Leveraging Context Information for Self-Admitted Technical Debt Detection

Miki Yonekura*, Yutaro Kashiwa*, Bin Lin†, Kenji Fujiwara‡, Hajimu Iida*
*Nara Institute of Science and Technology, Japan
†Hangzhou Dianzi University, China
‡Nara Women's University, Japan

***Abstract*—Self-Admitted Technical Debt (SATD) refers to non-optimal software design or implementation that is acknowledged and explicitly documented in the code by developers. Detecting SATD and understanding its evolution can help developers better manage their development activities and monitor the software quality. In recent years, numerous approaches have been proposed to automatically identify SATD. However, these approaches still suffer from a high number of false positives (*i.e.,* non-SATD comments being detected as SATD).**

To further advance this field, in this paper, we conduct an empirical study to evaluate the performance of the state-of-the-art SATD detection tools and investigate the causes behind the false positives. By manually analyzing 135 false positive cases, we identify the main types of comments that are easily misclassified. To address this issue, we propose a new approach, CASTI, which integrates context information into CodeBERT, a pre-trained model for programming languages. Our evaluation demonstrates that CASTI can significantly reduce the false positives and that the context information does help improve the performance.

***Index Terms*—SATD, CodeBERT, Context-Aware Detection**

I. Introduction

In software projects, developers often opt for a temporary and quick solution instead of adopting the best software development practices due to time constraints. The implied costs of reworking the non-optimal source code are often referred to as "technical debt" [1]. Similar to financial debt, technical debt also accumulates "interest" if not repaid early, making the corrections more difficult and expensive over time [2]. In practice, developers sometimes explicitly acknowledge the technical debt in the source code as comments. For example, developers can inform the team that the current implementation is not optimal and that future maintenance will be necessary using certain keywords (*e.g.,* `//TODO:` or `//FIXME:`) in the comments of the source code. This kind of documented technical debt is called Self-Admitted Technical Debt (SATD) [3]. SATD helps clarify the issues that need to be recognized by the development team, making it easier to prioritize and allocate resources.

Recently, several studies have investigated the impact of SATD on code quality and found that code containing SATD is more likely to have lower quality. For example, researchers have found that the files with SATD tend to undergo more bug fixes [4], [5] and be modified more frequently [6] than files without SATD. These results indicate that while temporary quick implementation solutions may speed up the software product delivery at some point, they often come with higher costs for future maintenance and evolution. In practice, SATD becomes increasingly difficult to address over time, as the software system itself becomes more complex. Not handling SATD in a timely manner not only enlarges the maintenance effort but also delays the addition of new features or the evolution of existing functionalities due to potential workarounds needed to accommodate non-optimal code. In fact, studies have found that in projects with a high occurrence of SATD, many developers are reluctant to address SATD even though they are aware of its existence [7]. This situation can lead to a decline in project quality, potentially hindering the project's long-term success.

Given the importance of addressing SATD in a timely manner, many recent studies have proposed new approaches to detect SATD in software projects. The emergence of these techniques has also significantly contributed to software quality-related empirical studies [8]–[16]. Particularly, machine learning-based methods are gaining more and more popularity, leveraging state-of-the-art models such as Convolutional Neural Network (CNN) [14] and Gated Graph Neural Network (GGNN) [15]. While these methods have overall improved the performance of SATD detection, they still produce a considerable number of false positives. We conjecture that the high number of false positives may result from the fact that these approaches only take into account the comments, without considering the relevant source code as contexts. For example, when we see "`//we need to break apart for 1.8 ver`," one might think that this is a SATD comment. However, when checking the source code (as shown in Snippet 1), it is not difficult to conclude that this comment only explains the rationale behind the `if` condition, which should be classified as a non-SATD comment.

To verify our assumption and understand why state-of-the-art approaches produce false positives, we first conducted an empirical study to evaluate the performance of existing approaches and manually analyzed what types of non-SATD are more likely to be misclassified as SATD. We then proposed a novel SATD detection approach, CASTI, leveraging context information. More specifically, we fed the comments and their context (the source code around them) into the CodeBERT language model. Our evaluation shows that CASTI can achieve better precision than state-of-the-art approaches while maintaining a high recall. Moreover, the context information plays a critical role in reducing the occurrences of false positives.

Snippet 1: Example of False Positive Detections

```
// we need to break apart for 1.8 ver.
if (lex_state == EXPR_CMDARG) {
  result = tLPAREN_ARG;
} else if (lex_state == EXPR_ARG) {
  result = tLPAREN2;
}
```

The main contributions of this study are as follows:

1) **We compared the performance of state-of-the-art approaches using the same datasets:** Existing SATD detectors are claimed to have a high accuracy in the original studies. While the evaluations are often conducted on the dataset by Maldonado *et al.* [11], they do not use a consistent approach (*e.g.,* using different subsets of the dataset). Therefore, a fair comparison of these tools on the exact same dataset is needed. Our study fills this gap and reveals their performance on a large dataset.
2) **Our study revealed which types of non-SATD are often misclassified as SATD:** We manually inspected the false positives generated by state-of-the-art approaches and our approach. The summarized categories indicate the research direction for further improving SATD detectors. We also analyzed the prevalence of each type of false positives.
3) **We proposed a context-aware SATD detection approach:** Our approach takes into account the context information around comments and can significantly reduce the false positives while maintaining a high recall. We also demonstrated which context information is more beneficial for the performance improvement.

The remainder of this paper is structured as follows. Section II introduces the related work. Section III presents an empirical study which compares the performance of state-of-the-art SATD detection approaches and analyzes the causes of false positives. Section IV proposes Context-Aware Self-admitted Technical Debt Identifier (CASTI), a new approach for SATD detection leveraging context information. The methodology and the results for evaluating CASTI are also presented in this section. Section V discusses threats to validity, and Section VI concludes this paper.

Replication Package: To facilitate replication and further studies, we provide the data used in our replication package.[1]

II. Related Work

In this section, we introduce existing SATD detection techniques, categorized as pattern-based, machine learning-based, and deep learning-based.

A. Pattern-Based SATD Detection

Potdar and Shihab [3] first coined the term SATD. They manually inspected approximately 100,000 comments collected from four repositories and identified 63 text patterns that indicate the presence of SATD. For example, phrases like `need to` and `should be` are typically included in SATD comments, indicating the need for future revisions of the source code by oneself or others. Based on this study, Bavota *et al.* [17] leveraged these patterns to identify SATDs in other repositories and classify the types of SATDs.

