How Far Have We Progressed in Identifying Self-admitted Technical Debts? A Comprehensive Empirical Study

ZHAOQIANG GUO and SHIRAN LIU, Nanjing University
JINPING LIU, Jiangsu University
YANHUI LI, LIN CHEN, HONGMIN LU, and YUMING ZHOU, Nanjing University

Background. Self-admitted technical debt (SATD) is a special kind of technical debt that is intentionally introduced and remarked by code comments. Those technical debts reduce the quality of software and increase the cost of subsequent software maintenance. Therefore, it is necessary to find out and resolve these debts in time. Recently, many automatic approaches have been proposed to identify SATD. **Problem.** Popular IDEs support a number of predefined task annotation tags for indicating SATD in comments, which have been used in many projects. However, such clear prior knowledge is neglected by existing SATD identification approaches when identifying SATD. **Objective.** We aim to investigate how far we have really progressed in the field of SATD identification by comparing existing approaches with a simple approach that leverages the predefined task tags to identify SATD. **Method.** We first propose a simple heuristic approach that fuzzily Matches task Annotation Tags (*MAT*) in comments to identify SATD. In nature, *MAT* is an unsupervised approach, which does not need any data to train a prediction model and has a good understandability. Then, we examine the real progress in SATD identification by comparing *MAT* against existing approaches. **Result.** The experimental results reveal that: (1) *MAT* has a similar or even superior performance for SATD identification compared with existing approaches, regardless of whether non-effort-aware or effort-aware evaluation indicators are considered; (2) the SATDs (or non-SATDs) correctly identified by existing approaches are highly overlapped with those identified by *MAT*; and (3) supervised approaches misclassify many SATDs marked with task tags as non-SATDs, which can be easily corrected by their combinations with *MAT*. **Conclusion.** It appears that the problem of SATD identification has been (unintentionally) complicated by our community, i.e., the real progress in SATD comments identification is not being achieved as it might have been envisaged. We hence suggest that, when many task tags are used in the comments of a target project, future SATD identification studies should use *MAT* as an easy-to-implement baseline to demonstrate the usefulness of any newly proposed approach.

CCS Concepts: • **Software and its engineering → Software maintenance tools**;

Additional Key Words and Phrases: Self-admitted technical debt, task annotation tag, code comment, match, baseline

This work is partially supported by the National Key Basic Research and Development Program of China (2014CB340702) and the National Natural Science Foundation of China (61772259, 61872177).

Authors' addresses: Z. Guo, S. Liu, Y. Li (corresponding author), L. Chen (corresponding author), H. Lu, and Y. Zhou (corresponding author), State Key Laboratory for Novel Software Technology, Nanjing University; emails: {gzq, shiranliu}@smail.nju.edu.cn, {yanhuili, lchen, hmlu, zhouyuming}@nju.edu.cn; J. Liu, School of Computer Science and Communication Engineering, Jiangsu University; email: 553439465@qq.com.

1049-331X/2021/07-ART45 $15.00
https://doi.org/10.1145/3447247

ACM Reference format:
Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. 2021. How Far Have We Progressed in Identifying Self-admitted Technical Debts? A Comprehensive Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 45 (July 2021), 56 pages.
https://doi.org/10.1145/3447247

1 INTRODUCTION

Technical debt (TD) is a useful metaphor proposed by Cunningham [11] to describe the situation that developers neglect the maintenance of long-term code quality to achieve short-term goals [12, 13, 15, 16, 19, 20, 23–26, 29–32, 45]. For a software project, writing high-quality and well-structured code is the initial goal. In the process of software evolution, however, uncertainties (e.g., restricted human resources, release time pressure, and cost reduction) [15, 16] often arise to interrupt the scheduled development plans. To cope with these changes and complete the previously designated tasks timely, developers are forced to adjust delivery projects with sub-optimal selection (e.g., hard-coded parameters and functional reduction) [17]. This solution temporarily solves the problem, but in the long term, it results in a chaotic structure and increases the effort of refactoring code [14, 30].

In particular, there is a kind of technical debt that is intentionally introduced (e.g., temporarily modifying a parameter) by developers and marked in code comments. Many studies have shown the key role of code comments in ensuring the quality of software artifacts [27, 28, 34–41]. Therefore, Potdar et al. first investigated technical debts in the perspective of code comments and called them **self-admitted technical debt (SATD)** [14]. Their study indicated that SATD was common and might bring a negative impact to software maintenance. Also, Wehaibi et al. [15] conducted an empirical study to examine the relationship between SATD and software quality. Their findings showed that SATD not only might lead to software defects but also might make the software system more difficult to change in the future. Therefore, there is an urgent need to identify SATD and fix them timely.

In the past decade, a variety of approaches have been proposed to identify SATD. Potdar et al. [14] studied SATD according to code comments and made the first attempt to identify the SATD comments by summarizing 62 SATD comment patterns (i.e., keywords and phrases) manually. Subsequently, Bavota et al. [16] reported that the above 62 patterns could identify SATDs for other projects with a high precision. However, such a *Pattern* approach heavily depends on the manually identified SATD patterns. Furthermore, in practice, the *Pattern* approach may have a low recall, since such patterns, manually summarized from a limited number of projects, cannot represent all possible patterns in real world projects [10]. To tackle these problems, many supervised learning techniques are introduced to automatically learn features in comments to identify SATD, including Maldonado et al.'s ***NLP* (natural language processing)** approach [17], Huang et al.'s ***TM* (text-mining)** approach [10], and Ren et al.'s ***CNN* (convolutional neural network)** approach [49]. In particular, it has been reported that these supervised approaches produce a promising SATD identification performance.

Yet, all the existing studies neglect the fact that, in a large amount of real projects, many comments have already been marked with various task annotation tags, predefined either by popular **IDEs (integrated development environments)** or by developers, to indicate SATD. Clearly, these predefined task annotation tags are prior strong SATD indicators, even if their misuses are considered. Intuitively, if we **Match task Annotation Tags (*MAT*)** in comments to identify SATD, a high identification accuracy should be achieved. Unlike the *Pattern* approach, *MAT* does not need to manually inspect comments to summarize SATD patterns. Unlike the existing supervised approaches, *MAT* does not require any labeled training data to build a model. In particular, *MAT* has a low computation cost and is easy to implement. In contrast, due to the use of complex

modeling techniques, many existing supervised SATD identification approaches not only occupy a high computation cost but also involve a large number of parameters needed to be carefully tuned. This brings non-negligible obstacles for practitioners to apply them in practice. Given this situation, an interesting question hence naturally arises: "How do the existing SATD identification approaches perform compared with *MAT*?" The answer to this question will inform how far we have really progressed in the field of SATD identification. This answer is important for both practitioners and researchers. For practitioners, it will help judge if it is worth to apply the existing SATD identification approaches in practice. If *MAT* performs similarly or even better, then there is no practical reason to apply the complex approaches. For researchers, if *MAT* performs similarly or even better, then there is a strong need to improve the prediction accuracy of the existing SATD identification approaches. Otherwise, the motivation of applying those SATD identification approaches could not be well justified.

In this study, we attempt to investigate how far we have really progressed in the field of SATD identification by comparing existing approaches with a simple approach, *MAT,* that leverages the predefined task tags to identify SATD. More specifically, we want to know: (1) Is the difference in SATD identification performance significant and important?; (2) Do they lead to the identification of substantially different SATDs or non-SATDs?; and (3) What are the weaknesses of existing supervised SATD identification approaches and can they be improved by incorporating the idea of *MAT*? The investigation of the first question does not need the source code of a compared existing approach, as its performance has been reported in the original SATD identification study. However, we need the source code of a compared existing approach to investigate the second and third questions. In our study, we take the following two measures to ensure a fair comparison. On the one hand, for those SATD identification approaches whose source codes are open source, we use the source codes provided by their original authors to conduct the experiment. On the other hand, for those SATD identification approaches whose source codes are not publicly available, we tried our best to re-implement them after carefully reading the corresponding research papers and communicating with the original authors. If there is a large difference in performance between the original and re-implemented approaches, then we will use the performance values reported in the original SATD identification studies to investigate only the first question. If the re-implemented approach exhibits a competitive or even better performance, then we will use the re-implemented approach to investigate the first to third questions. These measures ensure that we can draw a reliable conclusion on the benefits of existing SATD identification approaches w.r.t. *MAT* in practice.

Under the above experimental settings, we conduct a comprehensive comparison between the existing SATD identification approaches and *MAT*. Surprisingly, our experimental results show: (1) *MAT* has a similar or even superior performance for SATD identification compared with the existing approaches, regardless of whether non-effort-aware or effort-aware evaluation indicators are considered; (2) the SATDs (or non-SATDs) correctly identified by the existing approaches are highly overlapped with those identified by *MAT*; and (3) supervised approaches misclassify many SATDs marked with task tags as non-SATDs, which can be easily corrected by their combinations with *MAT*. The above results indicate that, for those projects that use task tags to indicate SATD, the task of SATD identification has in fact been complicated. Consequently, for practitioners, it would be better to apply *MAT* rather than the existing approaches to identify SATDs in a target project. This is especially true when considering the application cost (e.g., model building cost). More importantly, the results reveal that, the current progress in SATD identification studies is not being achieved as it might have been envisaged. Therefore, we strongly recommend that future SATD identification studies should consider *MAT* as a baseline approach to be compared against. As stated in Reference [63], there are two-fold benefits when using a baseline approach.

On the one hand, this would ensure researchers compare and assess the predictive power of a newly proposed SATD identification approach more adequately. On the other hand, *"the ongoing use of a baseline model in the literature would give a single point of comparison."* This will lead to a meaningful assessment of any new SATD identification approach against previous SATD identification approaches.

In summary, in this article, we make the following contributions:

(1) We collect a new SATD comment dataset from 10 popular Java projects, which allows for other researchers to conduct more comprehensive studies in the field of SATD identification. To ensure that other researchers can use this dataset, we make it publicly available [80].
(2) We propose a simple heuristic approach *MAT* to identify SATD without involving any training data. In particular, *MAT* is very easy to understand and apply in real-world projects.
(3) We conduct a comprehensive experimental comparison between existing approaches and *MAT* to investigate the real progress in SATD identification. We are surprised to find that the real progress in SATD identification is not being achieved as it might have been envisaged, regardless of whether the identification performance or the difference in the identified true positive (negative) instances is considered.
(4) We analyze the weaknesses of existing supervised SATD identification approaches and explore how to improve their effectiveness by combining them with *MAT*.
(5) We provide the source code for this study, which can be easily used in future SATD identification studies. This will facilitate researchers to use *MAT* as a baseline SATD identification approach, thus helping develop really effective approaches.

The rest of this articleis organized as follows: Section 2 introduces the background on self-admitted technical debt identification. Section 3 presents the process of constructing a simple baseline approach. Section 4 describes the experimental setup, and Section 5 reports the experimental results. We discuss additional results in Section 6. In Section 7, we summarize the implications in our study. Section 8 analyzes the threats to the validity of our study. Section 9 concludes the articleand outlines the direction for our future work.

2 SELF-ADMITTED TECHNICAL DEBT IDENTIFICATION

Before we start to conduct our empirical study, it is necessary to outline the cognitive status (i.e., background) for the field of **self-admitted technical debt (SATD)** identification in this section. More specifically, we first describe the problem that SATD identification aims to address. This subsection sheds light on the essential mechanism of popular SATD identification approaches. Then, we give a literature overview of current valuable progress in this area. Finally, we summarize the main challenges that may have negative influences on the application of these existing SATD identification approaches and hope that these challenges can attract the attention of our community.

2.1 Problem Statement

The purpose of SATD identification is to address the accuracy of classification results for code comments [10, 17, 49, 81–83, 95]. To predict the labels of comments in a target project, it is common to use comment sentences and their labels from other projects to construct a classification model and then apply the model to predict the labels in a target project. In other words, it is a two-phase process, including model constructing phase and model prediction phase. At the model constructing phase, the comments are first extracted from training projects using code parser tools such as JDeodorant [48]. Then, the real labels of these comments are marked by manually reading the

sentences. After that, these labeled comments data are leveraged to construct a classification model that can capture the features of SATD comments. At the model prediction phase, the comments in a target project are extracted first in the same way. Then, the classification model can output a predicted label (SATD or non-SATD) for each target comment. Thus, there are two sets of labels (real labels and predicted labels) for comments in the target project. After that, the classification performance is evaluated by comparing differences between the two sets of labels. According to the above process, the task of self-admitted technical debt identification is a typical binary classification problem.

2.2 State of Progress

The concept of self-admitted technical debt was first introduced by Potdar et al. [14]. Subsequently, many automatic approaches were proposed for identifying SATD comments. In this section, we introduce the representative approaches to illustrate the state of research progress in this field.

Pattern matching-based approach (Pattern, 2014). Potdar et al. first studied SATD according to code comments. They summarized SATD comment patterns manually and proposed a pattern matching approach (*Pattern* for short) to identify SATD [14]. More specifically, they manually read through 101,762 source code comments and summarized 62 SATD patterns in four projects (i.e., Eclipse, Chromiun OS, ArgoUML, and Apache). Every pattern is a keyword (e.g., "stupid") or a phrase (e.g., "get rid of this") that frequently appears in SATD comments. The rationale of their approach is that a comment is considered to indicate SATD if one of the patterns appears in the target comment. According to the rationale, *Pattern* is an unsupervised approach that can be used directly to identify SATD. According to Huang et al.'s [10] experimental result on eight open-source projects, *Pattern* achieved an excellent performance in precision (0.770 on average) while it had a very low recall. This means that the *Pattern* can only identify a very small portion of SATD.

Natural language processing-based approach (NLP, 2017). To overcome the limitations of *Pattern*, Maldonado et al. [17] proposed an approach based on natural language processing (*NLP* for short) to automatically identify design and requirement SATD comments. In their experiment, they collected datasets consisting of the comments from 10 open-source Java projects. For the datasets, they filtered out the comments that were less likely to be classified as self-admitted technical debt by applying heuristics. After that, they manually classified the label (i.e., SATD or non-SATD) of each comment. Based on the datasets, they used a Java implementation of a maximum entropy classifier, Stanford Classifier [60], to identify SATD comments. In particular, for the prediction of comments in each target project, they used the data of comments and labels from other projects as the training data. Their results showed that the *NLP* approach achieved a good performance even with a relatively small training dataset when identifying SATD comments.

Text mining-based approach (TM, 2018). Huang et al. [10] proposed a text-mining-based supervised approach to automatically identify SATD comments. Assume that there are n source projects and one target project, *TM* consists of the following two phases: model building phase and model prediction phase. At the former phase, it built a sub-classifier (**Naïve Bayes Multinomial (NBM)** [21] model by default) based on the comments of each source project. At the latter phase, it used a vote strategy to composite n sub-classifiers to jointly predict the label (SATD or non-SATD) of an unknown comment of the target project. To this end, source code comments were first preprocessed using a general *NLP* process (i.e., tokenization, stop-word removal, and stemming) to get the set of features (i.e., tokens). After that, it used **Information Gain (IG)** feature selection technology [43] to select a useful subset of features to avoid a curse-of-dimensionality problem. Then, it converted the features into a **Vector Space Model (VSM)** model [33]. Huang et al. used eight open-source projects to evaluate *TM*. Their experimental results showed that the proposed *TM* was superior to *Pattern*.

