–8 November 2010; pp. 47–52.
8. Tom, E.; Aurum, A.; Vidgen, R. An exploration of technical debt. *J. Syst. Softw.* **2013**, *86*, 1498–1516. [CrossRef]
9. Kamei, Y.; Maldonado, E.d.S.; Shihab, E.; Ubayashi, N. Using Analytics to Quantify Interest of Self-Admitted Technical Debt. In Proceedings of the QuASoQ/TDA@ APSEC, Hamilton, New Zealand, 6–9 December 2016; pp. 68–71.
10. Sierra, G.; Shihab, E.; Kamei, Y. A survey of self-admitted technical debt. *J. Syst. Softw.* **2019**, *152*, 70–82. [CrossRef]
11. Li, Y.; Soliman, M.; Avgeriou, P. Identifying self-admitted technical debt in issue tracking systems using machine learning. *Empir. Softw. Eng.* **2022**, *27*, 131. [CrossRef]
12. Alves, N.S.; Mendes, T.S.; De Mendonça, M.G.; Spínola, R.O.; Shull, F.; Seaman, C. Identification and management of technical debt: A systematic mapping study. *Inf. Softw. Technol.* **2016**, *70*, 100–121. [CrossRef]
13. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. When and why your code starts to smell bad. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 403–414.
14. Potdar, A.; Shihab, E. An exploratory study on self-admitted technical debt. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; pp. 91–100.
15. Li, Y.; Soliman, M.; Avgeriou, P. Identification and remediation of self-admitted technical debt in issue trackers. In Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Kranj, Slovenia, 26–28 August 2020; pp. 495–503.
16. Huang, Q.; Shihab, E.; Xia, X.; Lo, D.; Li, S. Identifying self-admitted technical debt in open source projects using text mining. *Empir. Softw. Eng.* **2018**, *23*, 418–451. [CrossRef]
17. Yu, D.; Wang, L.; Chen, X.; Chen, J. Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt. *Front. Comput. Sci.* **2021**, *15*, 154208. [CrossRef]
18. Chen, X.; Yu, D.; Fan, X.; Wang, L.; Chen, J. Multiclass classification for self-admitted technical debt based on XGBoost. *IEEE Trans. Reliab.* **2021**, *71*, 1309–1324. [CrossRef]
19. Li, Y.; Soliman, M.; Avgeriou, P. Automatic identification of self-admitted technical debt from four different sources. *Empir. Softw. Eng.* **2023**, *28*, 65. [CrossRef]
20. Wang, X.; Liu, J.; Li, L.; Chen, X.; Liu, X.; Wu, H. 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, Melbourne, Australia, 21–25 September 2020; pp. 871–882.
21. Ren, X.; Xing, Z.; Xia, X.; Lo, D.; Wang, X.; Grundy, J. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM Trans. Softw. Eng. Methodol.* **2019**, *28*, 1–45. [CrossRef]
22. Kenton, J.D.M.W.C.; Toutanova, L.K. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the NAACL-HLT, Minneapolis, MN, USA, 2–7 June 2019; Volume 1, p. 2.
23. Xavier, L.; Ferreira, F.; Brito, R.; Valente, M.T. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 25–26 May 2020; pp. 137–146.
24. Wang, C.; Yang, Y.; Gao, C.; Peng, Y.; Zhang, H.; Lyu, M.R. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 382–394.
25. Han, X.; Zhao, W.; Ding, N.; Liu, Z.; Sun, M. Ptr: Prompt tuning with rules for text classification. *AI Open* **2022**, *3*, 182–192. [CrossRef]
26. Schick, T.; Schütze, H. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv* **2020**, arXiv:2001.07676.
27. Liu, X.; Zheng, Y.; Du, Z.; Ding, M.; Qian, Y.; Yang, Z.; Tang, J. GPT understands, too. *arXiv* **2021**, arXiv:2103.10385. [CrossRef]
28. Chen, X.; Xie, X.; Zhang, N.; Yan, J.; Deng, S.; Tan, C.; Huang, F.; Si, L.; Chen, H. Adaprompt: Adaptive prompt-based finetuning for relation extraction. *arXiv* **2021**, arXiv:2104.07650.
29. Jia, M.; Tang, L.; Chen, B.C.; Cardie, C.; Belongie, S.; Hariharan, B.; Lim, S.N. Visual prompt tuning. In Proceedings of the European Conference on Computer Vision, Tel Aviv, Israel, 23–27 October 2022; Springer: Cham, Switzerland, 2022; pp. 709–727.
30. Lester, B.; Al-Rfou, R.; Constant, N. The power of scale for parameter-efficient prompt tuning. *arXiv* **2021**, arXiv:2104.08691.
31. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
32. Schick, T.; Schütze, H. It's not just size that matters: Small language models are also few-shot learners. *arXiv* **2020**, arXiv:2009.07118.
33. Gu, Y.; Han, X.; Liu, Z.; Huang, M. Ppt: Pre-trained prompt tuning for few-shot learning. *arXiv* **2021**, arXiv:2109.04332.
34. Hu, S.; Ding, N.; Wang, H.; Liu, Z.; Wang, J.; Li, J.; Wu, W.; Sun, M. Knowledgeable prompt-tuning: Incorporating knowledge into prompt verbalizer for text classification. *arXiv* **2021**, arXiv:2108.02035.
35. Zhu, J.; Li, L.; Yang, L.; Ma, X.; Zuo, C. Automating method naming with context-aware prompt-tuning. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, Australia, 15–16 May 2023; pp. 203–214.