Guo *et al.* [10] proposed a fuzzy matching approach to identify SATDs, which uses only typical SATD patterns such as `TODO`, `FIXME`, `XXX`, and `HACK`. Like other pattern-based approaches, this approach does not require any training data to build a prediction model. Nevertheless, their empirical evaluation using four open-source software projects shows a higher precision than simple machine-learning approaches, which require a time-consuming and resource-intensive manual labeling process to construct large datasets.

B. Machine Learning-Based SATD Detection

The pattern-based approaches typically use regular expressions that need to be pre-defined. While these approaches can often achieve a high precision, they might not capture most of the SATD comments (*i.e.,* having a low recall and missing numerous SATDs not matched with regular expressions) [13]. To address this issue, many machine learning-based approaches have been proposed and become the mainstream.

Maldonado *et al.* [11] proposed an NLP-based machine learning approach to automatically identify two common types of SATD: design debt and requirement debt. To reduce the noise, several filtering heuristics are applied to remove the comments which are less likely to contain SATD, such as license comments and Javadoc comments. Their model employs a maximum entropy classifier, which not only automatically extracts the most important features but also identifies the features negatively contributing to the classification results. To evaluate the approach, they collected a dataset of 62,566 comments and manually categorized them into five types of SATD. Their empirical evaluation of two types of SATD (design and requirement SATD) shows that their approach significantly outperforms the pattern-based approach [3].

Huang *et al.* [12] proposed a novel text mining-based approach for classifying comments as SATD or non-SATD. This approach employs a composite classifier of multiple sub-classifiers that are built from different repositories. A voting strategy [18] is used to determine the final label. Each sub-classifier is trained with Naïve Bayes Multinomial. Their empirical evaluation using the dataset created by Maldonado *et al.* [9] demonstrates that their approach increases the *F1-value* over the pattern-based approach [3] and the NLP-based approach [11] by 499.19% and 27.95%, respectively.

Sala *et al.* [16] proposed a new approach, DebtHunter, that applies two-fold predictions. It first performs binary classification (*i.e.,* SATD or not) and then categorizes the type of SATD. DebtHunter employs Sequential Minimal Optimization (SMO) [19] for both binary and multi-classification. Their evaluation also uses the dataset by Maldonado *et al.* and their approach achieves a *precision* of 0.892, a *recall* of 0.751, and an *F1-value* of 0.816.

[1] https://github.com/mikiyonekura/CASTI-Replication

C. Deep Learning-Based Detection

Given the emergence of advanced deep learning techniques in recent years, many studies have proposed deep learning-based SATD detection approaches.

Ren *et al.* [14] developed an approach using convolutional neural networks (CNNs) to extract important features representing the characteristics of SATD comments. Specifically, they used Word2Vec to generate embedding vectors. Their experiments on the dataset by Maldonado *et al.* [11] exhibit a *precision* of 0.669, a *recall* of 0.887, and a *F1-value* of 0.752. Similarly, Wang *et al.* [20] also proposed a CNN-based model but leveraged an attention mechanism. Their evaluation using the same dataset [11] shows that their new model achieves a *F1-value* 9.14% higher than the basic CNN model [14].

Salle *et al.* [21] proposed PILOT, a technical debt detector built on natural language processing and deep learning techniques. More specifically, the approach employs TF-IDF and word embedding techniques like Word2Vec to gain a deeper understanding of the semantics of comments and generate high-dimensional feature vectors. PILOT also uses a feedforward neural network (FFNN) as the classifier, which excels in processing high-dimensional data and improving the detection accuracy of SATD comments thanks to its simple structure and effective learning capabilities. Their empirical evaluations on the dataset by Maldonado *et al.* [11] demonstrate that PILOT outperforms DebtHunter, a traditional machine-learning approach, in detecting SATD comments.

In more recent years, several studies have adopted language models for SATD detection. Prenner and Robbes [22] evaluated the effectiveness of several models based on pre-trained Transformers for SATD identification. The evaluation using the Maldonado dataset [11] showed that a BERT-based model achieved an average *F1-value* of 0.821, significantly surpassing the 0.766 achieved by the aforementioned CNN model [20]. Similarly, Sabbah and Hanani [23] explored the SATD identification performance of several word embedding methods using different pre-trained language models (Word2Vec, GloVe, BERT) and different classifiers (SVM, RF, Naive Bayes, and CNN). Their evaluation using various datasets shows that the CNN classifier with BERT achieved the highest performance in the dataset they created, while Random Forest with TF-IDF had the best performance on the Maldonado *et al.*'s dataset.

These existing approaches have explored various models and feature selection techniques to improve SATD detection performance. However, these approaches typically only use comments as input. As described before, to precisely identify SATDs, even developers have to understand the context of the source code around the comments. Therefore, providing a certain amount of code to models might help improve the performance. As the closest work to ours, Sheikhaei *et al.* [24] very recently explored the effectiveness of Large Language Models in identifying and categorizing SATDs. Their experiments using Maldonado *et al.*'s dataset demonstrate that all fine-tuned LLMs can improve *F1-value* by 4.4% to 7.2% compared to the state-of-the-art non-LLM baselines for SATD identification tasks. More importantly, they investigated the effectiveness of incorporating contextual information *to categorize the types of SATDs* (*e.g.,* defects, design, tests). Their experiment showed that providing file names, method bodies, and comments in the prompts improves the performance of classification tasks. Despite the similarity, our study differs in two aspects. First, their work uses context information for SATD type classification, while our study focuses on the more fundamental problem: identifying whether a comment is SATD or not. It is unclear whether providing contexts can help with this task. Second, we explore different types of context information and analyze how different granularities of contexts will impact the performance.

III. Evaluating State-of-the-Art SATD Detectors

While emerging SATD detectors have demonstrated a high accuracy, there is currently no clear winner. Although most approaches were evaluated with the Maldonado dataset [11], the testing sets are not exactly the same: some approaches used the whole dataset (*e.g.,* [14], [22]), while others removed duplicates and undersampled the dataset to ensure balanced numbers of SATD and non-SATD comments [16], [21]. To understand the real performance of state-of-the-art SATD detectors, in this study, we use the same dataset to evaluate two approaches which are not directly compared and achieved better performance than other compared techniques: PILOT based on a traditional deep-learning model [21] and BERT+CNN based on a language model [23].

A. Research Questions

To evaluate the state-of-the-art approaches, we propose the following research questions (RQs):

RQ1: How do state-of-the-art approaches perform in SATD identification? This RQ aims to perform a fair performance comparison of the state-of-the-art SATD detectors.

RQ2: What are the main sources of false positives? This RQ aims to understand the main causes of false positives produced by the state-of-the-art approaches. We focus on false positives because, in practice, low precision introduces a lot of noise and puts extra burden on developers, significantly diminishing the practical value of such tools. The insights gained from this RQ can help further enhance SATD detectors.