Convolutional neural network-based approach (CNN, 2019). Ren et al. noticed that the prediction results of *TM* are hard to explain, since the features (i.e., tokens) *TM* used are less intuitive than human-summarized patterns [49]. Therefore, they proposed a convolutional neural network–based approach. In their approach, they first learned a domain-specific word-embedding [51] using the skip-gram model [50] of word2vec from comments. In the process, each word was represented by a vector containing semantic information. Then, they concatenated all vectors of the words in a comment to a matrix as the input data of the *CNN* model. They used the data from source projects to train the classification model and used the model to predict the labels (i.e., SATD or non-SATD) of comments in a target project. According to their experimental results, *CNN* outperforms *Pattern, NLP*, and *TM* in identifying SATD comments. In addition to a good classification performance, their approach can exploit the computational structure of *CNN* to automatically identify key phrases and patterns in code comments that are most relevant to SATD. This mechanism can reduce the effort of manually summarizing patterns and extract a large number of useful patterns. On the whole, *CNN* is the state-of-the-art approach that works well in performance, generalizability, and explainability.

A two-step approach (Jitterbug , 2020). Recently, Yu et al. [95] distinguished SATD comments to two types: (1) **"easy to find" SATDs**; and (2) **"hard to find" SATDs**. The former contains keywords occurring frequently in SATD comments such as "*fixme*" and "*todo*" that are almost always related to SATDs. Therefore, "easy to find" SATDs can be identified easily by matching patterns. The latter does not contain keywords that strongly indicate SATDs. For example, "*Modify the system class loader instead—horrible! But it works!*" According to Yu et al.'s opinion, "hard to find" SATDs can only be accurately identified by human experts. As such, they proposed a two-step framework called *Jitterbug* to identify "*Easy*" and "Hard" SATD comments separately. For a given target project, *Jitterbug* first leveraged a pattern recognizer to extract the patterns from the comments in the training projects and used them to identify the "*Easy*" SATD comments. Then, *Jitterbug* filtered out the comments with patterns in the training and test projects and leveraged the comments data whose labels were available to train (for the first time) or retrain a supervised model. This supervised model recommended top 10 SATD comment candidates to human experts for manually labelling. This process was iterated until the evaluated recall of SATD had achieved an expected recall (say, 90%). In this way, *Jitterbug* can identify "*Easy*" SATDs with a high precision and "Hard" SATDs with a high recall.

2.3 Challenges

In the literature, it has been highlighted that there are two major challenges one SATD identification approach has to deal with [10, 17, 49].

(1) ***Term diversity.*** This challenge denotes that the comments with similar semantic usually will be written in various forms [10, 14, 17, 49]. On the one hand, the comment sentences are described by natural language that has a strong flexibility when constructing sentences. For instance, it is obvious that the meanings of the "*perhaps not really necessary*" and "*not absolutely necessary*" are exactly the same while the expressions of them are different. On the other hand, there might be different formally designated rules for writing comments in a specific real project. Thus, the comments from different projects may exhibit diverse style characteristics. For example, the tag "TODO" is frequently used in project ArgoUML, while the tag "Note" appears almost exclusively in project JEdit.

(2) ***Explainability.*** This challenge highlights that it is difficult to give intuitive SATD patterns to interpret the classification results [49]. Generally, there are some intuitive patterns (e.g., "todo") in the SATD comments that are easily identified by humans. However, for a given comment, most of automatic identification approaches (e.g., *NLP* and *TM*) cannot

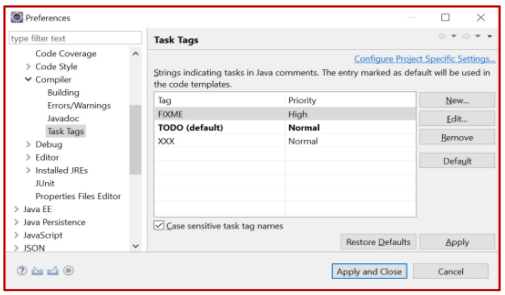
provide such intuitive patterns that indicate SATD. Practitioners will be confused by the classification results of such models. In particular, there are many misclassification cases in these classification models (e.g., *NLP*). Therefore, it is necessary to be able to explain the reason that a comment is classified as SATD comments.

In recent years, supervised approaches have been introduced to tackle the above two challenges. To deal with the first challenge, many studies employed various supervised models (e.g., *TM* [10], *NLP* [17], and *CNN* [49]) to extract common SATD-related semantic information from the comments of multiple source projects. To deal with the second challenge, Ren et al. exploited the computational structure of *CNN* to explain the prediction results by the SATD features and patterns learned by *CNN* [49]. In spite of the advance in SATD identification, the following new challenges are raised due to the use of (complex) supervised modeling techniques:

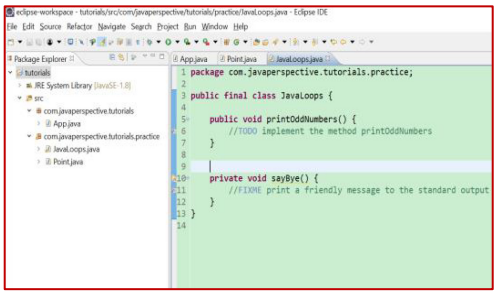
(1) ***Data dependency.*** For a supervised approach, there is a need to use the labeled data to train a model (e.g., *TM* [10], *NLP* [17], and *CNN* [49]). As such, its performance heavily depends on the quality and quantity of comment instances in the training data. If the training data has a low quality or/and is not enough, then such an approach will have a low performance. In the most extreme case, it cannot be applied if the training data is not available. For example, in Huang et al.'s study [10], *TM* performed worse in the test project JEdit than in other projects, which only achieved a precision of 0.410. Therefore, data dependency is an inevitable challenge for supervised modeling techniques.
(2) ***Risk of reproducibility.*** A supervised model often involves many parameters needed to be carefully tuned. In our community, it is not uncommon to see that not all the parameter settings are reported in detail when reporting a supervised model. Furthermore, for many supervised models, their source codes are not publicly available (e.g., *CNN* and *NLP*). Given this situation, for a supervised model, it is challenging for practitioners to replicate the excellent performance reported in the literature. This is especially true, when considering the fact that a tiny difference in parameter setting could lead to a huge difference in prediction performance. Therefore, this challenge may hinder the application of many supervised approaches.
(3) ***Computational costs.*** It is often the case that a supervised approach will consume a large amount of computational costs including time cost and hardware cost. In particular, the computational costs may be too large to be ignored when a large number of training data is used to build a supervised model. For instance, the *CNN* model built on a deep learning framework needs to spend several hours to train a new model on a GPU device [49]. At the same time, a large number of temporary files need be stored on the hard disk. In real software development, it is a strong desire for practitioners to use a lightweight approach to deal with such a simple task (i.e., text binary classification problem). Therefore, the computational costs may also pose a challenge for practitioners to use a supervised model in practice.

3 *MAT*: A SIMPLE HEURISTIC MATCHING TASK-ANNOTATION-TAGS APPROACH

In this section, we describe a simple heuristic approach (*MAT*) to identify SATD comments, which is a very natural but neglected baseline in SATD identification. First, we introduce our observations about the popular use of task tags in SATD comments in modern software development practices. Then, we give an overview of our heuristic baseline approach *MAT* base on these observations. Finally, we show the high feasibility of building such a simple-yet-effective baseline. With such a simple baseline, we are able to examine how far our community has really progressed in the journey of SATD identification.



(a) The preferences window for configuring task tags



(b) The code comments that contain task tags

(c) The task list shown in the task window

Fig. 1. An example usage scenario of task tags in Eclipse.

3.1 Popularity of Task Annotation Tags in Software Development

3.1.1 The Usefulness of Task Tags. According to the definition of SATDs, we can see that this kind of technique debt is often added by the developers themselves intentionally. In other words, developers write these technique debts for the sake of handling them easily in the future. In the real process of a project development, a project manager usually requests developers to use task tags or their team has an informal agreement to use task tags to mark these debts explicitly in purpose of checking and refactoring them in the task list more conveniently later [22].

Task tags are indicative words used as reminders for a work or an action that needs to be done by a developer [2]. Usually, task tags are embedded in source code comments, i.e., a string followed by a short description [59]. As a special kind of words, different task tags can express various problems in a project. Developers can roughly understand what types of problems exist in one code by reviewing the task tag directly. Well-known task tags are "TODO," "FIXME," and "XXX" [22]. Many popular IDEs (e.g., Eclipse[1] and NetBeans[2]) have supported developers to use task tags for team collaboration and task communication in development [58]. Figure 1(a) presents a "preferences" window for configuring task tags in Eclipse. It can be seen that the above popular task tags have been predefined in the default setting.

Task tags are very useful in programming. For example, Figures 1(b) and (c) present a usage scenario of task tags (taken from Reference [2]). When a developer is developing an application, he/she may define some methods (e.g., *printOddNumbers()*) and decide to implement them later. To avoid forgetting to implement a method, the developer can add a task comment (//TODO implement the method *printOddNumbers()*) to remind of the task. When checking the completeness of code, they can easily find out the code that needs to be implemented in time according to these (task) comments. In particular, Eclipse provides a predefined list of task tags that a developer can add to his/her code and view in a single location: the task view. A developer just needs to type a comment starting with the tag words "TODO" or "FIXME" in a new line. This line will automatically appear in the task window (Figure 1(c)) as soon as a developer saves the source code.

[1] https://www.eclipse.org/.
[2] https://netbeans.org/.

Table 1. The Default Task Tags in Popular IDEs (Four Representative Tags are Highlighted in Bold)

Popular IDEs	Default task tags
Eclipse [5]	**TODO, FIXME, XXX**
Visual Studio [7]	**TODO, HACK,** UNDONE, NOTE
IntelliJ IDEA [6]	**TODO, FIXME**
NetBeans [3]	@todo, **TODO, FIXME, XXX,** PENDING, <<<<<<
AndroidStudio [9]	**TODO, FIXME**
CodeClimate [8]	**TODO, FIXME, HACK, XXX,** BUG
Code::Blocks [4]	**TODO**
The representative tags	**TODO, FIXME, HACK, XXX**

According to Storey et al. [22], task tags play a key role in the work practices of software developers. In particular, task tags support articulation work such as the problem indicator and the edge case that would introduce underlying defects into a project. Similarly, task annotation tags in code comments would also occur near the problematic code. This reveals that task tags should be an excellent indicator of SATD comments. In their paper, they have listed many popular task tags (such as "TODO," "FIXME," "XXX," and "HACK").

3.1.2 The Representative Task Tags. After years of practice and development, many task tags have been gradually formed. Table 1 lists the default tags supported in popular IDEs [3–9]. As can be seen, "TODO" is the most popular task tag used by all the seven tools. In addition, "FIXME," "XXX," and "HACK" are supported by at least two popular IDEs. In this sense, "TODO," "FIXME," "XXX," and "HACK" are the representative four task tags. For each of these four tags, Table 2 lists the corresponding meanings and use scenario [1]. As can be seen, different task tags can indicate a specific type of problem at a more fine-grained level. In particular, Table 2 provides usage examples of task tags in the comments of the projects collected by Maldonado et al. [17].

3.1.3 Preliminary Statistical Evidence: The Availability and Universality of Task Tags in Real Projects. To investigate whether task tags have a good availability and universality for identifying SATD comments in real projects, we conduct a preliminary statistic based on the dataset provided by Maldonado et al. [17] (it is used in many prior studies [10, 49, 95]) to study the distribution of the above-mentioned four representative task tags in comments. In total, there are 10 projects in the dataset with 37,056 comment instances (see Table 5). To reduce the manual effort, we randomly select 10% SATD comments and 10% non-SATD comments from each project to manually inspect. We find that the number of the selected SATD comments ranges from 7 (EMF) to 96 (ArgoUML) on these projects, with a total of 277 SATD comments. The number of the selected non-SATD comments ranges from 211 (Hibernate) to 455 (ArgoUML) on these projects, with a total of 3,420 non-SATD comments.

Figure 2 depicts the percentage of comments (including SATD comments and non-SATD comments) that contain representative task tags. According to the statistical results, the percentages of task tags embedded in comment instances are higher than 60% for 8 out of the 10 projects. In particular, for two projects (i.e., ArgoUML and JRuby), almost all the sampled SATD instances include representative task tags. This indicates that, in real software development, it is common to use representative task tags to tag the code as suboptimal or remark underlying dangers. Note that, there are two special cases whose task tag inclusion ratio is lower than one-third (26% for JEdit and 29% for EMF). One possible reason is that the developers for these two projects use their

Table 2. The Meanings and Usage Scenarios of four Representative Task Tags

Tags	Meanings and usage scenarios	Examples

TODO	Comments that mark something for later work, later revision, or at least later reconsideration. TODO comments should be considered a very useful technique, although like all good things on Earth, there's certainly potential for abuse.	// TODO: Fully implement this! - *[from ArgoUML]*
		// TODO implement clear() - *[from Columba]*
		// TODO we should generate this. - *[from EMF]*

FIXME	A standard put in comments near a piece of code that is broken and needs work. Use FIXME to flag something that is bogus and broken.	// FIXME: not very efficient - *[from JRuby]* // FIXME: There's some code duplication here... - *[from JRuby]*
XXX	A marker that attention is needed. Commonly used in program comments to indicate areas that are kluged up or need to be. Some hackers like "XXX", to the notional heavy-porn movie rating. Use it to flag something that is bogus but works.	// XXX - why not simply new File (dir, filename)? -*[from Ant]* // XXX this should not hardcoded - *[from jEdit]*
HACK	Temporary code to force inflexible functionality, or simply a test change, or work around a known problem.	// HACK: force the controller to load its tree - *[from JMeter]*

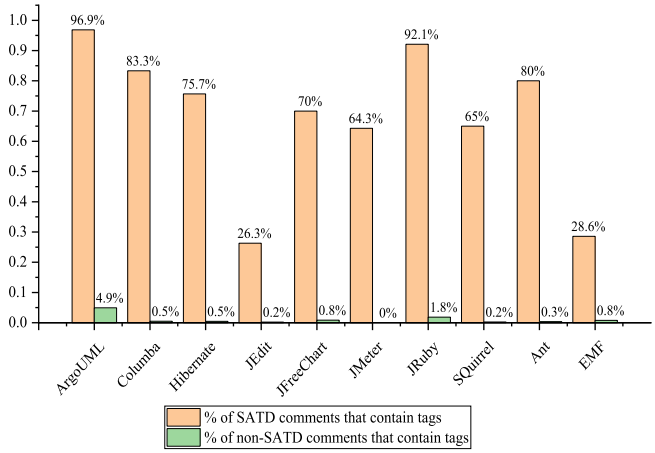


Fig. 2. The percentage of instances that contain task annotation tags.

user-defined tags (JEdit uses "WORKAROUND" and EMF uses "TBD") rather than the representative tags (shown in Table 2) provided by popular IDEs. When looking at the statistical result for non-SATD instances, we can see that, the percentage of non-SATD instances that contain representative task tags is very low (less than 5%). This indicates that most non-SATD instances do not contain representative task tags.