36. Ding, N.; Hu, S.; Zhao, W.; Chen, Y.; Liu, Z.; Zheng, H.T.; Sun, M. Openprompt: An open-source framework for prompt-learning. *arXiv* **2021**, arXiv:2111.01998.
37. Huang, Y.; McCullagh, P.; Black, N.; Harper, R. Feature selection and classification model construction on type 2 diabetic patients' data. *Artif. Intell. Med.* **2007**, *41*, 251–262. [CrossRef] [PubMed]
38. Dash, M.; Liu, H. Feature selection for classification. *Intell. Data Anal.* **1997**, *1*, 131–156. [CrossRef]
39. Tang, J.; Alelyani, S.; Liu, H. Feature selection for classification: A review. In *Data Classification: Algorithms and Applications*; Chapman and Hall/CRC: Boca Raton, FL, USA, 2014; p. 37. [CrossRef]
40. Estévez, P.A.; Tesmer, M.; Perez, C.A.; Zurada, J.M. Normalized mutual information feature selection. *IEEE Trans. Neural Netw.* **2009**, *20*, 189–201. [CrossRef] [PubMed]
41. Vergara, J.R.; Estévez, P.A. A review of feature selection methods based on mutual information. *Neural Comput. Appl.* **2014**, *24*, 175–186. [CrossRef]
42. Bennasar, M.; Hicks, Y.; Setchi, R. Feature selection using joint mutual information maximisation. *Expert Syst. Appl.* **2015**, *42*, 8520–8532. [CrossRef]
43. Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.* **2023**, *55*, 1–35. [CrossRef]
44. Liu, Z.; Lin, W.; Shi, Y.; Zhao, J. A robustly optimized BERT pre-training approach with post-training. In Proceedings of the China National Conference on Chinese Computational Linguistics, Hohhot, China, 13–15 August 2021; Springer: Cham, Switzerland, 2021; pp. 471–484.
45. Yu, J.; Zhao, K.; Liu, J.; Liu, X.; Xu, Z.; Wang, X. Exploiting gated graph neural network for detecting and explaining self-admitted technical debts. *J. Syst. Softw.* **2022**, *187*, 111219. [CrossRef]
46. da Silva Maldonado, E.; Shihab, E.; Tsantalis, N. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1044–1062. [CrossRef]
47. Chouchen, M.; Ouni, A.; Mkaouer, M.W.; Kula, R.G.; Inoue, K. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Appl. Soft Comput.* **2021**, *100*, 106908. [CrossRef]
48. Saheb-Nassagh, R.; Ashtiani, M.; Minaei-Bidgoli, B. A probabilistic-based approach for automatic identification and refactoring of software code smells. *Appl. Soft Comput.* **2022**, *130*, 109658. [CrossRef]
49. Zhao, K.; Liu, J.; Xu, Z.; Li, L.; Yan, M.; Yu, J.; Zhou, Y. Predicting crash fault residence via simplified deep forest based on a reduced feature set. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021; pp. 242–252.
50. Zhao, K.; Liu, J.; Xu, Z.; Liu, X.; Xue, L.; Xie, Z.; Zhou, Y.; Wang, X. Graph4Web: A relation-aware graph attention network for web service classification. *J. Syst. Softw.* **2022**, *190*, 111324. [CrossRef]
51. Liu, J.; Liu, W.; Allechy, F.B.; Zheng, Z.; Liu, R.; Kouadio, K.L. Machine learning-based techniques for land subsidence simulation in an urban area. *J. Environ. Manag.* **2024**, *352*, 120078. [CrossRef] [PubMed]
52. Aiken, W.; Mvula, P.K.; Branco, P.; Jourdan, G.V.; Sabetzadeh, M.; Viktor, H. Measuring Improvement of F 1-Scores in Detection of Self-Admitted Technical Debt. In Proceedings of the 2023th IEEE International Conference on Technical Debt (TechDebt), Melbourne, Australia, 14–15 May 2023; pp. 37–41.
53. Mikolov, T.; Grave, E.; Bojanowski, P.; Puhrsch, C.; Joulin, A. Advances in pre-training distributed word representations. *arXiv* **2017**, arXiv:1712.09405.
54. Zhang, S.; Zheng, D.; Hu, X.; Yang, M. Bidirectional long short-term memory networks for relation classification. In Proceedings of the 29th Pacific Asia Conference on Language, Information and Computation, Shanghai, China, 30 October–1 November 2015; pp. 73–78.
55. Lancaster, H.O.; Seneta, E. Chi-square distribution. *Encycl. Biostat.* **2005**, *2*. [CrossRef]
56. McHugh, M.L. The chi-square test of independence. *Biochem. Medica* **2013**, *23*, 143–149. [CrossRef]
57. Di Salle, A.; Rota, A.; Nguyen, P.T.; Di Ruscio, D.; Fontana, F.A.; Sala, I. PILOT: Synergy between text processing and neural networks to detect self-admitted technical debt. In Proceedings of the International Conference on Technical Debt, Pittsburgh, PA, USA, 16–18 May 2022; pp. 41–45.
58. Ramos, J. Using tf-idf to determine word relevance in document queries. In Proceedings of the First Instructional Conference on Machine Learning, Los Angeles, CA, USA, 23–24 June 2003; Volume 242, pp. 29–48.
59. Ni, R.; Zhang, X.; Jiang, Y.; Xie, X.; Xue, Y.; Liu, Y. Multi-Level Contextual Analysis for Bug Report Classification. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1137–1155.
60. Joho, H.; Sanderson, M. Document frequency and term specificity. In Proceedings of the Recherche d'Information Assistée par Ordinateur Conference (RIAO), Pittsburgh, PA, USA, 30 May 2007.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.