B. Study Design

1) Dataset: In this study, we adopt the dataset by Maldonado *et al.* [11], which is the most commonly used in previous studies. Maldonado *et al.*'s dataset was collected from ten open-source projects from different application domains and of varying sizes. A team of experts with extensive experience as software engineers manually annotated whether a comment is SATD or not. Note that SATD is rarely contained in Javadoc and license comments. Maldonado *et al.* thus excluded all these comments except for those that clearly include SATD, such as comments containing task annotation tags (*e.g.,* `TODO`). Table I presents the detailed information of the dataset.

TABLE I: Details of the Maldonado Dataset

Repository	Release	#Comments	#SATD	% SATD
Ant	1.7.0	4,137	131	0.60
ArgoUML	0.34	9,548	1,413	2.08
Columba	1.4	6,478	204	0.60
EMF	2.4.1	4,401	104	0.41
Hibernate	3.3.2	2,968	472	4.05
JEdit	4.2	10,322	256	1.50
JFreeChart	1.0.19	4,423	209	0.89
JMeter	2.10	8,162	374	1.86
JRuby	1.4.0	4,897	662	5.57
SQuirrel	3.0.3	7,230	285	1.04
Average	-	6,257	411	1.86
Total	-	62,566	4,110	-

As one of our ultimate goals is to examine whether context information can help reduce false positives and the original dataset does not include the source code around comments, we extract the code ourselves. To do so, we employ the Source Code-Comments Comment-Context Miner (SoCCMiner) [25], which receives a file path and automatically identifies comments and the relevant source code. It is worth noting that the tool can also identify the part of the snippet related to the given comment. That is, no matter if a comment is associated with a class, method, block, or line, it can provide the corresponding source code. Below we provide an example of each type of comment and its "relevant code".

Line comments: This type of comments is relevant to the source code in the next line. Snippet 2 depicts an example of single-line comments describing the behavior of the next line.

Snippet 2: Example of single-line comments [26]

```
// add ant properties
allProps.putAll(getProperties());
```

Inline comments: This type of comments function similarly as single-line comments, and the only difference is that it is written right after the code in the same line, which is often short. Snippet 3 is an example of such an inline comment. The comment introduces what the declared variable is used for.

Snippet 3: Example of inline-line comments [27]

```
File remoteFile; // use for interface
```

Block comments: This type of comments is relevant to multiple lines of code (*i.e.,* block). They are often used for conditional statements (*e.g.,* if), loops (*e.g.,* for, while), try-catch blocks, or highly connected statements. Snippet 4 presents an example of block comments, noting that the loop outside will be broken when there is a non-ASCII character.

Snippet 4: Example of block comments [28]

```
// if it's not an ASCII, break here
if (ch >= 128) {
    break;
}
```

Method comments: This type of comment is located above the method and explains the purpose and functionality of the method. Snippet 5 shows an example of method comments. This comment explains how the method can be used.

Snippet 5: Example of method comments [29]

```
// Get the jar files in .ant/lib
private URL[] getUserURLs() {
    File LibDir = new File(HOMEDIR);
    return getLocationURLs(LibDir);
}
```

Class comments: Snippet 6 shows an example where the relevant code is a class. Class comments usually explain the purpose and functionality of the class, including a brief description of major methods and fields.

Snippet 6: Example of class comments [30]

```
// A class used to parse the output.
class ChangeLogParser {
    private static int GET_FILE = 1;
    public Entry[] getEntrySet() {
        Entry[] array = new Entry[];
        return array;
    }
}
```

SoCCMiner takes given file names as input and returns a set of a comment and relevant code. To provide inputs for SoCCMiner, we first extract the names of the files that contain the comments included in Maldonado *et al.*'s dataset. We then match the comments in the output of SoCCMiner with the comments in Maldonado *et al.*'s dataset. After mapping, we obtain a set of comments, relevant code, and corresponding label (*i.e.,* SATD or not). Note that there are a non-negligible number of cases where the comments in Maldonado *et al.*'s dataset and the comments returned by SoCCMiner cannot be matched. This is because the comments in Maldonado *et al.*'s dataset were modified to remove redundant spaces, line breaks, etc [31]. After removing the unmatched comments, 72.7% of the comments (*i.e.,* 45,498 comments) in Maldonado *et al.*'s dataset remain. Moreover, we removed duplicate comments following the approach in the previous study [16], treating comments with identical text as duplicates regardless of whether their corresponding source code was identical or not. As a result, the final set consisted of 25,835 comments, which represents 41.3% of the original dataset. This step is crucial to ensure that the same comments are not included in both the training and the testing sets.

Table II summarizes the number of comments that are successfully matched, the number of SATD comments, and the ratio of SATD comments. The numbers in brackets in the second and third columns indicate the percentage of the matched comments in each repository of the original dataset.

2) Data Collection and Analysis: To evaluate the performance of state-of-the-art SATD detectors (**RQ1**), we split the dataset and use 60% as training data, 20% as validation data, and the remaining 20% as testing data. The following four performance metrics are used in our evaluation:

TABLE II: Details of the studied dataset.

Repository	#Comments	#SATD	% SATD
Ant	2,181 (52.7%)	57 (43.5%)	2.61
ArgoUML	3,874 (40.6%)	591 (41.8%)	15.26
Columba	3,299 (50.9%)	87 (42.6%)	2.64
EMF	1,214 (27.6%)	32 (30.8%)	2.64
Hibernate	1,780 (60.0%)	244 (51.7%)	13.71
JEdit	2,007 (19.4%)	115 (44.9%)	5.73
JFreeChart	1,980 (45.0%)	60 (28.7%)	3.03
JMeter	2,926 (35.8%)	195 (52.1%)	6.66
JRuby	3,072 (62.7%)	259 (39.1%)	8.43
SQuirrel	3,502 (48.4%)	104 (36.5%)	2.97
Average	2,584 (41.3%)	174 (42.4%)	6.37
Total	25,835 (41.3%)	1,744 (42.4%)	-

- *Precision:* The proportion of comments predicted as SATD that are actually SATD.
- *Recall:* The proportion of actual SATD comments that are predicted as SATD.
- *F1 score:* The harmonic mean of Precision and Recall. Since there is a trade-off between Precision and Recall, the F1 score evaluates the balance between Precision and Recall. In other words, it assesses whether the increase in Precision (or Recall) outweighs the decrease in Recall (or Precision).
- *ROC-AUC:* ROC-AUC stands for the Area Under the Receiver Operating Characteristic Curve. The ROC curve plots the True Positive Rate (TPR) and False Positive Rate (FPR) obtained when changing the threshold used for SATD classification. ROC-AUC ranges from 0 to 1, with a higher value indicating better model performance in discriminating between SATD and non-SATD across all classification thresholds.