Table 3. The Detailed Number of Sampled Instances That Contain Each Task Tag on Each Project

		ArgoUML	Columba	Hibernate	JEdit	JFreeChart	JMeter	JRuby	SQuirrel	Ant	EMF
SATD Comment	HACK	0	0	0	2	0	2	4	1	1	0
	TODO	92	8	28	1	4	16	21	12	3	2
	FIXME	1	1	0	0	3	0	10	0	0	0
	XXX	0	1	0	2	0	0	0	0	4	0
	#All Tags	93	10	28	5	7	18	35	13	8	2
	#SATD	96	12	37	19	10	28	38	20	10	7
non-SATD Comment	HACK	0	0	0	0	0	0	1	0	0	0
	TODO	20	1	1	0	0	0	3	1	0	0
	FIXME	0	1	0	0	2	0	2	0	0	0
	XXX	2	0	0	1	0	0	0	0	1	2
	#All Tags	22	2	1	1	2	0	6	1	1	2
	#non-SATD	445	396	211	444	239	386	326	427	295	251

Table 3 reports the detailed number of sampled comments that contain each representative task tag on each project. In this table, "#All Tags" denotes the number of comments that contain any one of the four task tags. In other words, it is the sum of numbers of any one task tag. For example, "#All Tags" in the third column and the six row is 93 (93 = 0 + 92 + 1 + 0). "#SATD" (or "#non-SATD") denotes the number of SATD (or non-SATD) comments in the samples. As can been seen, "TODO" occurs in all the 10 projects for SATD comments and occupies most of the proportion (i.e., 85.39%) compared with the other tags. One possible reason is that many developers use auto-generated annotations by the IDE (e.g., Eclipse), which sets "TODO" as the default tag [5]. Therefore, "TODO" is the most popular tag for indicating SATD. The other tags such as "FIXME" and "XXX" have preferences on specified projects such JRuby and Ant but appear less frequently in the other projects. Considering the result for non-SATD comments, there are few task tags for non-SATD instances. Note that, some task tags (e.g., "HACK" and "XXX") do not occur in many projects.

In summary, we conclude that the four representative task tags (i.e., "TODO," "FIXME," "XXX," and "HACK") occur more frequently in SATD comments than in non-SATD comments, and they exist in almost all the investigated 10 projects. In other words, they should have a good availability and universality to distinguish between SATD comments and non-SATD comments.

3.2 *MAT*: Fuzzily Matching Annotation Tags to Identify SATD

Inspired by the results in our observations, we develop a simple heuristic approach that matches the four representative task annotation tags (*MAT* for short) to identify SATD comments. To ensure the accuracy of our approach, we take the following two phases to complete the identification process: "preprocessing" and "fuzzy matching" (as shown in Figure 3). Additionally, a formal algorithm of *MAT* is shown in Figure 4.

(1) **Preprocessing:** The first phase is to preprocess the text of comments. Different developers may have different habits of comment writing. For example, "TODO" can be written in the form of "todo," "Todo," or "TODO." Therefore, we conduct a natural language preprocessing similar to that in Huang et al.'s study [10]. The "preprocessing" phase contains the following two main steps:

(a) *Tokenization:* Divide the continuous text into single words and keep only English letters in a token. During this process, all words are converted to lowercase for the convenience of matching them.

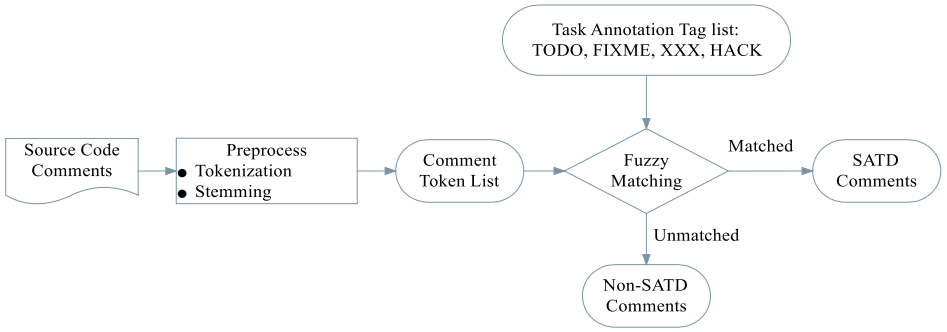


Fig. 3. An overview of *MAT*.

Algorithm 1: *MAT(C, D)*

// Predicting the label (SATD or non-SATD) of a comment
Input: C is a comment instance
D is a dictionary of task tags, In default, $D = \{todo, fixme, hack, xxx\}$
Output: L is the prediction label of C *i.e., SATD or non-SATD*
1: $L = non\text{-}SATD$
2: $tokens = Tokenization(C)$
3: $stemmed\_tokens = Stemming(tokens)$
4: **for** $tag_i \in D$ **do**
5: **if** ($FuzzyMatcher(tag_i, stemmed\_tokens)$ == true) **then**
6: $L = SATD$
7: **break**
8: **end if**
9: **end for**
10: **return** L

Fig. 4. The algorithm description of *MAT*.

(b) *Stemming:* Transform every word into the original form (e.g., "hacks" to "hack") to improve the matching accuracy.

(2) **Fuzzy matching:** The second phase is to match task tags in the preprocessed token list. A comment is considered indicating SATD if and only if there is at least one task tag ("TODO," "FIXME," "XXX," or "HACK") that occurs in the corresponding token list. At the same time, although the comments are preprocessed at the first phase, there are some unexpected cases that exist in the token set. For example, two words are linked together by deleting the space by a mistaken operation (e.g., "pleasefixme" and "hackhere"). These words cannot be directly matched with task tags. Therefore, we use a fuzzy matching strategy, which will match a word if a task tag is contained in the start or the end of a word, to find task tags. More specifically, we will consider tokens such as "fixme," "pleasefixme," or "fixmehere" as the matched tokens of "FIXME." If a comment consists of the above tokens, then it will be classified as a SATD comment. As can be seen, the fuzzy matching strategy can be easily implemented by using loose regular expressions.

3.3 Potential as a Baseline in SATD Identification

Due to the simplicity of *MAT*, it is very likely to be a baseline approach in SATD identification. In this section, we state why *MAT* can be used as a baseline approach in the perspective of necessity and possibility.

3.1.1 The Necessity of a Simple-yet-effective Baseline Approach in SATD Identification. In recent years, many researchers have highlighted the necessity of a simple-yet-effective baseline approach in their respective research fields [61]. According to the latest studies [63], the benefits of using a simple-yet-effective baseline approach are mainly two-fold. On the one hand, this would ensure researchers to adequately compare and evaluate the performance of a newly proposed approach (SATD identification approach in our context). To be useful, a newly proposed approach should have a significantly better performance than the simple baseline approach and the corresponding effect size should be non-trivial. Otherwise, the motivation for introducing such a new approach could not be well justified. This is especially true if the proposed new approach is highly complex compared with the baseline approach. On the other hand, the *"ongoing use of a baseline approach in the literature would give a single point of comparison"* [63]. In the literature, a newly proposed approach is often compared against the state-of-the-art approaches. The underlying assumption is that a new approach is useful if it shows a superior performance. However, in our community, it is not a common practice to share their codes. As a result, researchers often have to re-implement the state-of-art approaches, where a tiny difference in the implementation may lead to a degraded performance. In this case, it will be misleading to report that the proposed new approach advances the state-of-the-art. Furthermore, due to the lack of a common baseline approach, it is not clear how far have we really progressed in the journey. These problems can be avoided if there is an ongoing use of a baseline approach whose code is publicly available. In other words, a baseline approach defines a meaningful point of reference and hence allows a meaningful evaluation of any new approach against previous approaches.

The importance of a baseline approach has been well recognized in our community. In the field of software engineering [61–64], many baseline approaches have been proposed. For example, Krishna et al. proposed a baseline approach for transfer learning [61]. Chen et al. used "sampling" as a baseline optimizer for search-based software engineering [62]. Zhou et al. suggested that ManualDown and ManualUp should be used as the baseline approaches in cross-project defect prediction [64]. In addition to software engineering, a large number of baseline approaches [66–72] also were proposed in other fields. For instance, Xiao et al. proposed simple baselines for human pose estimation and tracking [69], while Ethayarajh used a strong but simple baseline to build unsupervised random walk sentence embedding [70].

In SATD identification, there is no simple-yet-effective baseline approach available. Although a variety of approaches have been proposed (i.e., *Pattern, NLP, TM, CNN*, and *Jitterbug*, shown in Table 4), they do not satisfy the characteristics that a baseline approach should have. According to Whigham et al. [63] and Krishna et al. [61], to be both useful and widely used, a baseline approach at least should have the following important characteristics: (1) it should be simple to describe, implement, and interpret; (2) it should be deterministic in its outcomes; (3) it should be publicly available via a reference implementation and associated environment for execution; and (4) it should offer a comparable performance to standard approaches. A baseline approach holding the above characteristics will facilitate researchers in filtering out actually useless approaches and to determine really useful approaches in SATD identification. As shown in Table 4, *Pattern* has a low accuracy, while *NLP, TM*, and *CNN* are complex (many parameters need to be tuned). Meanwhile, *Jitterbug* spends a large amount of human effort. In particular, according to

Table 4. The Comparison among Existing SATD Identification Approaches

Approach	Solution	Advantage	Disadvantage	Reproducibility
Pattern	Matching 62 human-summarized patterns	Simple intuitive	Human-summarized low accuracy	Easy (open source)
NLP	Leveraging the maximum entropy classifier	High accuracy	Complex model	Medium (partial open source)
TM	Training Naive Bayes classifier and adopting voting strategy	High accuracy	Less intuitive complex model	Easy (open source)
CNN	Training convolution neural network	High accuracy intuitive	Very complex model	Hard (not open source)
Jitterbug	Extracting SATD-related patterns and utilizing human efforts	High accuracy intuitive	Cost human efforts	Easy (open source)

References [86, 87], *CNN* is hard to reproduce due to its complex structure and parameters. Currently, there is a need to develop a simple-yet-effective baseline approach for SATD identification.

3.3.2 The Possibility to Build a Simple-yet-effective Baseline Approach in SATD Identification. As Ren et al. stated, the essence of identifying SATD comments is a task of binary text classification [49]. The solutions to these kinds of tasks usually depend on the extraction of semantic information and the understanding of context. As can be seen, existing approaches described in Section 2 utilize the semantic information of comment texts in different ways. *Pattern* summarized patterns manually to represent the characteristics of SATD. *NLP* extracted the n-gram information to represent the semantic of sentences. *TM* extracted top features as the semantic context of SATD to train a classification model based on text-mining techniques. *CNN* utilized the semantic word-embedding vectors, which contained rich semantic information, of words in comments to train a powerful prediction model. The unsupervised *Pattern* is the most intuitive and easiest to understand. However, its recall rate is too low [10]. The other four supervised approaches can achieve a high accuracy, but their modeling processes are complex. For example, there is no intuitive explanation for a given result predicted by the *TM*.

Is it possible to build a simple-yet-effective baseline approach for SATD identification? As shown in Section 3.1, we have the following facts: (1) many task tags such as "TODO" and "FIXME" have been designed by our community to be used as "reminders of actions, work to do or any other action required by the programmer" [5]; and (2) these task tags have been supported by popular IDEs and have been widely used in many real-world projects. This means that these tags should be strong indicators of SATD in practice, even if some developers may misuse or do not use them for some reasons (e.g., someone does not know how to use "TODO" or how to use "FIXME"). Given this situation, we naturally have the following idea: If we use task tags to identify SATD, a high accuracy will be obtained. In other words, intuitively, it is highly possible to build a simple-yet-effective baseline approach to identifying SATD by matching task tags in comments. ***Interestingly, we find that all prior supervised approaches learn to know that these task tags are strong SATD indicators. In fact, such knowledge is clearly prior, i.e., there is no need to get such knowledge by a learner.*** In this sense, matching task tags in comments is a simple, direct, and natural idea for identifying SATD comments, which are yet neglected by our community. This intuition motivates us to use prior task tag knowledge to develop *MAT*. By comparing *MAT* with existing SATD identification approaches, we can get an in-depth understanding of *"how far we have really progressed in the journal of automatic SATD identification."*

4 EXPERIMENTAL SETUP

This section describes the experimental setup. First, we introduce the research questions that the experiments would like to answer. Then, we describe the studied datasets and their collection process. After that, we shed light on the two commonly used prediction scenarios. Next, we explain the experimental results collection method of the compared approaches. Finally, we present the indicators used for performance evaluation.

4.1 Research Questions

To conduct a fair comparison between *MAT* and existing approaches, we first set up the following research question (RQ1) in our empirical study:

- **RQ1 Reproducibility of existing approaches**

Could we reproduce existing approaches so the performances of reproduced approaches are closed to the original ones?

Then, in the following research questions (RQ2 and RQ3), we conduct detailed experiments for comparing the classification performance and classification difference between *MAT* and existing approaches:

- **RQ2 Classification performance comparison**

How effective are existing approaches in identifying SATD comments compared with MAT?

- **RQ3 Difference in correct classification results**

What is the difference of the correct classification results between existing approaches and MAT in SATD identification?

Finally, we analyze for supervised approaches the weaknesses in SATD identification and explore possible improvement strategies in the following research question (RQ4):

- **RQ4 Weakness and possible improvement**

(1) Why are the supervised approaches not very outstanding compared with MAT?
(2) Which SATD comments cannot be identified by the supervised approaches?
(3) Can the effectiveness of supervised approaches be promoted by incorporating MAT?

4.2 Studied Datasets

To answer above RQs, we use two datasets, collected by Maldonado et al. [17] and ourselves, respectively, in our experiments.

(1) Dataset-M: collected by Maldonado et al. This dataset was shared by Huang et al. online.[3] On their website, there are two important data files: "comments" and "labels." The "comments" file lists the comments after filtering with heuristic rules from 10 open-source software projects, while the "labels" file lists the label (i.e., a SATD comment or a non-SATD comment) for each comment. These 10 open-source projects include ArgoUML 0.34, Columba 1.4, Hibernate 3.3.2, JEdit 4.2, JFreeChart 1.0.19, Jmeter 2.10, Jruby 1.4.0, Squirrel 3.0.3, Ant 1.7.0, and EMF 2.4.1. As stated by Huang et al., the dataset was originally provided by Maldonado et al. [17, 18]. For each project, Maldonado et al. used an open-source Eclipse plug-in (i.e., JDeodorant [48]) to parse source code and extract code comments. In particular, they applied the following five heuristic rules to filter out those comments that were obviously impossible to be SATD comments: (1) removed license comments; (2) grouped consecutive single-line comments as one comment; (3) removed commented

[3] https://github.com/tkdsheep/TechnicalDebt.

source code fragments that usually did not contain SATD; (4) removed automatically generated comments by the IDE (e.g., Eclipse); and (5) removed Javadoc comment unless they contained at least one task annotation tag (e.g., "TODO:" and "FIXME:"). After obtaining the comments filtered with these heuristic rules, Maldonado et al. manually determined their labels (i.e., whether a comment was a SATD comment).

(2) Dataset-G: collected by ourselves. Following the same data collection method as used in Dataset-M, we collected a new dataset, Dataset-G, from another 10 popular open-source Java projects. The purposes of this are two-fold. First, using a new dataset provides us an opportunity to observe whether the findings obtained on Dataset-M (the commonly used dataset in the literature) can be generalized to other projects. Second, combining Dataset-M with Dataset-G together allows us to have a large sample to draw statistically meaningful conclusions. The new projects involved in Dataset-G include Dubbo-2.7.4, Gradle-5.6.3, Groovy-2.5.8, Hive-3.1.2, Maven-3.6.2, Poi-4.1.1, SpringFramework-5.2.0, Storm-2.1.0, Tomcat-9.0.27, and Zookeeper-3.5.6. These projects are selected because they are open source, are well commented, and vary in the number of contributors and the size of projects. In total, we extract 266,980 comments from these 10 projects and obtain 81,260 comments after filtering. In particular, the number of comments of project Hive is the largest (i.e., 29,340) and that of project Maven is smallest (i.e., 1,219). In our study, the first author manually labeled these comments as SATD and non-SATD following Maldonado et al.'s labeling tutorial.[4] As a result, we got 2,995 SATD comments (about 1.10% of the total comments). This process took about 200 hours (one month). To mitigate the risk of providing a biased dataset, we also extracted a statistically significant sample that was created based on the total number of comments (81, 260) with a confidence level of 99% and a margin of error of 5%, resulting in a stratified sample of 661 comments.[5] According to the principle of stratified sampling, there are 96% comments without self-admitted technical debt (634 comments) and 4% (i.e., 27) SATD comments. The first and second authors relabeled these comments individually. Based on the two group labels, we calculated the Cohen's Kappa coefficient [78] to evaluate inter-rater agreement level for categorical scales. The level of agreement measured between two reviewers achieved to 0.86 (that is larger than 0.75), an excellent agreement according to Fleiss [79]. To facilitate the research in this field, the dataset is provided online [80].

Table 5 provides the detail statistics of the 20 projects that belong to different application domains used in our study. The first column shows the group name of each dataset. In the following, we use **Dataset-M** and **Dataset-G** to refer to the dataset provided by Maldonado et al. and ourselves, respectively. The second column reports for each project the name. The third and fourth columns report the number of original comments and number of the resulting comments after filtering. The fifth and sixth columns report the number and the proportion of SATD comments in the resulting comments. The seventh column reports the number of contributors extracted from an online community and public directory OpenHub.[6] The last two columns report the project size statistics by SLOCCount, including the number of classes and the code size in KLOC.[7] On average, the number of the resulting comments for a project is about 5,916, of which 1.10% are SATD comments.

Note that Huang et al. only used 8 out of 10 datasets to conduct their study (Ant 1.7.0 and EMF 2.4.1 were not used) [10]. Furthermore, we find that, for each of the 8 projects, the number of comments filtered with the heuristic rules on their website is slightly different from that in their

[4] https://github.com/Naplues/tse.satd.data/blob/master/replication%20and%20understanding/labeling_turorial.md.
[5] https://www.calculator.net/sample-size-calculator.html?type=1&cl=99&ci=5&pp=50&ps=62566&x=90&y=10.
[6] https://www.openhub.net/.
[7] http://www.dwheeler.com/sloccount/sloccount.html.

Table 5. Statistics of the Comments in the 20 Projects

Dataset	Project	#Comments	After filtering	#SATD	% of SATD	Cont.	# of classes	KLOC
Maldonado et al. collected (Dataset-M)	Ant	21,587	3,052	102	0.47%	74	1,475	115
	ArgoUML	67,716	5,426	969	1.43%	87	2,609	926
	Columba	33,895	4,090	128	0.38%	10	1,711	155
	EMF	25,229	2,585	74	0.29%	30	1,458	228
	Hibernate	11,630	2,492	377	3.24%	314	1,356	703
	JEdit	16,991	4,644	195	1.15%	57	800	310
	JFreeChart	23,474	2,494	101	0.43%	19	1,065	317
	JMeter	20,084	4,148	282	1.40%	41	1,181	354
	JRuby	11,149	3,652	383	3.44%	374	1,486	841
	SQuirrel	27,474	4,473	201	0.73%	40	3,108	708
	Total	**259,229**	**37,056**	**2,812**	**1.08%**	**1,046**	**16,249**	**4,657**
We collected (Dataset-G)	Dubbo	5,875	1,649	85	1.45%	255	2,532	141
	Gradle	15,901	3,324	321	2.02%	409	13,541	406
	Groovy	14,199	4,435	249	1.75%	284	2,729	181
	Hive	81,127	29,340	1,046	1.29%	192	15,463	1,257
	Maven	5,448	1,219	136	2.50%	87	1,158	84
	Poi	45,666	15,033	618	1.35%	12	4,793	406
	SpringFramework	42,574	7,712	98	0.23%	401	14,686	654
	Storm	12,258	3,639	92	0.75%	304	4,787	282
	Tomcat	37,038	12,218	287	0.77%	31	4,120	335
	Zookeeper	6,894	2,691	63	0.91%	93	1,322	87
	Total	**266,980**	**81,260**	**2,995**	**1.12%**	**2,068**	**65,131**	**3,833**
Total	-	**526,209**	**118,316**	**5,807**	**1.10%**	**3,114**	**81,380**	**8,490**
Average	-	**26,310**	**5,915.8**	**290**	**1.10%**	**156**	**4,096**	**425**

published paper. After communicating with Huang, we were told that their dataset was updated after their paper was published. In other words, the dataset on their website is more reliable. Therefore, in our article, we use the up-to-date dataset (from 10 projects) on their website to evaluate the effectiveness of *MAT* as well as the state-of-the-art approaches. It is worth mentioning that Ren et al. [49] also conducted their experiments on Dataset-M, although the statistics information reported in their paper is inconsistent with the information in Table 5 (this was confirmed after communicating with Ren).

4.3 Prediction Scenarios

According to the existing works [10, 17, 49], we design the following two prediction scenarios based on the above datasets to conduct the experiments:

- ***Many-to-one (MTO) prediction.*** Under this scenario, for each supervised approach (i.e., *NLP, TM*, and *Easy*), each project is used as the test dataset to evaluate a model built on the training dataset combined from the remaining 19 projects. For each unsupervised approach (i.e., *Pattern* and *MAT*), it is directly applied to the test data without a training process. For each approach except *CNN*, we have 20 experiments. For *CNN*, we have 10 experiments (i.e., the results obtained from the 10 projects on Dataset-M). In total, we have 110 classification results (5 approaches $\times$ 20 projects + 1 approach (*CNN*) $\times$ 10 projects). After obtaining the results on the 20 projects, we can investigate the non-effort-aware classification effectiveness (i.e., precision, recall, and F_1) and effort-aware classification effectiveness (i.e., ER and RI) of *MAT* compared with *Pattern, NLP, TM, CNN*, and *Easy*.

- ***One-to-One (OTO) prediction.*** Under this scenario, on a given test project, for each of the three supervised approaches *NLP, TM*, and *Easy*, we have 19 experiments (i.e., we train 19 models using 19 source projects to predict the labels of comments in the target project, respectively) and use their average performance as the resulting performance. For *CNN*, we have 9 experiments (i.e., the results obtained from the 10 projects on Dataset-M) and use their average performance as the resulting performance. For *MAT*, we have 1 experiment (i.e., we predict the labels of comments in the target project directly) on each testing dataset. In total, we have 90 classification results (4 approaches (*NLP, TM, Easy*, and *MAT*) × 20 projects + 1 approach (*CNN*) × 10 projects). After obtaining the results on 20 (or 10) projects, we can investigate the effectiveness of *MAT* compared with the *NLP, TM*, and *Easy* (or *CNN*) approaches. Note that, we do not compare *MAT* and *Pattern*, since their results are same as those in Scenario 1.

4.4 Compared Approaches

We examine the real progress in SATD identification by comparing *MAT* with the following approaches proposed in the literature: *Pattern* [14], *NLP* [17], *TM* [10], *CNN* [49], and *Easy* in *Jitterbug* [95]. As described in Section 2, *Pattern* is an unsupervised approach, while the *NLP, TM, CNN*, and *Easy* are supervised approaches. In the following, we describe how to obtain the experimental results of each approach.

Pattern. We use the patterns (i.e., keywords or phrases appearing frequently in SATD comments) summarized by Potdar et al. to identify the label of each comment in a target project. If any of these patterns is found in a comment, it will be classified as a SATD comment. Otherwise, it will be classified as a non-SATD comment. Similar to the *Pattern*, our *MAT* approach is also an unsupervised approach. However, their differences are obvious. As can be seen, *MAT* uses popular task annotation tags that are inherently supported by many IDEs designed by our community to indicate SATD. Furthermore, unlike the *Pattern, MAT* takes a fuzzy matching strategy, rather than a strict matching strategy, to identify SATD.

NLP. Since Maldonado et al. [17] and Huang et al. [10] did not share the source code of the *NLP*, we read their papers carefully and implement the *NLP* by ourselves to compare the performance with *MAT*. Note that the core of the *NLP* is based on the Stanford Classifier [60]. Our implementation also used the same classifier to train the *NLP* model and the reproduced code is available online [80].

TM. We use the code shared by Huang et al. [10] for *TM*. Specifically, for a given target project, we first train a single sub-classifier for each of the other individual source projects. Then, we use the resulting sub-classifiers to vote for the label of each comment in the target project.

CNN. Since the code of *CNN* is not shared online, we first obtained the *CNN* code for within-project prediction after communicating with Ren [49]. Then, we modified the *CNN* code for cross-project prediction according to the descriptions in Reference [49]. In particular, we set the parameters in *CNN* with Ren's help. However, as shown in the results of RQ1 (Reproducibility for the state-of-the-art approaches), we are unable to reproduce the experimental results reported in Ren et al.'s paper [49]. Moreover, this is not surprising, as reproducibility in deep learning is a huge challenge.[8] The main reason is that many factors contribute to non-determinism: initialization of layer weights, dataset shuffling, randomness in hidden layers, updates to ML frameworks, libraries, or drivers, and even hardware used during the training process. Therefore, to avoid the implementation bias, we will use the prediction performance values of *CNN* reported in their study directly

[8] https://docs.paperspace.com/machine-learning/wiki/reproducibility-in-machine-learning.

Table 6. The Confusion Matrix

		Actual label	
		SATD	Non-SATD
Predicted label	SATD	TP	FP
	Non-SATD	FN	TN

to conduct the comparison in Dataset-M. This ensures that we make a fair comparison between *CNN* and *MAT* in SATD identification.

Easy. *Jitterbug* is a two-step framework that can automatically identify "*Easy*" SATD comments and recommend a "Hard" SATD comments list to developers for manually labelling iteratively. In Section 5, we only use *Easy* as our compared classification approach, since the Hard component is not automatic ("Hard" needs to interact with developers to manually label the recommended comments as SATD or non-SATD). In Section 6, we will further compare *MAT* with *Jitterbug*. We use the code shared by Yu et al. [95] to obtain the result of "*Easy.*"

4.5 Performance Evaluation

As can be seen, the process of identifying SATD comments is a typical binary classification problem [42–44] in nature, which aims to determine whether a comment indicates SATD or not. For the classification result, there are a total four situations (TP, FP, TN, and FN) shown in a confusion matrix in Table 6. Each row represents the comments in a predicted class, while each column represents the comments in an actual class. More specifically, TP (true positive) denotes the set of correctly classified SATD comments, FP (false positive) denotes the set of mistakenly classified non-SATD comments, TN (true negative) denotes the set of correctly classified non-SATD comments, and FN (false negative) denotes the set of mistakenly classified SATD comments.

Based on the aforementioned confusion matrix, the following three popular performance indicators are often used to evaluate the classification performance of a SATD identification approach [10, 17, 49, 95]:

Precision is the proportion of comments that are correctly classified as SATD comments among those classified as SATD comments. If the precision of an approach is high, then the SATD comments identified by this approach are usually correct.

$$Precision = \frac{|TP|}{|TP| + |FP|} \tag{1}$$

Recall is the proportion of comments that are correctly classified as SATD comments among those true SATD comments. If the recall of an approach is high, then a large percentage of real SATD comments can be found in the result of classification.

$$Recall = \frac{|TP|}{|TP| + |FN|} \tag{2}$$

F_1 is the harmonic mean of precision and recall. According to Han et al. [21], precision and recall are complementary indicators. That is, in many cases, the increase of the precision of an approach will lead to a decrease of the recall and vice versa. Therefore, it is not comprehensive to evaluate the performance using the two indicators separately. F_1 reflects the comprehensive contribution of both.

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

Although the above three indicators measure the accuracy of classification results, they do not reflect the effort of reviewing the classification results that developers actually need to spend.

Many software quality assurance studies (e.g., defect prediction [53, 55, 56], bug localization [96], vulnerability prediction [72], and fault prediction [76]) have emphasized the importance of using effort-aware indicators to evaluate a prediction model. To conduct a comprehensive comparison, we hence include the following two effort-aware indicators:

ER (Effort Reduction) is the proportion of the reduced number of comments to be inspected (i.e., effort) by a model *m* compared with a random model *random* that achieves the same recall of SATD comments. For a target project, let N be the total number of comments and *n* be the number of actual SATD comments. Assume that a model (approach) *m* predicts *x* instances as SATD comments, in which *y* comments are actual SATD comments. Thus, the model *m* will recommend *x* comments to developers for review. In this case, the recall of model m is *y/n*. After a perfect review, developers can find *y* SATD comments. That is to say, the effort ratio of reviewing the result output by model *m* is as follows:

$$Effort_m = \frac{x}{N}. \tag{4}$$

For a random model *random*, to achieve the same recall rate as model *m* (i.e.,y/n), there is a need to randomly select $y/\ n * N$ comment instances for review. Therefore, the effort ratio of reviewing the result output by a random model is:

$$Effort_{random} = \frac{\frac{y}{n} * N}{N} = \frac{y}{n}. \tag{5}$$

Therefore, according to the definition of $Effort_m$ (Equation (4)) and $Effort_{random}$ (i.e., Equation (5)), we can get the effort reduction (ER) of model *m* compared with a random model as follows:

$$ER_m = \frac{Effort_{random} - Effort_m}{Effort_{random}} = \frac{y * N - x * n}{y * N}. \tag{6}$$

According to the confusion matrix, we can get that $x = |TP| + |FP|$, $y = |TP|$, $n = |TP| + |FN|$, $N = |TP| + |FP| + |FN| + |TN|$. $ER_m > 0$ indicates that model *m* is better than a random model, and $ER_m < 0$ indicates that model *m* is worse than a random model. For two models *m1* and *m2, m1* is better if $ER_{m1} > ER_{m2}$.

RI (Recall Increase) is the proportion of the increased recall by a model *m* compared with a random model *random* when *m* and *random* inspect the same number of comments (i.e., the same effort). As described above, the recall of model *m* is as follows:

$$Recall_m = \frac{y}{n}. \tag{7}$$

In this context, the corresponding number of inspected comments (i.e., effort) is *x*. Under the same effort, the recall of a random model is as follows:

$$Recall_{random} = \frac{\frac{n}{N} * x}{n} = \frac{x}{N}. \tag{8}$$

Here, n/N denotes the probability of a random model predicting an instance as SATD comment. Therefore, $\frac{n}{N} * x$ is the number of SATD comments identified by a random model. According to the definition of $Recall_m$ (Equation (7)) and $Recall_{random}$ (i.e., Equation (8)), we can get the **recall increase (RI)** of model *m* compared with a random model as follows:

$$RI_m = \frac{Recall_m - Recall_{random}}{Recall_{random}} = \frac{y * N - x * n}{x * n}. \tag{9}$$

$RI_m > 0$ indicates that model *m* is better than a random model, and $RI_m < 0$ indicates that model *m* is worse than a random model. For the two models *m1* and *m2, m1* is better if $RI_{m1} > RI_{m2}$.

5 EXPERIMENTAL RESULTS

In this section, we report the experimental results in detail. To enable external replication, we make the datasets, source codes of *MAT*, and experimental results in our study publicly available [80].