To investigate what kinds of non-SATDs are more likely to be misclassified as SATD (**RQ2**), we adopted a card sorting approach [32] to manually inspect and categorize the false positives produced by PILOT and BERT+CNN. More specifically, two of the authors independently examine each false positive and determine the purposes of the misclassified non-SATD comment. It is worth noting that inspectors are also asked to check the source code around these comments. During the annotation, each new label becomes available for reuse to obtain consistent annotation and avoid unnecessary highly similar labels. Conflicts between the two inspectors are resolved by a third inspector, who discusses the annotation with the first two inspectors until an agreement is reached. Note that the three inspectors have programming experience ranging from 7 to 15 years.

C. Results

TABLE III: Comparison of existing methods

Method	Precision	Recall	F1	ROC-AUC	FP	FN
PILOT	**0.793**	0.838	0.813	0.976	**75**	55
BERT+CNN	0.762	**0.888**	**0.820**	**0.985**	95	**38**

1) RQ1: How do state-of-the-art approaches perform in SATD identification?: Table III presents the performance metrics of PILOT and BERT+CNN.

As can be seen in Table III, PILOT achieves a *precision* of 0.793, higher than that of BERT+CNN (0.762). On the other hand, BERT+CNN has a higher *recall* of 0.888, compared to 0.838 for PILOT. BERT+CNN also obtains higher *F1-value* and ROC-AUC. That is, BERT+CNN outperforms PILOT in all metrics except *precision*. While a precision of around 80% is not low, the number of false positives is still not negligible.

Previous studies [33], [34] show that software development support tools tend not to be adopted when they exhibit a high false positive rate. In particular, for SATD detection, the large volume of comments imposes a significant burden on developers due to high false positive rates. This study thus prioritizes precision over recall. While increasing both precision and recall is optimal, even improving the precision without reducing the recall can already bring significant benefits.

RQ1: *BERT+CNN outperforms PILOT in terms of recall and ROC-AUC. However, PILOT has a higher precision, which produces fewer false positives.*

2) RQ2: What are the main sources of false positives?: To understand what kinds of non-SATD comments are more likely to be misclassified as SATD, we manually inspected false positives produced by PILOT and BERT+CNN. More specifically, we examined 75 false positive (FP) comments from PILOT and 95 from BERT+CNN (Table III). In total, we inspected 135 comments due to the overlaps between these two groups. The manual annotation by the two of the authors reached agreement in 90.6% of the cases, with Cohen's kappa coefficient being 0.871 (indicating substantial agreement).

Table IV shows the manual agreement results, listing the types of false positives and their numbers of occurrences. The 135 false positive comments can be categorized into seven different types (while in reality two of them are not real false positives). Below we illustrate each type of false positives.

TABLE IV: Classification results of FP comments

Type	PILOT	BERT+CNN
Labeling error	42 (56.0%)	35 (36.8%)
Implementation feedback request	1 (1.3%)	19 (20.0%)
Rationale behind implementation	10 (13.3%)	17 (17.9%)
Code behavior explanation	19 (25.3%)	11 (11.6%)
Limitations/issues of implementation	1 (1.3%)	5 (5.3%)
Indication of unexpected scenarios	1 (1.3%)	5 (5.3%)
Implementation update note	1 (1.3%)	3 (3.2%)
Total	75	95

*Cases not in bold are not real false positives.

Labeling errors: This category refers to comments labeled as non-SATD in the Maldonado dataset but are actually SATD. That is, this category does not represent real false positives. In our manual inspection, "Labeling Errors" is the most common source of "false positives", accounting for 37.8% (*i.e.,* 51/135) of the instances. An example of "Labeling Errors" can be seen

in Snippet 7. In fact, we found that many comments including explicit keywords such as "//TODO:" and "//FIXME:" are still labeled as non-SATD. These cases should be included as true positives. Some other studies [4], [31] also mentioned this issue, indicating that a more carefully refined dataset would be appreciated to advance this field of research.

Snippet 7: Example of Labeling error [35]

```
private void explain(ActionEvent e){
  //TODO: make a new history item
  ToDoList list =
  ↪  Designer.theDesigner().getToDoList();
  ...
```

Implementation feedback request: This category refers to comments that ask other developers for feedback on the implementation. Developers sometimes use comments as a communication channel to discuss the feasibility of implementations or ask questions when they do not understand the source code. For example, the comment in Snippet 8 asks for feedback on whether the implementation is correct. In fact, these comments are also not real false positives, as they do require developers to respond later in some way. A previous study [36] also observed these cases and classified them as SATD. As can be seen in Table IV, PILOT is less likely to classify these comments as SATD, indicating that it is less sensitive to detecting such cases.

Snippet 8: Example of Implementation feedback request [37]

```
private static boolean isPrimitive
↪  (String typeDescriptor){
  return typeDescriptor.length()== 1; //
  ↪  right?
}
```

Rationale behind implementation: This category refers to comments explaining the background or reasons for implementing the source code in the current way. For instance, the comment in Snippet 9 shows that the `if` condition is created to deal with a bug related to `jdk1.4.2`. This is the most prevalent category for real false positives, accounting for 17.0% (*i.e.,* 23/135) of the instances.

Snippet 9: Example of Rationale behind implementation [38]

```
//deal with jdk1.4.2 bug:
if (ex != null) {
  if (results.indexOf("zip file
  ↪  closed")>= 0) {
    log("You are running " +
    ↪  JARSIGNER_COMMAND + " against a
    ↪  JVM with" + " a known bug that
    ↪  manifests as an
    ↪  IllegalStateException.",
    ↪  Project.MSG_WARN);
  } else {
    throw ex;
  }
}
```

Code behavior explanation: This category refers to comments that explain what the source code is supposed to do. These comments help enhance the readability of code, which is essential for program comprehension. Snippet 10 shows an example comment explaining that the code is supposed to connect to a remote server. This type of false positive is also quite common in our study, with slightly fewer occurrences than "rationale behind implementation," and accounting for 19.3% (*i.e.,* 26/135) of the cases.

Snippet 10: Example of Code behavior explanation [39]

```
//connect to the remote site
connection.connect();
```

Limitations/issues of implementation: This category refers to comments that warn about the limitations or issues of the current implementation. It often notes the special conditions of using certain functions or the potential side effect of the code. Snippet 11 shows a comment that explicitly indicates the potential recursive calls caused by the proxy initialization.