5.1 RQ1: Reproducibility of Existing Approaches

To conduct accurate comparisons and get reliable conclusions, all comparisons between existing approaches and *MAT* should be based on accurate results. As mentioned above, *Pattern* is a simple keyword matching approach that can be reproduced easily and accurately. Meanwhile, the implementation codes of *TM* and *Easy* have been opened by their authors. Therefore, we can get the entirely same results as they reported. However, there are no open-source codes for both *NLP* and *CNN*. Therefore, we need to reproduce the two approaches to conduct subsequent experiments. The answer to this RQ can ensure the correction of the conclusions obtained in subsequent experiments and provide an evidence for researchers the difficulty of reproduction in different approaches. For each of the compared SATD approaches, we read the corresponding paper carefully and implement the approach according to its description strictly. In particular, we use the same datasets as used the original studies to conduct the experiments, thus enabling a direct comparison between our reproduced results and their original results. The following are the reproducibility details of each approach:

- ***Reproduction details of NLP approach.*** To the best of our knowledge, the main component of *NLP* is the Stanford Classifier, which contains all the parameters (i.e., 14) of *NLP*. To reproduce this approach, we carefully adjust each parameter according to their description. The parameter settings of Stanford Classifier used in our reproduced *NLP* are available in Reference [80].
- ***Reproduction details of CNN approach.*** According to Ren et al.'s work [49], the implementation of *CNN* is based on a simple sentence classification model [94]. In particular, they leveraged weighted loss function to deal with the class imbalance issue and conducted a sensitivity analysis to obtain the optimal values of three important hyper-parameters: the dimension size of word embedding, the number of filters, and the combination of filter size [52]. According to Ren et al.'s description and a code draft they provided, we apply their weighted loss function and tuned values for the parameters on the simple sentence classification model. The parameter settings of our reproduced *CNN* are also available in Reference [80].

Results of reproduced NLP approaches. To examine the correctness of *NLP* we implemented, we use Table 7 to report the comparison of *NLP* reproduced by Huang et al., Ren et al., and ourselves. Note that there are only 8 projects considered in Huang et al.'s experiment (Ant and EMF were not used), while there are 10 projects considered in Ren et al.'s experiment. To make a fair comparison, we use the same projects as used in their studies. Considering the F_1 indicator, on average, our *NLP* can achieve a value of 0.705 on 8 projects (and 0.654 on 10 projects), which is higher than 0.576 obtained by Huang et al. on 8 projects and 0.624 obtained by Ren et al. on 10 projects. Similar results can be found in terms of average recall. As for average precision, our result is higher than Huang et al.'s result. Although our precision is slightly lower than Ren et al.'s result, the average F_1 value of our result is higher. Note that, the performances of our implemented *NLP* on 10 projects are lower than that on 8 projects. The reason is that Ant and EMF have different comment characteristics from the other 8 projects (e.g., many SATD comments in the two projects do not contain important features such as "todo"). Therefore, it is easy to understand that adding the comments of these two projects into the training set will have negative impacts on the

Table 7. The Comparison of *NLP* Approaches Implemented by Different Authors Based on Dataset-M

	Compared with Huang et al. under 8 projects						Compared with Ren et al. under 10 projects					
	Precision		Recall		F_1		Precision		Recall		F_1	
Implemented by	Huang et al.	Ourselves	Huang et al.	Ourselves	Huang et al.	Ourselves	Ren et al.	Ourselves	Ren et al.	Ourselves	Ren et al.	Ourselves
Ant	-	-	-	-	-	-	0.524	0.476	0.431	**0.480**	0.473	**0.478**
ArgoUML	0.726	**0.801**	0.897	0.886	0.802	**0.841**	0.821	0.798	0.851	**0.878**	0.836	0.836
Columba	0.677	**0.792**	0.688	**0.742**	0.682	**0.766**	0.786	0.754	0.719	**0.742**	0.751	0.748
EMF	-	-	-	-	-	-	0.449	0.433	0.297	**0.392**	0.358	**0.411**
Hibernate	0.565	**0.837**	0.610	**0.682**	0.587	**0.751**	0.862	0.822	0.663	**0.674**	0.750	0.741
Jedit	0.473	**0.686**	0.446	0.359	0.459	**0.471**	0.691	0.667	0.333	**0.369**	0.450	**0.475**
JFreeChart	0.516	**0.633**	0.485	**0.614**	0.500	**0.623**	0.846	0.663	0.436	**0.644**	0.575	**0.653**
Jmeter	0.503	**0.779**	0.624	**0.738**	0.557	**0.758**	0.836	0.757	0.741	0.706	0.786	0.730
Jruby	0.589	**0.782**	0.580	**0.760**	0.584	**0.771**	0.791	**0.796**	0.504	**0.794**	0.616	**0.795**
Squirrel	0.325	**0.623**	0.657	**0.692**	0.435	**0.656**	0.655	**0.657**	0.642	**0.687**	0.648	**0.672**
Average	0.547	**0.742**	0.623	**0.684**	0.576	**0.705**	**0.726**	0.682	0.562	**0.637**	0.624	**0.654**
Median	0.541	**0.781**	0.617	**0.715**	0.571	**0.755**	**0.789**	0.711	0.573	**0.681**	0.632	**0.701**

Table 8. The Performance of *CNN* Approaches Implemented by Different Authors Based on Dataset-M

	Precision		Recall		F_1	
Implemented by	Ren et al.	Ourselves	Ren et al.	Ourselves	Ren et al.	Ourselves
Ant	0.584	0.515	0.758	0.686	0.660	0.588
ArgoUML	0.816	0.766	0.950	0.928	0.878	0.839
Columba	0.830	0.390	0.875	0.773	0.852	0.518
EMF	0.793	0.158	0.594	0.527	0.679	0.243
Hibernate	0.930	0.771	0.743	0.849	0.826	0.808
Jedit	0.773	0.239	0.489	0.456	0.599	0.314
JFreeChart	0.686	0.199	0.802	0.851	0.739	0.323
Jmeter	0.873	0.815	0.787	0.780	0.828	0.797
Jruby	0.805	0.752	0.930	0.697	0.836	0.724
Squirrel	0.794	0.812	0.692	0.622	0.739	0.704
Average	**0.788**	0.542	**0.762**	0.717	**0.764**	0.586
Median	**0.800**	0.634	**0.773**	0.735	**0.783**	0.646

performance of a supervised approach (i.e., *NLP*). According to the comparison result, the *NLP* we implemented is excellent, so it can be used for subsequent comparison.

Results of reproduced CNN approaches. Table 8 reports the performance comparison of CNNs provided by Ren et al. and reproduced by us on Dataset-M. As can be seen, there is a large difference in performance between the two implementations. On average, the precision, recall, and F_1 of *CNN* implemented by us can achieve 0.542, 0.717, and 0.586, which, respectively, has a deterioration of 31.22%, 5.91%, and 23.30% compared with the results reported by Ren et al. This means that our implemented *CNN* cannot reproduce the excellent performance reported by Ren et al. In particular, the precision of our *CNN* implementation on EMF, JEdit, and JFreeChart are especially low (less than 0.3). The reason may be that the parameters of *CNN* are too complex to reproduce. To make a fair comparison between *CNN* and *MAT* in later experiments, we only use the results reported by Ren et al. on the Dataset-M to represent the performance of *CNN*.

Observations from the reproduced results and reproduction processes. According to the reproduction process and the comparison results of performance aforementioned above, we make the following two key observations about the *NLP* and *CNN*:

(1) **Complex models.** The numerous parameters in the structure of the supervised approaches can bring huge challenges for understanding and applying them. For *NLP*, there are 14 parameters (including 5 binary parameters and 9 numerical parameters) that can affect the performance of a *NLP* model. It is a challenge to set up a group of suitable parameters that can have a good predictive capability on many test projects. Compared with *NLP*, the *CNN* model is more complicated. First, there are many processing layers (including input layer, embedding layer, convolution layer, pooling layer, fully connected layer, and output layer) to construct the prediction model. For example, the input layer accepts and transforms comment data for subsequent modeling, and the pooling layer is added to extract important features and to prevent overfitting. The combination of these layers can enhance the performance of a *CNN* model. However, there are many parameters in each of the above layers that need to be tuned carefully to build an effective prediction model. Second, various additional functions (e.g., activation function and loss function) are added to ensure the robustness of a *CNN* model. These functions play an important role in *CNN* and various neural network models. For example, the activation function can elevate the expression ability of a liner model, and the loss function is set to measure the difference between the output and actual values. Third, many detailed operations (e.g., batch normalization) and hyper parameters (e.g., the size of vocabulary and the number of filters) will also affect the performance of a *CNN* model. Last but not least, it cannot be ignored for the consumption of the computing resources of a *CNN* model. In total, the *NLP* and *CNN* are both complex in their structures.

(2) **High reproduction cost.** For complex approaches, there are huge parameter spaces to explore, large training time to spend, and many details to take into consideration. Therefore, it is difficult to reproduce these approaches exactly as the result reported by their original authors. Many studies [46, 86, 87] have confirmed this standpoint. For example, Beam et al. point out that there are great challenges to the reproducibility of machine learning models [87]. According to their findings, even if the same code is used by different researchers, they could obtain substantially different conclusions if some parameters are given different values. In our study, to obtain the performance that is consistent with the original study, we try our best to reproduce these approaches according to the descriptions from their papers. However, we do not get exactly the same results, since not all details (e.g., some parameters) about their approaches are reported explicitly and clearly. Although it is possible that our implementation is biased, the more important lesson we learned is the hard reproducibility of these approaches. This is especially true for a *CNN* model, where a trivial change in its parameter values can have a huge influence on the prediction performance. A recent empirical study on the replicability and reproducibility of deep learning models [46] also validates the standpoint. In particular, they summarized that the difficulties of reproduction or replication are from four factors: model stability, model convergence, out-of-vocabulary issue, and testing data size. Their empirical results show that these factors can influence the performance of a deep learning model substantially, and hence the reproduction cost is increased largely. Therefore, there are non-negligible challenges when applying these approaches in practice due to their high reproduction cost.

Due to the high reproduction cost of these approaches, it may hinder developers to apply them in practice. In other words, there are non-ignorable challenges for developers to implement these approaches that can achieve exactly the same performance reported by the original authors. For example, the performance of our implemented *CNN* is poor compared to the one reported by Ren

et al. Note that, although our reproduced *CNN* has a low performance, it does not mean that *CNN* is not effective. On the contrary, as stated in Reference [49], *CNN*-based models have been shown very useful in many research fields (e.g., bug localization [89, 90], defect prediction [91], and software community question retrieval [92, 93]). However, the low results of our implementation indicate that, as a practitioner, there are great challenges to achieve an equally excellent performance without their complete *CNN* source code.

Conclusion. *In summary, the reproduction difficulty of existing approaches can be classified into three categories: (1)* ***Low*** *(Pattern, TM, and Easy). These approaches can be used directly, since they can be easily implemented or have open-source code; (2)* ***Medium*** *(NLP). Although the code of the approach is not open source, we can reproduce it with an acceptable cost; (3)* ***High*** *(CNN). There are considerable challenges to reproduce this approach, since it contains complicated structure and many parameters.*

5.2 RQ2: Classification Performance Comparison

As mentioned in previous sections, *MAT* is an intuitive, simple, and easy-to-understand approach in identifying SATD comments. This is very helpful for practical use. Being an unsupervised approach, *MAT* can be applied to identify SATD directly in a target project without a need to train a model in advance. This is an obvious advantage over the (complex) supervised approaches. In this RQ, we want to know how well existing approaches perform in classification performance compared with *MAT*. If *MAT* exhibits a very competitive or even superior classification performance, then we can conclude: (1) the current progress in identifying SATD comments is not as good as they described; and (2) developers should apply a simple approach instead of using a complex approach in practice. To answer this research question, we use the same datasets to evaluate the classification performance of *MAT* as well as existing approaches under both MTO and OTO scenarios.

Under each scenario, we employ the BH-corrected p-values from the Wilcoxon signed-rank test to examine whether there is a statistically significant difference between *MAT* and each of existing approaches at the significance level of 0.05. Furthermore, we use the Cliff's delta (δ) to quantify the effect size of the difference. This enables us to know whether the magnitude of the difference in performances between two approaches is important from the viewpoint of practical application. By convention, the magnitude of the difference is considered trivial ($|\delta| < 0.147$), small (0.147–0.33), moderate (0.33–0.474), or large (>0.474).

(1) ***How effective are existing approaches in identifying SATD comments compared with MAT under many-to-one cross-project prediction scenario?***

Figures 5–6 report the performance comparison results between existing approaches and *MAT*. For each figure, the detailed performance scores of each approach and the improvement rates that *MAT* achieves are available in Reference [80]. On the 20 projects, on average *MAT* can achieve a precision of 0.815, a recall of 0.687, an F_1 value of 0.726, an ER of 0.926, and an RI of 19.483. This result is excellent for SATD comments identification. However, there are some exceptional cases. For instance, *MAT* has a recall lower than 0.500 in three projects (i.e., 0.205 for JEdit, 0.461 for Ant, and 0.351 for EMF). The reason for this is that the four task tags used in *MAT* rarely exist in these projects. According to the result, we have the following observations:

MAT vs. Pattern. In terms of precision, *MAT* has a better (higher) value on 12 out of 20 projects, with an average improvement of 2.6%. This indicates that *MAT* does not lose precision due to matching fewer words (i.e., only four tags). Considering the recall and F_1 of *MAT*, it has an average improvement of 1,188.4% and 643.3% compared with *Pattern*, respectively. The great improvement on recall indicates that *MAT* can identify more SATD comments than *Pattern*. This is

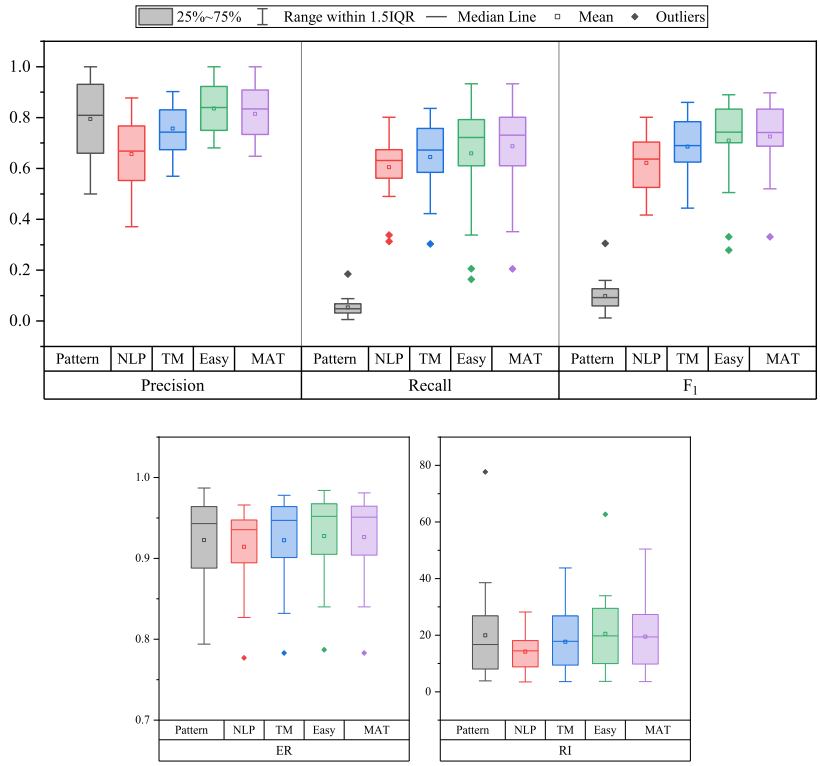


Fig. 5. The performance comparison based on Dataset-M and Dataset-G under the MTO scenario: *Pattern, NLP, TM*, and *Easy* vs. *MAT*.

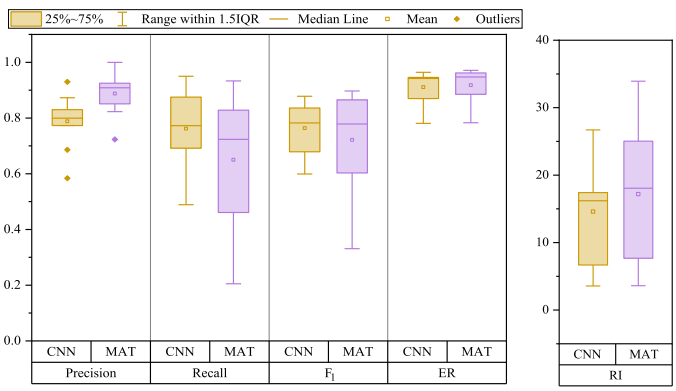


Fig. 6. The performance comparison based on Dataset-M under the MTO scenario: *CNN* vs. *MAT*.

mainly caused by the inherent imprecision of text matching via limited patterns in *Pattern* (i.e., keywords or phrase). In other words, natural language is more diversified, so a fixed pattern cannot match another expression of that pattern. For example, there is a pattern called "remove this code," according to Potdar et al. However, a SATD comment "todo remove this old implementation after it's demonstrated that it's not needed" (from ArgoUML project) cannot be identified by the pattern even though their semantics are very similar. Because of this reason, *Pattern* may leave out

Table 9. The Results from the Wilcoxon-singed Rank Test and the Cliff's Delta when Comparing *MAT* with Existing Approaches under the MTO Scenario (Bold Fonts Denote that the P-value is Less Than 0.05) (Abbr. N: Negligible; S: Small; M: Medium; L: Large)

	BH corrected p-value					Cliff's delta				
Indicators	Pattern	NLP	TM	Easy	CNN	Pattern	NLP	TM	Easy	CNN
Precision	0.519	**0.000**	**0.001**	**0.008**	**0.004**	0.080 (N)	**0.650 (L)**	0.330 (M)	−0.138 (N)	**0.620 (L)**
Recall	**0.000**	**0.003**	**0.018**	**0.020**	**0.008**	**1.000 (L)**	0.445 (M)	0.230 (S)	0.068 (N)	− 0.260 (S)
F1	**0.000**	**0.001**	**0.004**	0.400	0.366	**1.000 (L)**	**0.490 (L)**	0.265 (S)	0.015 (N)	− 0.060 (N)
ER	0.329	**0.000**	**0.001**	**0.008**	**0.008**	0.058 (N)	0.265 (S)	0.110 (N)	−0.063 (N)	0.230 (S)
RI	0.546	**0.000**	**0.001**	**0.008**	**0.004**	0.055 (N)	0.265 (S)	0.115 (N)	−0.060 (N)	0.220 (S)

many of SATD comments. Considering the two effort-aware indicators (i.e., ER and RI), *MAT* has an improvement of 0.8% and 16.1% in terms of median value of them, respectively, which shows a slight advantage over *Pattern*.

MAT vs. NLP. On average, *MAT* performs better than *NLP* with respect to all five indicators. In terms of precision, *MAT* has an average improvement of 24.2% compared with *NLP*. Note that, the precision of *MAT* is higher than that of *NLP* in all target projects. When looking at recall, we find that *MAT* can identify more SATD comments in the majority (18 out of 20) of projects, with an average improvement of 13.6%. In terms of F_1 score, on average, *MAT* has an improvement of 16.7% compared with *NLP*. Considering ER and RI, *MAT* has an average improvement of 1.3% and 37.2%, respectively.

MAT vs. TM. Compared with *TM, MAT* performs better in the majority of projects. On average, *MAT* has an improvement of 7.6%, 6.6%, 5.8%, 0.4%, and 10.2% in terms of precision, recall, F_1, ER, and RI, respectively. In particular, *MAT* achieves promising improvements in Ant (i.e., 16.4%) and Tomcat (i.e., 19.0%) in terms of F_1 and in Ant (i.e., 31.3%) and EMF (i.e., 25.9%) in terms of RI.

MAT vs. Easy. *MAT* has a very similar effectiveness compared with *Easy*. The reason is that *MAT* and *Easy* use very similar patterns (e.g., "fixme," "todo") to identify SATD comments. In particular, on average, *MAT* shows 4.2% improvement in recall and 2.3 improvement in F_1. However, on average, *MAT* shows 2.5% reduction in precision, 0.1% reduction in ER, and 4.8% reduction in RI, respectively.

MAT vs. CNN. Compared with *CNN*, we find that *MAT* can achieve an average improvement of 12.6% in terms of precision. Note that, the precision of *MAT* is higher in all target projects. In particular, *MAT* can achieve a large improvement on many projects (e.g., 49.0% for Ant). As an unsupervised approach, this is a great advantage for accurate identification of SATD comments. The comparison between *MAT* and *CNN* in terms of recall shows that *MAT* can identify fewer SATD comments than *CNN*. This is understandable, because *MAT* cannot identify those SATD comments that are not marked by task tags (e.g., "Nasty hardcoded values" from JEdit). However, *CNN* can identify these comments by semantic analysis. The situation is particularly evident for JEdit, on which *MAT* decreases the recall value of 58.1% compared with *CNN*. According to the median F_1, *MAT* achieves a value of 0.779 that is very close to that (0.783) of *CNN*. In particular, *MAT* has higher ER and RI than *CNN*, regardless of whether the mean or median value is considered. This reveals that *MAT* is more cost-effective than *CNN* in identifying SATD when compared with a random model: On the one hand, *MAT* leads to a higher inspection effort reduction in finding the same number of SATD; and on the other hand, *MAT* leads to a higher recall increase when inspecting the same number of comments.

Table 9 reports the results from statistical analyses under the MTO prediction scenario. In the view of the statistical test, we make the following observations: First, *MAT* is significantly better

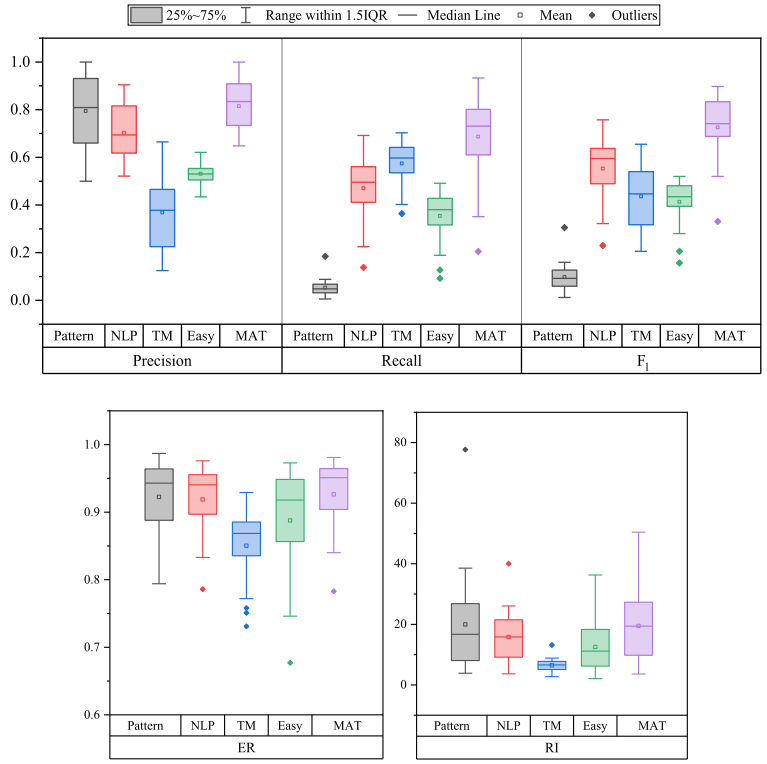


Fig. 7. The performance comparison based on Dataset-M and Dataset-G under the OTO scenario: *Pattern, NLP, TM*, and *Easy* vs. *MAT.*

than *Pattern* (the effect size is large) when recall and F_1 is considered. Second, when compared with *NLP* and *TM, MAT* exhibits a significantly better performance for all indicators (the effect size is non-negligible in most cases). Third, although there are significant differences between *Easy* and *MAT* in precision, recall, ER, and RI, the corresponding effect sizes are negligible. Finally, *MAT* is significantly worse than *CNN* in terms of recall (the effect size is small) but is significantly better than *CNN* in terms of precision, ER, and RI (the effect sizes are small to large).

Combining the above results, we find that, under the MTO prediction scenario: *MAT* has an outstanding classification performance compared with the unsupervised approach *Pattern*; in addition, *MAT* is very competitive or even superior to all the supervised approaches (*NLP, TM, Easy*, and *CNN*).

(2) ***How effective are existing approaches in identifying SATD comments compared with MAT under one-to-one cross-project prediction scenario?***

Figures 7–8 report the performance comparison results under the OTO prediction scenario. Since *MAT* and *Pattern* are unsupervised approaches, they are not influenced by the training data. As a result, they have the same performance under the MTO and OTO scenarios. According to the results, we have the following observations:

MAT vs. NLP. In terms of precision, *MAT* has a better (higher) value on 18 out of 20 projects, which has an average improvement of 15.9%. Considering the recall, *MAT* performs better than

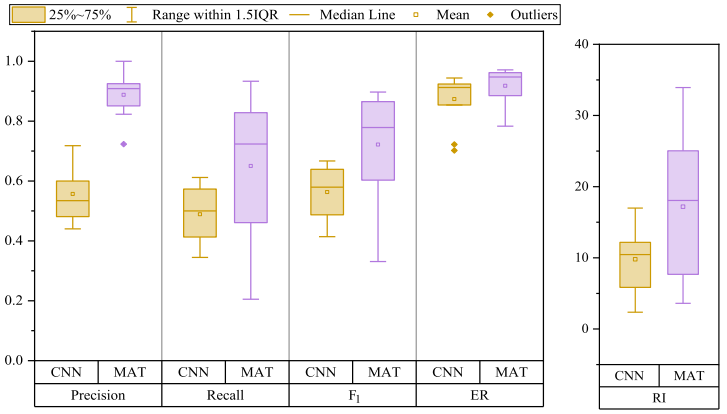


Fig. 8. The performance comparison based on Dataset-M under the OTO scenario: *CNN* vs. *MAT*.

NLP in all 20 projects, which has an average improvement of 45.9%. Similarly, we can see that *MAT* achieves a better F_1 than *NLP* in all the 20 projects, with an average improvement of 31.2%. As for ER and RI, *MAT* is better than *NLP* for most projects. Note that, the average precision of *NLP* increases under the OTO prediction scenario (0.656 under MTO vs. 0.703 under OTO). Meanwhile, the average F_1 under OTO (0.553) is slightly lower than that under MTO (0.621) when the number of projects for training are reduced largely (from 19 projects to one project). The above results show that *NLP* can maintain a good performance even with a relatively small training dataset when identifying SATD comments, which is consistent with that found in Reference [17]. Also, we find that *NLP* has similar average ER and RI values under OTO and MTO.

MAT vs. TM. Compared with *TM*, on average, *MAT* has an improvement of 120.7%, 19.6%, 66.4%, 8.9%, and 201.7% in terms of precision, recall, F_1, ER, and RI, respectively. In particular, for precision, ER, and RI, *MAT* is better than *TM* in all projects. We can see that, under the OTO scenario, the average precision of *TM* decreases largely (0.757 under MTO vs. 0.369 under OTO). This indicates that, the precision of *TM* is very sensitive to training set size. Many false positive instances will be introduced by *TM* when the training set is small. As for the average recall, there is no large difference under the two scenarios (0.644 under MTO vs. 0.574 under OTO), which means that the recall of *TM* is not sensitive to training set size. Note that, there is a small reduction when looking at the average ER (0.922 under MTO vs. 0.850 under OTO), while there is a large reduction for the average RI (17.673 under MTO vs. 6.458 under OTO).

MAT vs. Easy. Under the OTO scenario, *MAT* performs better than *Easy* on all the projects. On average, *MAT* has an improvement of 53.4%, 93.9%, 75.6%, 4.4%, and 55.6% in terms of precision, recall, F_1, ER, and RI, respectively. For *Easy*, there is a large performance drop when comparing OTO with MTO, regardless of which performance indicator is considered. The main reason is that the patterns extracted from a single project are not representative. For instance, *Easy* only extracts one pattern (i.e., "xxx") from Ant and it extracts nothing from Hive. In this case, it is not possible to make accurate predictions for a target project by matching these poor patterns. To facilitate external examination, we make all the extracted patterns online.[9]

MAT vs. CNN. *MAT* performs better than *CNN*, regardless of which performance indicator is considered. Compared with *CNN*, *MAT* can achieve an average improvement of 59.5% in terms of precision. Different from the MTO scenario, *MAT* has a better average recall than *CNN* under the

[9]https://github.com/Naplues/MAT/blob/master/result/pattern.txt.

Table 10. The Results from the Wilcoxon-singed Rank Test and the Cliff's Delta when Comparing *MAT* with Existing Approaches under the OTO Scenario (Bold Fonts Denote that the P-value is Less than 0.05) (Abbr. N: Negligible; S: Small; M: Medium; L: Large)

	BH corrected p-value					Cliff's delta				
Indicators	Pattern	NLP	TM	Easy	CNN	Pattern	NLP	TM	Easy	CNN
Precision	0.519	**0.000**	**0.000**	**0.000**	**0.002**	0.080 (N)	**0.528 (L)**	**0.990 (L)**	**1.000 (L)**	**1.000 (L)**
Recall	**0.000**	**0.000**	**0.001**	**0.000**	0.066	**1.000 (L)**	**0.700 (L)**	**0.570 (L)**	**0.835 (L)**	**0.510 (L)**
F1	**0.000**	**0.000**	**0.000**	**0.000**	**0.007**	**1.000 (L)**	**0.695 (L)**	**0.870 (L)**	0.913 (M)	**0.660 (L)**
ER	0.301	**0.000**	**0.000**	**0.000**	**0.002**	0.058 (N)	0.185 (S)	**0.675 (L)**	0.358 (M)	**0.490 (L)**
RI	0.546	**0.000**	**0.000**	**0.000**	**0.002**	0.055 (N)	0.185 (S)	**0.675 (L)**	0.360 (M)	**0.480 (L)**

OTO scenario, with an average improvement of 33.0%. Considering the F_1 score, *MAT* performs better than *CNN* in most (8 out of 10) projects, with an average improvement of 28.1%. This means that *MAT* is better than the latest supervised approach when the training set is small. Note that, *CNN* achieves a low average recall (0.489) under the OTO scenario compared with that (0.762) under the MTO scenario. The large degradation reveals that many SATD comments cannot be identified by *CNN* when the training data are limited. Considering the effort-aware indicators, there is also a large reduction for RI (14.595 under MTO vs. 9.792 under OTO).

Table 10 reports the results from statistical analyses under the OTO prediction scenario. In the view of the statistical test, we make the following observations: In terms of the majority of comparisons, *MAT* is significantly better (p-value < 0.05) than existing approaches (the effect sizes are medium to large in most cases).

Combining the above results, we can see that, under the OTO scenario, the effectiveness of existing supervised approaches declines in various degrees. Overall, *MAT* outperforms all the existing approaches in identifying SATD comments under the scenario that only limited training data are available.