Snippet 11: Example of Limitations of implementation [40]

```
boolean hasNoQueuedAdds =
↪  lce.getCollection().endRead(); //
↪  warning: can cause a recursive calls!
↪  (proxy initialization)
```

Indication of unexpected scenarios: This category refers to comments that declare impractical, unlikely, or undesired scenarios. Snippet 12 illustrates a comment indicating a condition that should never happen, which contains an exception handling.

Snippet 12: Example of Indication of unexpected scenarios [41]

```
if (retry > MAX_CONN_RETRIES) {
    // This should never happen
    throw new BindException();
}
```

Implementation update note: This category refers to comments which describe the updates applied to the original source code. An example in Snippet 13 explains that the implementation in the block has been moved from another location in the program.

Snippet 13: Example of Implementation update note [42]

```
if ( element.isFetch() ) {
  // This is now handled earlier.
  if ( element.getQuery != null ) {
  }
}
```

RQ2: *There are five categories of comments that are inappropriately classified as SATD. The two most common categories are "Rationale behind implementation" and "Code behavior explanation". The high number of incorrect "false positives" reaffirms the need for extra caution when comparing the performance of SATD detectors and a more carefully curated dataset.*

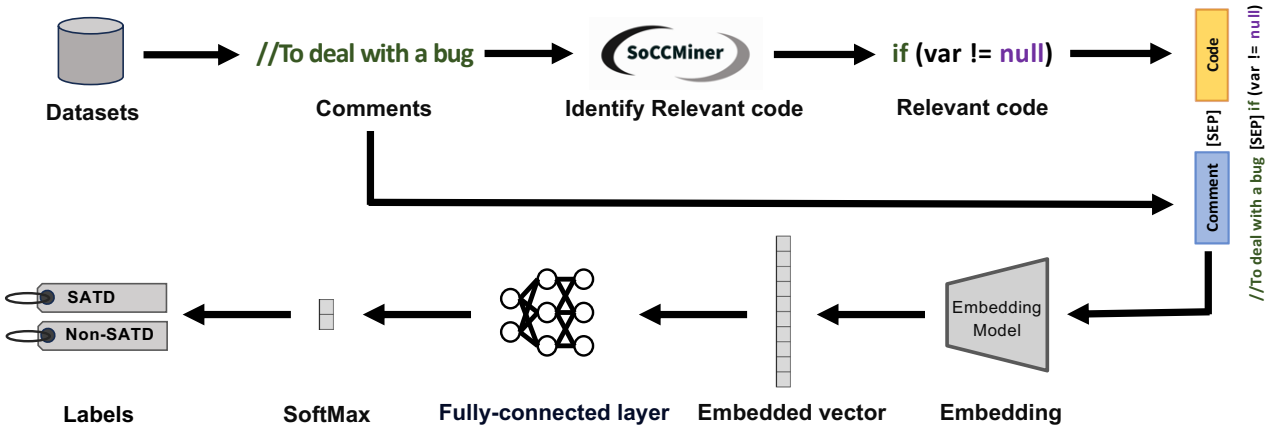

Fig. 1: Overview process of CASTI

IV. Context-Aware Self-admitted Technical Debt Identifier (CASTI)

In this section, we propose Context-Aware Self-admitted Technical Debt Identifier (CASTI), a novel model leveraging CodeBERT [43] and context information for identifying SATD comments.

A. Methodology

Figure 1 depicts the overall architecture of CASTI. CASTI takes comments and relevant code as input, transforms them into feature vectors, and returns a binary value (*i.e.,* SATD or not) via the fully connected layers. The details of each process are described as follows.

1) Input Preparation: As discussed in Section I, even developers sometimes need to check the source code to decide whether a comment is SATD or not. Existing machine-learning approaches usually receive only the comments and classify the comment based on word frequency (*i.e.,* ignoring the context), resulting in many false positives. In recent years, modern language models (*e.g.,* CodeBERT [43] and CodeT5 [44]) have been demonstrated the strong ability to take into consideration the context information. Therefore, to create our own language model to identify SATD, we will use both comments and their contexts as input.

As presented in Section III-B1, we consider five different types of comments (single-line comments, inline comments, block comments, method comments, and class comments). SoCCMiner is used to retrieve their corresponding their "relevant codes". These comments, together with the relevant code, then go through the embedding process to be converted into feature vectors.

2) Embedding: We use the pre-trained model CodeBERT [43] to generate embeddings. CodeBERT is a bimodal model that uses two types of data: natural language (NL) and six programming languages (Python, Java, JavaScript, PHP, Ruby, Go). It excels in capturing the similarity between text and code, enabling tasks such as code summarization and code search by natural language texts. Since our approach also requires capturing the relation between comments and relevant code, CodeBERT is considered as a suitable model for embedding generation.

In our approach, the CodeBERT model receives strings that contain comments and their relevant code, divided by a separator "[SEP]", which allows CodeBERT to recognize that the input has two segments (*i.e.,* comments and source code) and improve the performance [43]. These strings are then transformed into integer values using a standardized tokenizer, BertTokenizer, which tokenizes the source code using a pre-trained vocabulary file. Particularly, uncommon words are divided into subwords using the WordPiece algorithm [45]. This algorithm, widely used in language models such as BERT, first creates a vocabulary that includes all characters, then selects the most frequently occurring character combinations in the training data. For example, the word "TestCase" is split into "Test" and "##Case", where the special symbol "##" indicates that it is a subword token (*i.e.,* part of a longer word). If a token is not found in the pre-trained vocabulary file, the unknown token ⟨UNK⟩ is used.

BertTokenizer then maps each token, including subword tokens, out-of-vocabulary tokens (⟨UNK⟩), and other special tokens (⟨s⟩) and ⟨/s⟩), to integers specified in the vocabulary file. The generated list of integers is padded with 0 at the front and back to ensure all lists have the same length. In this study, as we need to handle data of a long length such as classes, we set the limit of token size to 512 to ensure the data variability. Finally, the tokenized data are fed into the encoder, and embedding vectors are returned by CodeBERT.

Note that we do not perform any preprocessing for the texts of comments such as stopword removal, stemming, and lemmatization. This is because the BERT architecture directly uses the input words for masking and predicts the original words from their context. The original paper [46] also recommends using raw texts. In fact, we did not observe any significant differences in our preliminary experimentation with a preprocessing step.