Conclusion. *In summary, MAT exhibits a very competitive or even superior overall performance to existing approaches when identifying self-admitted technical debt. This indicates that, if the prediction performance is the goal, then the current progress in identifying SATD comments is not being achieved as it might have been envisaged in the literature.*

5.3 RQ3: Difference in Correct Classification Results

In RQ2, we calculate an overall performance score (such as precision, recall, and F_1) to represent how well an approach correctly predicts SATD comments by considering the level of incorrect predictions made. However, only the overall performance score cannot illustrate the classification result of a single comment classified by different approaches. According to Reference [74], the overall performance can hide a variety of differences in the defects that each classifier predicts. Similarly, for SATD classification, it is important to investigate the difference in individual SATD and non-SATD comments identified by various approaches. Studying the difference will help us understand the classification characteristics of various approaches such as the overlapped instances and unique instances identified by different approaches. As a result, we can analyze the advantages and disadvantages of each approach. More specifically, in our study, we investigate the difference of the correct classification results among *MAT, NLP, TM*, and *Easy*. Note that, in RQ3, we do not consider the *Pattern* and the *CNN*. The reasons are the following: For the former approach, the recall is too low to conduct the analysis; for the latter approach, its code is not shared online (it is hard to implement it by ourselves to exactly reproduce Ren et al.'s results due to many

parameters involved), and their original classification results (i.e., which instances are classified as SATD or non-SATD) are also not publicly available.

According to Maldonado et al.'s study [17] and Huang et al.'s study [10], both the *NLP, TM*, and *Easy* are evaluated under the MTO scenario. Consistent with their studies, we investigate RQ3 under the MTO scenario. First, for each project, we obtain the true positive SATD comments (and true negative SATD comments) that four approaches (i.e., the *MAT, NLP, TM*, and *Easy*) can identify. Then, we use a set diagram to describe the difference among four approaches in terms of the specific SATD comments (and non-SATD comments) that each approach identifies and does not identify. Second, we leverage McNemar's test [84] to compare if there exists a statistical significance for the prediction errors between any supervised approach and *MAT*. According to Zhang et al. [88], the calculation of McNemar's test between two approaches a1 and a2 is based on a contingency matrix shown as follows:

$$\begin{bmatrix} N_{cc} & N_{cw} \\ N_{wc} & N_{ww} \end{bmatrix}, \tag{10}$$

where N_{cc} (or N_{ww}) denotes the number of instances that both approaches make correct (or wrong) predictions; N_{cw} (or N_{wc}) denotes the number of instances that *a1* makes correct (or wrong) prediction, but a2 produces wrong (or correct) prediction. After obtaining the contingency matrix, we also apply R function mcnemar.exact from R package *exact2* × *2* to conduct McNemar's test as used in Reference [88]. In addition, we use odds ratio (OR) [85] to measure the effect size of this kind of difference. The definition of OR is shown as follows:

$$OR = \frac{N_{cw} + 1}{N_{wc} + 1}. \tag{11}$$

Note that, we add 1 to both N_{cw} and N_{wc} to avoid generating a divisor equal to 0. An OR > 1 indicates that a2 makes more wrong predictions than a1 and vice versa. An OR = 1 indicates that a1 and a2 make the same number of wrong predictions.

Figure 9 reports the number of true positive instances (TPs) identified by *NLP, TM, Easy*, and *MAT*, while Figure 10 reports the number of true negative instances (TNs) identified by *NLP, TM, Easy*, and *MAT*. In each figure, the orange, blue, red, and green ellipses, respectively, denote the number of SATD comments that are correctly classified by *NLP, Easy, MAT*, and *TM*. Each Venn diagram (corresponds to a project) in Figure 9 (or Figure 10) shows the number of overlaps of TP (or TN) instances among the prediction results of each approach. For example, for the project Ant shown in Figure 9, there are 14 same SATD comments correctly identified by all four approaches, 10 same SATD comments are identified correctly by only both *NLP* and *MAT*, and 6 unique SATD comments are correctly identified by only *TM*.

From Figure 9, we can see that, the SATD comments correctly classified by *NLP, TM, Easy*, and *MAT* are largely overlapped. Specifically, all four approaches agree on 3,055 (63.02%) out of 4,848 true SATD comments. These overlapped SATD comments accounts for 79.66% of the 3,835 SATD comments correctly identified by *NLP*, 75.98% of the 4,021 SATD comments correctly identified by *TM*, 70.12% of the 4,357 SATD comments correctly identified by *Easy*, and 68.62% of the 4,452 SATD comments correctly identified by *MAT*. This indicates that *MAT* can identify more SATD comments than *NLP, TM*, and *Easy*.

According to Figure 10, it can be seen that, most of the non-SATD comments correctly classified by *NLP, TM, Easy*, and *MAT* are also overlapped. In total, all four approaches agree on 109,936 (98.15%) out of 112,011 true non-SATD comments. These overlapped non-SATD comments account for 99.36% of the non-SATD comments correctly identified by *NLP* (110,644), 98.76% of the non-SATD comments correctly identified by *TM* (111,311), 98.52% of the non-SATD comments correctly

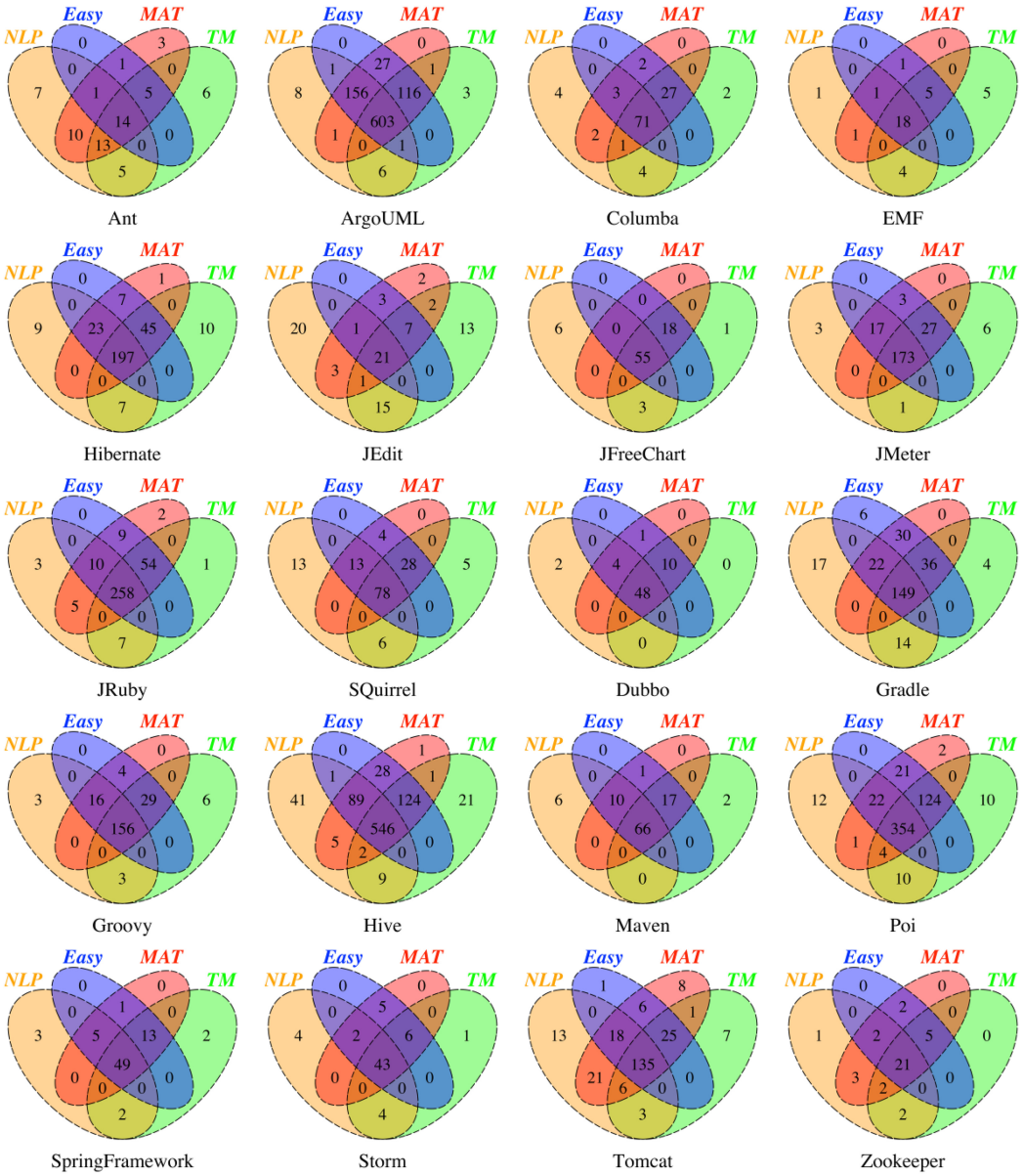


Fig. 9. Set diagrams for the number of true positive instances identified by *MAT* and three supervised approaches (*NLP*, *TM*, and *Easy*) on 20 projects.

identified by *Easy* (111,583), and 98.62% of the non-SATD comments correctly identified by *MAT* (111,479). This indicates that most non-SATD comments can also be correctly identified by all four approaches. Note that, there is almost no difference among each approach in terms of the number of the identified correct non-SATD comments. Compared with the overlapped non-SATD comments, the unique non-SATD comments identified by each approach are very few (i.e., the percentage is lower than 1%) and hence there are no large differences among the four approaches.

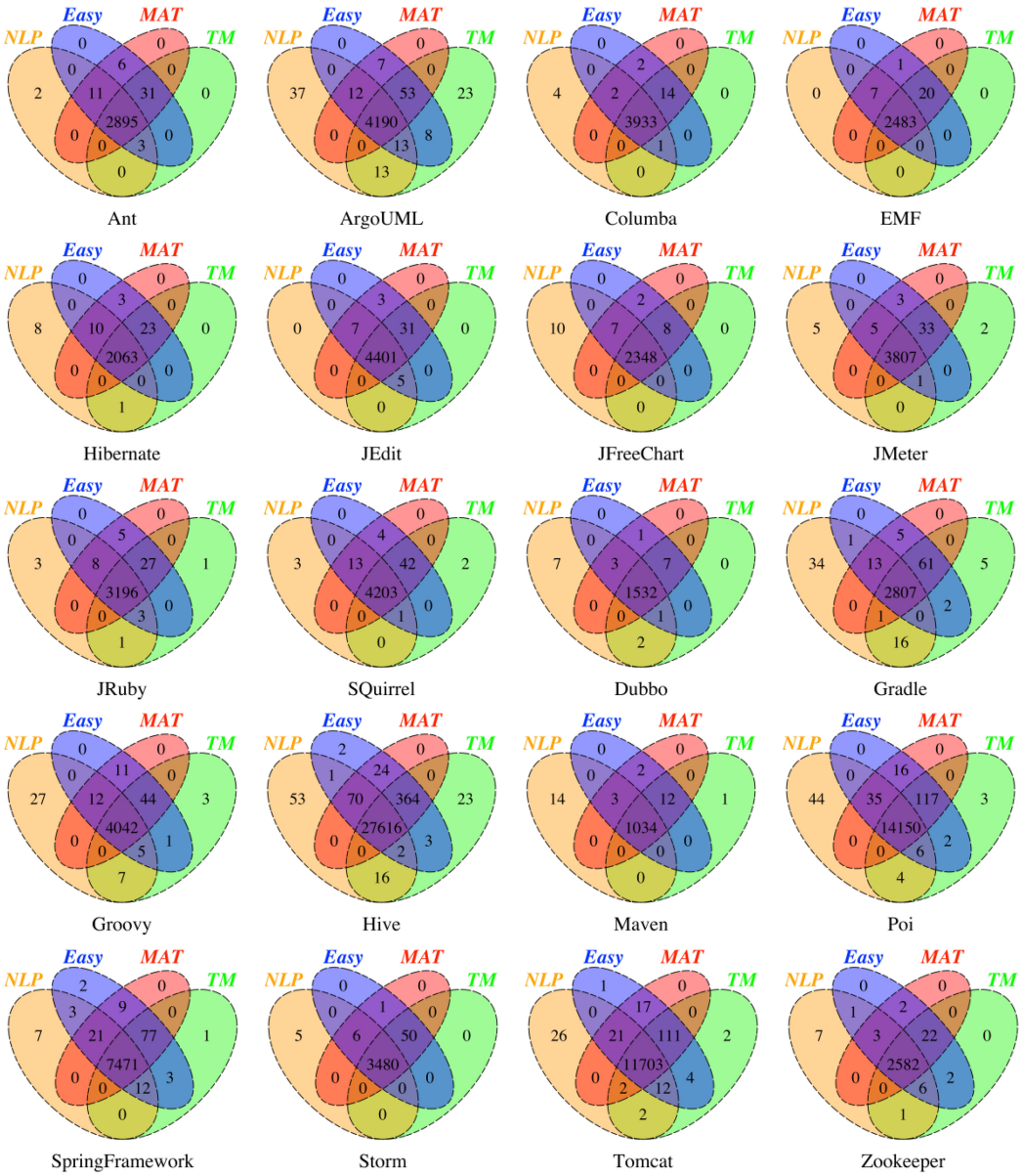


Fig. 10. Set diagrams for the number of true negative instances identified by *MAT* and three supervised approaches (*NLP*, *TM*, and *Easy*) on 20 projects.

Table 11 reports the comparison between *MAT* and each supervised approach in the perspective of TP instances. For each supervised approach, the first column reports the number of true positive instances, the second column (i.e., %Hit) reports the percentage of true positive instances that can also be correctly identified by *MAT*, and the third column (i.e., %Over) reports the ratio (in percentage) of the number of instances only correctly identified by *MAT* to the number of true

Table 11. The Comparison between *MAT* and Each Supervised Approach in the Perspective of TP Instances (%Hit = $|TP_{Supervised} \cap TP_{MAT}| / |TP_{Supervised}|$; %Over = $|TP_{MAT} - TP_{Supervised}| / |TP_{Supervised}|$)

Project	MAT v.s. NLP			MAT v.s. TM			MAT v.s. Easy		
	TP_{NLP}	%Hit	%Over	TP_{TM}	%Hit	%Over	TP_{Easy}	%Hit	%Over
Ant	50	76.00%	18.00%	43	74.42%	34.88%	21	100.00%	123.81%
ArgoUML	776	97.94%	18.56%	730	98.63%	25.21%	904	99.78%	0.22%
Columba	85	90.59%	34.12%	105	94.29%	6.67%	103	100.00%	2.91%
EMF	25	80.00%	24.00%	32	71.88%	9.38%	25	100.00%	4.00%
Hibernate	236	93.22%	22.46%	259	93.44%	11.97%	272	100.00%	0.37%
JEdit	61	42.62%	22.95%	59	52.54%	15.25%	32	100.00%	25.00%
JFreeChart	64	85.94%	28.13%	77	94.81%	0.00%	73	100.00%	0.00%
JMeter	194	97.94%	15.46%	207	96.62%	9.66%	220	100.00%	0.00%
JRuby	283	96.47%	22.97%	320	97.50%	8.13%	331	100.00%	2.11%
SQuirrel	110	82.73%	29.09%	117	90.60%	14.53%	123	100.00%	0.00%
Dubbo	54	96.30%	20.37%	58	100.00%	8.62%	63	100.00%	0.00%
Gradle	202	84.65%	32.67%	203	91.13%	25.62%	243	97.53%	0.00%
Groovy	178	96.63%	18.54%	194	95.36%	10.31%	205	100.00%	0.00%
Hive	693	92.64%	22.22%	703	95.73%	17.50%	788	99.87%	1.14%
Maven	82	92.68%	21.95%	85	97.65%	12.94%	94	100.00%	0.00%
Poi	403	94.54%	36.48%	502	96.02%	9.16%	521	100.00%	1.34%
SpringFramework	59	91.53%	23.73%	66	93.94%	9.09%	68	100.00%	0.00%
Storm	53	84.91%	20.75%	54	90.74%	12.96%	56	100.00%	0.00%
Tomcat	196	91.84%	20.41%	177	94.35%	29.94%	185	99.46%	19.57%
Zookeeper	31	90.32%	22.58%	30	93.33%	23.33%	30	100.00%	16.67%
Average	**191.75**	**87.97%**	**23.77%**	**201.05**	**90.65%**	**14.76%**	**217.85**	**99.83%**	**9.86%**
Total	**3835**			**4021**			**4357**		

positive instances. For example, for *NLP* on the Ant project, of the 50 true positive instances, 76% of the instances (i.e., 38 instances) are also correctly identified by *MAT*; in addition, *MAT* identifies 9 additional true positive (i.e., 18.0%) instances that are not identified by *NLP*.