3) Model Training: To optimize the model performance, several studies [14], [47] have used the cross-entropy loss function to adjust model weights during training. Specifically, a weighted cross-entropy loss is robust to imbalanced datasets, and thus it is adopted in our study. The loss function is defined as follows:

$$L(y, \hat{y}) = -\left(w_1 y \log(\hat{y}) + w_0 (1-y) \log(1-\hat{y})\right)$$

Here, y represents the correct class (0 representing non-SATD or 1 representing SATD), and $\hat{y}$ represents the predicted probability by the model. Additionally, w_0 and w_1 represent the weights for class 0 and class 1, respectively. The weights for each class are defined as inversely proportional to the size of the class:

$$w_0 = \frac{1}{n_0}, \quad w_1 = \frac{1}{n_1}$$

where n_0 and n_1 represent the sizes of class 0 and class 1, respectively. We also adopt early stopping to prevent overfitting. Specifically, the training process stops if the evaluation loss does not improve for five consecutive epochs.

4) SATD Prediction: To predict whether a comment contains SATD, we provide the fine-tuned model with that comment and its relevant code. Similarly, these inputs are processed by BertTokenizer and transformed into feature vectors. The fine-tuned model returns the likelihood of the class being SATD via the fully connected layer and the softmax function. The prediction result is represented as a floating-point number between 0 and 1.

B. Research Questions

To evaluate the performance of CASTI, we propose the following two research questions (RQs):

RQ3: How does CASTI perform compared to state-of-the-art approaches? This RQ aims to compare the performance of CASTI with state-of-the-art SATD detection tools. More specifically, we are interested in whether providing the context of the source code reduces the number of false positives.

RQ4: How does the context information impact the performance? This RQ aims to investigate the impact of the amount of context information provided on the model performance in SATD prediction.

C. Study Design

1) Dataset and metrics: To ensure a fair comparison, we adopt the same dataset (25,835 comments with 1,744 containing SATD) and metrics (Precision, Recall, F1-score, and ROC-AUC) used for answering RQ1 and RQ2. Note that we fix the incorrect false positive cases in the dataset identified by our previous manual analysis before running the experiments.
2) Data Collection and Analysis: In this study, we use the pre-trained CodeBERT model provided by Huggingface[2] as the encoder for embedding training instances. This pre-trained model provides a powerful context-aware data representation. To optimize our model during training, we use an optimizer of AdamW [48] with a maximum of 10 epochs, a batch size of 16, 500 warm-up steps, a learning rate of 5e-6, and a weight decay of 0.01. While a larger batch size could lead to faster convergence, it may require more memory, which can be constrained by our computational environment. Therefore, we use a batch size of 16 as a feasible setting for the training.

To answer RQ3, we run CASTI on the dataset and comapre the metrics with those obtained in RQ1.

To answer RQ4, we consider different sizes of the context length. While CASTI feeds the comments together with the detected relevant code to the model, it is unclear how different sizes of context information could impact the performance. Therefore, we change the size of the code provided as relevant code (only for class, method, and block) in our experiment. The rationale behind this choice is that inline comments and single-line comments have a very clear scope of code they are associated with. However, the class, method, and block comments, the comment of SATD might only be relevant to part of the code. Specifically, we create eight models that are trained with eight different sizes of relevant code including a maximum of 0, 1, 2, 4, 8, 16, 32, and 64 lines of code, extracted from the class, method, and blocks (Note that the lines we extract from classes, methods, and blocks, never surpass the size of themselves, *i.e.,* the extracted lines $\leqq$ the size of relevant code). We compare the performance of these eight models with CASTI.
3) Running Environment: The proposed model and the previous approaches are trained on machines with 2 Xeon Gold 6230R 26 core processors, 2 NVIDIA A100 GPUs, and 256GB memory. The training process took an average of 2 hours and 5 minutes for our proposed approach CASTI.

D. Results

1) RQ3: How does CASTI perform compared to state-of-the-art approaches?: Table V shows the performance of each approach to identify SATD comments. Note that as we have corrected the misclassified false positive cases identified in our previous RQs, the metric values of PILOT and BERT+CNN are slightly different from Table III.

When comparing the existing approaches with CASTI in terms of *precision*, CASTI achieves a value which is 4.1% and 5.9% higher than PILOT and BERT+CNN, respectively. The number of false positives is thus the lowest (*i.e.,* 19) among the three approaches, indicating that leveraging context information and CodeBERT can indeed reduce the false positives.

However, when it comes to *recall*, BERT+CNN shows a 2.2% higher recall than CASTI. As there is a trade-off between *precision* and *recall*, we then compare them in terms of *F1-value*, and *ROC-AUC*. For both metrics, CASTI has the highest performance (3.7% and 1.4% higher in *F1-value*, 1.5% and 0.9% higher in *ROC-AUC*, compared to PILOT and BERT+CNN, respectively).

We argue that in such an application, *precision* outweighs *recall*, as false positives will waste developers' effort and hinder their willingness to use these tools. At the same time,

[2]https://huggingface.co/microsoft/codebert-base

TABLE V: Comparison with existing methods after filtering (RQ3)

Method	Precision	Recall	F1	ROC-AUC	FP	FN
PILOT	0.899	0.838	0.867	0.971	32	55
BERT+CNN	0.884	**0.880**	0.887	0.977	41	**38**
CASTI	**0.936**	0.861	**0.899**	**0.986**	**19**	47

*The corrections for the false positives found by our manual inspection are applied to this table. Therefore, the *precision* and *F1-value* in this table are higher than those in Table III.

TABLE VI: Distribution of the causes of false positives produced by PILOT, BERT+CNN, and CASTI

Causes	PILOT (A)	BERT+CNN (B)	CASTI (C)	$A \cap C$	$B \cap C$	$A \cap B$	$A \cap B \cap C$
Labeling error	42	35	29	9	3	24	6
Implementation feedback request	1	19	5	0	0	0	0
Rationale behind implementation	10	17	**8**	1	0	4	2
Code behavior explanation	19	11	**9**	0	1	3	0
Limitations/issues of implementation	1	5	**1**	0	0	0	0
Indication of unexpected scenarios	1	5	**0**	0	0	1	0
Implementation update note	1	3	**1**	0	0	0	0
Total	75	95	53	10	4	32	8
(Real False Positives)	32	41	**19**	1	1	8	2

while the *recall* of CASTI is not the highest, it is still fairly comparable with BERT+CNN, which achieves the best recall.

It is worth noting that when comparing the results of the very latest paper [24] that leverages a T5-based model and evaluated the performance on Maldonado *et al.*'s dataset, *F1-value* of CASTI (*i.e.,* 0.853 without fixing the labeling errors) is higher than those of all variations in T5-based models (*i.e.,* 0.818–0.839). Our future work will include comparing more recent approaches with the same dataset settings.

We also manually investigated what types of false positives are produced by CASTI using the same annotation process as RQ2. Table VI shows the number of each type of false positives generated by the existing methods and by CASTI. CASTI produced 53 comments that were determined to be false positives. As can be seen, 34 comments are "Labeling error" and "Implementation feedback request", which are not false positives in reality. Therefore, CASTI only produces 19 false positives in total, which is the lowest among all three approaches (PILOT: 32, BERT+CNN: 41). Regarding the real false positives, the number associated with "Rationale behind implementation" is dramatically reduced by CASTI.

Snippet 14: Example of an case where the existing methods cannot correctly classify while CASTI can [49]

```
if (account == null) {
    /* this should not happen
       templates seem to be missing */
    text = "Account templates missing"
    throw new RuntimeException(text);
}
```

Snippets 1 and 14 illustrate two cases where the existing approach cannot, but CASTI can classify correctly. Both comments contain words typically used in SATD descriptions such as "should be" and "need to". However, by looking at the common false positive cases among these approaches (the last four columns in Table VI), most of the false positives produced by these three approaches are not identical. Our future work will investigate why these differences occur and explore the effectiveness of an ensemble approach using CASTI and other state-of-the-art approaches.

RQ3: *CASTI outperforms the existing approaches with respect to all the performance measures except recall. Notably, CASTI can correctly identify comments that explain the rationale behind implementation as non-SATD.*

2) RQ4: How does the context information impact the performance?: Figure 2 shows the performance of the models with different sizes of the relevant code. Note that 0 lines of code means that the model does not use class, method, and block code but uses inline and one-line comments. This approach helps to understand whether even a tiny bit of context for comments associated with multiple lines of code can also improve the performance.

When looking into *ROC-AUC*, the performance is the highest when only using one line of code as the context. However, the differences are subtle, with a *ROC-AUC* ranging from 0.984 to 0.989. On the other hand, *F1-value* dynamically varies across different settings. Comparing no context with one line of relevant code, using one line of code does help achieve better *F1-value*. After that, the performance decreases with the growing size of lines used until 32 lines are used. The *F1-value* increases again and using the maximum lines of relevant code achieves the highest score.

Interestingly, in terms of *precision*, while the trend is similar to *F1-value*, the highest *precision* is achieved when using one line of code. This suggests that too much information sometimes introduces noise to the model and has a negative impact on the SATD detection performance.

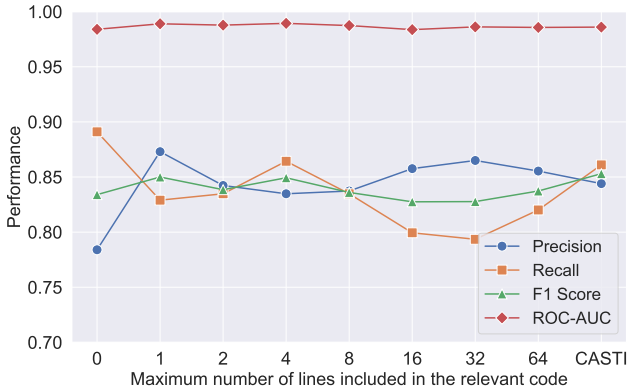

Fig. 2: Performance when using different sizes of classes, methods, and blocks of code

Takeaways: *To ensure accurate predictions (i.e., increasing precision) and save computational resources for training, using one line of relevant code with comments is more recommended than using all the relevant code. Whereas, if the tool users need a generally higher performance of the tools, they are suggested to use the model trained with the whole relevant code because it is when the most of performance metrics are the best.*

RQ4: *Training with the appropriate size of the source code as context information does improve the performance of SATD detection. However, using one line of code can already achieve very good performance, especially in terms of precision.*

V. Threats to Validity

Threats to internal validity concern the factors we did not consider that might impact the results. This study, especially RQ2 and RQ3, relies on the annotation of the authors, which is always a subjective process. To mitigate the potential bias, we examined SATD comments independently and discussed the comments that had conflicting annotations until an agreement was reached by all annotators.

Threats to construct validity concern the relation between theory and observation. Our evaluation employs Maldonado et al.'s dataset [11] that is also created through manual labeling. However, Section III shows that their dataset contains some labeling errors. As the number of labeled instances is more than 70,000, rechecking all the instances in the dataset takes tremendous time. Therefore, we do not relabel them in the same manner as previous studies. The misclassification introduced by the dataset may lead to slightly different results, while keeping the same trend.

Moreover, while our discussion favors precision rather than recall, we acknowledge that the priority might be different for different teams. For example, when a development team has abundant human resources and aims to remove all SATD in the code, they might also pay a significant amount of attention to the recall. However, we believe that for the majority of developers equipped with numerous development assistants already, presenting too many false positives is a show-stopper for adopting an extra tool.

Threats to external validity concern the generalizability of our findings. This study uses the most popular dataset created by Maldonado *et al.*. The SATD in the dataset is collected from 10 repositories. While the dataset contains a significant number of SATD comments, the number of repositories used is rather limited.

Additionally, our evaluation shows that using an appropriate size of relevant code improves the performance of identifying SATD. However, we have not confirmed it with other state-of-the-art SATD detectors or other language models, such as T5-based approaches [24]. We assume that there will be a similar conclusion when we use different context information, as CodeBERT is already quite good at capturing the relation between code and natural language comments. The importance of correct context can thus also be reflected in more advanced language models as well. However, this should be further verified with new experiments.

VI. Conclusions and Future Work

Given the popularity of addressing SATD in software development, many approaches have been proposed to detect SATD. In this study, we first conducted an empirical study to evaluate the performance of two state-of-the-art SATD detection approaches and analyzed what types of non-SATD are more likely to be misclassified as SATD. We then proposed our own approach, CASTI, leveraging CodeBERT and context information, with the aim of reducing false positives. Our evaluation demonstrated that CASTI can significantly improve the precision, and the amount of context information provided does impact the performance of the prediction models.

Our future work is three-fold. First, we would like to build a more reliable dataset and compare the performance of more state-of-the-art approaches, including large language models. Second, we will examine whether an ensemble approach adopting a voting strategy could further improve the SATD identification performance, given the fact that the false positives produced by different approaches are very different. Third, we would like to investigate methods to automatically narrow down the scope of relevant code. We believe that using precisely relevant code without extra unnecessary information might further enhance the SATD identification ability of the language models.

Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP21H03416, JP24K02921, JP24K02923), the Bilateral Program grant (JPJSBP120239929), as well as JST for the PRESTO grant (JPMJPR22P3), the ASPIRE grant (JPMJAP2415), and the AIP Accelerated Program (JPMJCR25U7).

REFERENCES

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

[2] Y. Kamei, E. da S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt," in *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ) and 1st International Workshop on Technical Debt Analytics (TDA)*, vol. 1771, 2016, pp. 68–71.

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

[4] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, and Y. Zhou, "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.

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

[6] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proceedings of the 2016 IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 179–188.

[7] F. Zampetti, G. Fucci, A. Serebrenik, and M. D. Penta, "Self-admitted technical debt practices: a comparison between industry and open-source," *Empirical Software Engineering (EMSE)*, vol. 26, no. 6, p. 131, 2021.

[8] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 2016 IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 315–326.

[9] E. da S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proceedings of the 7th IEEE International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 9–15.

[10] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, and B. Xu, "Mat: A simple yet strong baseline for identifying self-admitted technical debt," *arXiv preprint arXiv:1910.13238*, 2019.

[11] 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 (TSE)*, vol. 43, no. 11, pp. 1044–1062, 2017.

[12] 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 (EMSE)*, vol. 23, no. 1, pp. 418–451, 2018.

[13] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Satd detector: A text-mining-based self-admitted technical debt detection tool," in *Proceedings of the 40th International Conference on Software Engineering (ICSE): Companion Proceeedings*, 2018, pp. 9–12.

[14] 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.

[15] J. Yu, K. Zhao, J. Liu, X. Liu, Z. Xu, and X. Wang, "Exploiting gated graph neural network for detecting and explaining self-admitted technical debts," *Journal of Systems and Software (JSS)*, vol. 187, p. 111219, 2022.

[16] I. Sala, A. Tommasel, and F. A. Fontana, "Debthunter: A machine learning-based approach for detecting self-admitted technical debt," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2021, pp. 278–283.

[17] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 2016, pp. 315–326.

[18] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[19] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods*, pp. 185–208, 1999.

[20] 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 (ASE)*, 2020, pp. 871–882.

[21] A. D. Salle, A. Rota, P. T. Nguyen, D. D. Ruscio, F. A. Fontana, and I. Sala, "Pilot: Synergy between text processing and neural networks to detect self-admitted technical debt," in *Proceedings of the 2022 International Conference on Technical Debt (TechDebt)*, 2022, pp. 41–45.

[22] J. A. Prenner and R. Robbes, "Making the most of small software engineering datasets with modern machine learning," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 12, pp. 5050–5067, 2022.

[23] A. F. Sabbah and A. A. Hanani, "Self-admitted technical debt classification using natural language processing word embeddings," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 13, no. 2, pp. 2142–2155, 2023.

[24] M. S. Sheikhaei, Y. Tian, S. Wang, and B. Xu, "An empirical study on the effectiveness of large language models for SATD identification and classification," *Empirical Software Engineering*, vol. 29, no. 6, p. 159, 2024.

[25] M. Sridharan, M. Mantylä, M. Claes, and L. Rantala, "Soccminer: a source code-comments and comment-context miner," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 242–246.

[26] apache, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/taskdefs/optional/EchoProperties.java#L274, October 30th,2007, accessed: November 1st, 2024.

[27] ——, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/taskdefs/optional/ejb/IPlanetEjbc.java#L1104, October 30th,2007, accessed: November 1st, 2024.

[28] ——, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/launch/Locator.java#L275, October 30th,2007, accessed: November 1st, 2024.

[29] ——, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/launch/Launcher.java#L355, October 30th,2007, accessed: November 1st, 2024.

[30] ——, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/taskdefs/cvslib/ChangeLogParser.java#L36, October 30th,2007, accessed: November 1st, 2024.

[31] M. Sridharan, L. Rantala, and M. Mäntylä, "PENTACET data - 23 million contextual code comments and 250,000 SATD comments," in *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2023, pp. 412–416.

[32] J. R. Wood and L. E. Wood, "Card sorting: current practices and beyond," *Journal of Usability Studies*, vol. 4, no. 1, pp. 1–6, 2008.

[33] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 332–343.

[34] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[35] argouml-tigris org, "argouml," https://github.com/argouml-tigris-org/argouml/blob/20de5bdccc18e361010cb22aea5d79d1b047b1a8/src/argouml-app/src/org/argouml/cognitive/ui/DismissToDoItemDialog.java#L244, Janually 13rd, 2013, accessed: November 1st, 2024.

[36] Y. Kashiwa, R. Nishikawa, Y. Kamei, M. Kondo, E. Shihab, R. Sato, and N. Ubayashi, "An empirical study on self-admitted technical debt in modern code review," *Information and Software Technology*, vol. 146, p. 106855, 2022.

[37] albfan, "jedit," https://github.com/albfan/jEdit/blob/79057da41de68aeb94c503fe22b0a972b3eb3697/bsh/ClassGeneratorUtil.java#L1068, August 29th, 2004, accessed: November 1st, 2024.

[38] apache, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/taskdefs/VerifyJar.java#L181, October 30th,2007, accessed: November 1st, 2024.

[39] ——, "ant," https://github.com/apache/ant/blob/375b5132adfb66953cc698aedebb9b356c1ca180/src/main/org/apache/tools/ant/taskdefs/Get.java#L788, October 30th, 2007, accessed: November 1st, 2024.

[40] hibernate, "hibernate-orm," https://github.com/hibernate/hibernate-orm/blob/0dcbf6df712af5cdbb7d2767c140ddb000d44403/core/src/main/java/org/hibernate/engine/loading/CollectionLoadContext.java#L260, Janually 24th, 2009, accessed: November 1st, 2024.

[41] apache, "jmeter," https://github.com/apache/jmeter/blob/c6184b5e7ffeb56364ed43074504f9539a3b9f01/src/protocol/http/org/apache/jmeter/protocol/http/sampler/HTTPJavaImpl.java#L507, October 21th, 2013, accessed: November 1st, 2024.

[42] hibernate, "hibernate-orm," https://github.com/hibernate/hibernate-orm/blob/0dcbf6df712af5cdbb7d2767c140ddb000d44403/core/src/main/java/org/hibernate/loader/hql/QueryLoader.java#L186, Janually 24th, 2009, accessed: November 1st, 2024.

[43] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[44] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 8696–8708.

[45] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv preprint*, 2016.

[46] 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.

[47] D. Yu, L. Wang, X. Chen, and J. Chen, "Using bilstm with attention mechanism to automatically detect self-admitted technical debt," *Frontiers of Computer Science*, vol. 15, no. 4, p. 154208, 2021.

[48] A. Kumar, R. Shen, S. Bubeck, and S. Gunasekar, "How to fine-tune vision models with SGD," in *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[49] basmilius, "Columba," https://github.com/zieglerm/columbamail/blob/5d8b48043a99e77d529cb2a5b0cca3113b8755d7/columba/trunk/mail/src/main/java/org/columba/mail/gui/config/accountwizard/AccountCreator.java#L50, Janually 5th, 2007, accessed: November 1st, 2024.