On average, *MAT* can identify 87.97% of true positive instances identified by *NLP*, 90.65% of true positive instances identified by *TM*, and 99.83% of true positive instances identified by *Easy*. This indicates that most of the true positive instances identified by each supervised approach can also be correctly identified by *MAT*. Considering the %Over of *MAT* compared with each supervised approach, we can find that *MAT* can identify 23.77% distinct true positive instances compared with *NLP*, 14.76% distinct true positive instances compared with *TM*, and 9.86% distinct true positive instances compared with *Easy* on average. This means that the supervised approaches can misclassify some comments having task tags (they are SATDs) as non-SATD comments, especially for *NLP*. This is an obvious limitation of the supervised approaches, since these true positive instances can be easily identified by *MAT* in practice.

Table 12 reports the comparison between *MAT* and each supervised approach in the perspective of TN instances. We can see that, the true negative instances identified by *MAT* are very similar to those identified by the supervised approaches. Specifically, *MAT* can identify 99.51% of true negative instances identified by *NLP*, 99.78% of true negative instances identified by *TM*, and 99.90% of true negative instances identified by *Easy*. Meanwhile, *MAT* can identify 1.08% distinct true negative instances compared with *NLP*, 0.37% distinct true negative instances compared with *TM* on average. This means that, *MAT* has the almost same ability to identify non-SATD comments as the supervised approaches.

Table 12. The Comparison between *MAT* and Each Supervised Approach in the Perspective of TN Instances (%Hit = $|TN_{Supervised} \cap TN_{MAT}| / |TN_{Supervised}|$; %Over = $|TN_{MAT} - TN_{Supervised}| / |TN_{Supervised}|$)

Project	MAT v.s. NLP			MAT v.s. TM			MAT v.s. Easy		
	TN_{NLP}	%Hit	%Over	TN_{TM}	%Hit	%Over	TN_{Easy}	%Hit	%Over
Ant	2911	99.79%	1.31%	2929	99.90%	0.58%	2946	99.9%	0.00%
ArgoUML	4265	98.50%	1.41%	4300	98.67%	0.44%	4283	99.5%	0.00%
Columba	3940	99.90%	0.41%	3948	99.97%	0.10%	3952	100.0%	0.00%
EMF	2490	100.00%	0.80%	2503	100.00%	0.32%	2511	100.0%	0.00%
Hibernate	2082	99.62%	1.20%	2087	99.95%	0.62%	2099	100.0%	0.00%
JEdit	4413	99.91%	0.79%	4437	99.89%	0.23%	4447	99.9%	0.00%
JFreeChart	2365	99.62%	0.38%	2356	100.00%	0.38%	2365	100.0%	0.00%
JMeter	3818	99.79%	0.89%	3843	99.92%	0.21%	3849	100.0%	0.00%
JRuby	3211	99.81%	1.00%	3228	99.85%	0.40%	3239	99.9%	0.00%
SQuirrel	4220	99.91%	1.09%	4248	99.93%	0.40%	4263	100.0%	0.00%
Dubbo	1545	99.42%	0.52%	1542	99.81%	0.26%	1544	99.9%	0.00%
Gradle	2872	98.19%	2.30%	2892	99.20%	0.62%	2889	99.9%	0.03%
Groovy	4093	99.00%	1.29%	4102	99.61%	0.56%	4115	99.9%	0.00%
Hive	27758	99.70%	1.40%	28024	99.84%	0.34%	28082	100.0%	0.00%
Maven	1051	98.67%	1.33%	1047	99.90%	0.48%	1051	100.0%	0.00%
Poi	14239	99.60%	0.90%	14282	99.89%	0.36%	14326	99.9%	0.00%
SpringFramework	7514	99.69%	1.10%	7564	99.79%	0.40%	7598	99.7%	0.00%
Storm	3491	99.91%	1.49%	3530	100.00%	0.20%	3537	100.0%	0.00%
Tomcat	11766	99.70%	1.10%	11836	99.83%	0.32%	11869	99.9%	0.02%
Zookeeper	2600	99.38%	0.88%	2613	99.66%	0.19%	2618	99.7%	0.00%
Average	**5532.2**	**99.51%**	**1.08%**	**5565.55**	**99.78%**	**0.37%**	**5579.15**	**99.90%**	**0.00%**
Total	**110644**			**111311**			**111583**		

In summary, the SATD comments (or non-SATD comments) correctly identified by existing approaches are highly overlapped with those identified by *MAT*. This finding is surprising, given the fact that existing supervised approaches tackle the prediction task using very different techniques.

Table 13 reports the detailed results of McNemar's test and the corresponding effect size (i.e., odd ratio) from the comparison of prediction errors on actual SATD and non-SATD comments. Note that, the p-values of McNemar's test have been corrected according to Benjamini et al.'s method [47] that can control the false discovery. A p-value < 0.05 indicates that there is a significant difference on the prediction errors. Meanwhile, the effect size (OR) is the ratio of the number of instances that only the compared supervised approach makes a wrong prediction to the number of instances that only *MAT* makes a wrong prediction. Therefore, an OR = 1 means that two approaches make the same amount of wrong predictions. An OR > 1 indicates that *MAT* makes fewer wrong predictions than the compared supervised approach and vice versa.

According to Table 13, for the prediction error on actual SATD comments, *MAT* is significantly different (p-value < 0.05) from *NLP, TM*, and *Easy* on 65%, 45%, and 30% of 20 projects, respectively. This shows that *MAT* and other approaches can make different wrong predictions for actual positive instances on many projects. For the comparison between *MAT* and *NLP*, the majority of ORs are greater than 1, which means that *NLP* makes more wrong predictions than *MAT* in these projects. We can observe a similar result for the comparison between *MAT* and *TM* and between *MAT* and *Easy*. The above results reveal that *MAT* makes fewer wrong predictions compared with the investigated three supervised approaches. Meanwhile, we find that, for the prediction error on actual non-SATD comments, *MAT* is significantly different (p-value < 0.05) from *NLP, TM*, and *Easy* on 60%, 45%, and 30% of 20 projects, respectively. Furthermore, *MAT* makes fewer wrong predictions (i.e., OR > 1) compared with *NLP* and *TM* on most projects but makes more wrong predictions

Table 13. Comparison of Prediction Errors on Actual SATD Comments and Non-SATD Comments under the MTO Scenario (the BH-Corrected P-values of McNemar's Test and Odd Ratio)

Target Project	Prediction errors on actual SATD						Prediction errors on actual non-SATD					
	MAT v.s. NLP		MAT v.s. TM		MAT v.s. Easy		MAT v.s. NLP		MAT v.s. TM		MAT v.s. Easy	
	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR
Ant	0.885	0.769	0.777	1.333	**0.000**	27.000	**0.000**	6.333	**0.008**	4.500	0.375	0.250
ArgoUML	**0.000**	8.529	**0.000**	16.818	1.000	1.000	1.000	0.953	**0.000**	0.345	**0.000**	0.045
Columba	**0.003**	3.333	1.000	1.143	0.375	4.000	0.055	2.833	0.523	2.500	1.000	0.500
EMF	1.000	1.167	0.231	0.400	1.000	2.000	**0.000**	22.000	**0.018**	9.000	1.000	1.000
Hibernate	**0.000**	3.176	0.112	1.778	1.000	2.000	**0.016**	2.700	**0.006**	7.000	1.000	1.000
JEdit	**0.011**	0.417	**0.009**	0.345	**0.022**	9.000	**0.000**	5.833	0.442	1.833	0.117	0.167
JFreeChart	0.203	1.900	0.203	0.200	1.000	1.000	1.000	1.000	**0.011**	10.000	1.000	1.000
JMeter	**0.000**	6.200	**0.048**	2.625	1.000	1.000	**0.000**	5.286	0.358	2.250	1.000	0.500
JRuby	**0.000**	6.000	**0.009**	3.000	**0.041**	8.000	**0.000**	4.125	0.175	2.333	0.375	0.250
SQuirrel	0.158	1.650	0.493	1.500	1.000	1.000	**0.000**	9.400	**0.008**	4.500	1.000	0.500
Dubbo	**0.049**	4.000	0.112	6.000	1.000	1.000	1.000	0.818	1.000	1.250	1.000	0.500
Gradle	**0.002**	2.094	**0.000**	2.789	0.065	0.143	0.333	1.288	0.710	0.792	0.815	0.500
Groovy	**0.000**	4.857	0.112	2.100	1.000	1.000	0.214	1.400	0.481	1.412	0.063	0.143
Hive	**0.000**	2.981	**0.000**	4.000	**0.049**	5.000	**0.000**	5.329	**0.000**	2.111	**0.018**	0.111
Maven	**0.049**	2.714	**0.049**	4.000	1.000	1.000	1.000	1.000	0.355	3.000	1.000	1.000
Poi	**0.000**	6.435	**0.007**	2.238	**0.041**	8.000	**0.000**	2.436	**0.000**	3.250	**0.018**	0.111
SF	0.112	2.500	0.983	1.400	1.000	1.000	**0.000**	3.783	0.105	1.824	**0.000**	0.048
Storm	0.883	1.333	0.989	1.333	1.000	1.000	**0.000**	8.667	**0.035**	8.000	1.000	1.000
Tomcat	**0.007**	2.412	**0.000**	4.909	**0.000**	18.500	**0.000**	3.146	0.053	1.857	**0.003**	0.167
Zookeeper	0.493	2.000	0.276	2.667	0.112	6.000	0.333	1.563	0.578	0.600	**0.011**	0.100
#of significance(%)	**13/20 (65%)**		**9/20 (45%)**		**6/20 (30%)**		**12/20 (60%)**		**9/20 (45%)**		**6/20 (30%)**	

than *Easy*. According to the results, we can observe that: (1) *MAT* is more reliable than *NLP* and *TM* in predicting SATD comments as well as in predicting non-SATD comments; and (2) *MAT* is more reliable than *Easy* in predicting SATD comments but not in predicting non-SATD comments. For practitioners, it is a matter to predict SATD comments rather than non-SATD comments. In this sense, *MAT* is more helpful for practitioners than *NLP*, *TM*, and *Easy*.

Conclusion. *In summary, most SATD (non-SATD) comments correctly identified by the investigated supervised approaches (i.e., NLP, TM, and Easy) can also be correctly identified by MAT. Furthermore, MAT makes few wrong predictions on actual SATD comments. In this sense, for practitioners, it appears reasonable to consider applying (simple) MAT to replace the investigated (complex) existing supervised approaches.*

5.4 RQ4: Weakness and Possible Improvement

According to the results in RQ2 and RQ3, supervised approaches do not exhibit a superior performance compared with the simple unsupervised approach *MAT*. This finding is very surprising, as supervised approaches leverage the knowledge learned from labelled data, but *MAT* does not. In this research question, we aim to analyze for supervised approaches the weaknesses in SATD identification and explore whether there is a simple strategy to improve their effectiveness by incorporating the idea of *MAT*.

(1) ***Why are the supervised approaches not very outstanding compared with MAT?***

One of the assumptions of supervised approaches is that the testing data has a similar data distribution with the training project data. In our context, the training and testing data are from different projects, which may exhibit different data distribution. Consequently, the phenomena of concept drift may exist, which can seriously affect the validity of a prediction model [65, 73].

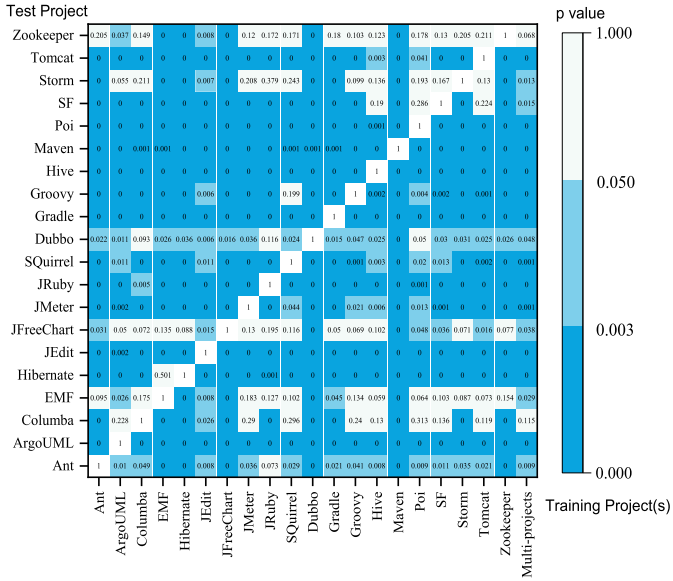


Fig. 11. The concept drift detection of *NLP*.

In the following, we employ **WTSD (Wilcoxon rank sum test drift detector)** to examine to what extent the investigated supervised approaches suffer from the concept drift phenomena [73]. For a given supervised model, WTSD uses the Wilcoxon rank sum test, a nonparametric test, to examine whether two independent samples (i.e., the predictions on two projects in our context) come from populations with the same distribution. Based on the p-value, one of the following cases will happen [39]: Yes (a concept drift detected) if p-value < 0.003, Warning (warning happened) if $0.003 \leq$ p-value < 0.05, or No (no concept drift detected) if p-value ≥ 0.05. Again, p-values are corrected using the BH method to control for false discovery [47].

Figures 11–13 report the results of concept drift detection for the three state-of-the-art supervised approaches. In each heat map, each row represents a test project, and each column represents a training project. Note that, the last column (i.e., Multi-projects) represents that the comments from all the other projects are merged to obtain the training data. Therefore, we have 420 (21×20) training-test pairs for each supervised approach. Each cell reports the BH-corrected p-value and its background color indicates the result of the WTSD for the corresponding training-test pair: blue background denotes a concept drift (Yes, $p < 0.003$), light blue background denotes a warning for concept drift (Warning, $0.003 \leq p < 0.05$), and white background denotes no concept drift (No, $p \geq 0.05$). From Figures 11–13, we have the following observations:

- For *NLP*, 20% of results (85 pairs) do not exhibit concept drift, 14% of results (59 pairs) may have concept drift, and the majority (66%) of results provide an evidence of concept drift.
- For *TM*, 20% of results (84 pairs) do not exhibit concept drift, 9% of results (39 pairs) may have concept drift, and the majority (71%) of results provide an evidence of concept drift.
- For *Easy*, 40% of results (168 pairs) do not exhibit concept drift, 6% of results (27 pairs) may have concept drift, and the majority (54%) of results provide an evidence of concept drift.

As can be seen, all the three investigated supervised approaches suffer from considerable concept drift (especially true for *TM*). This means that these supervised approaches can perform well

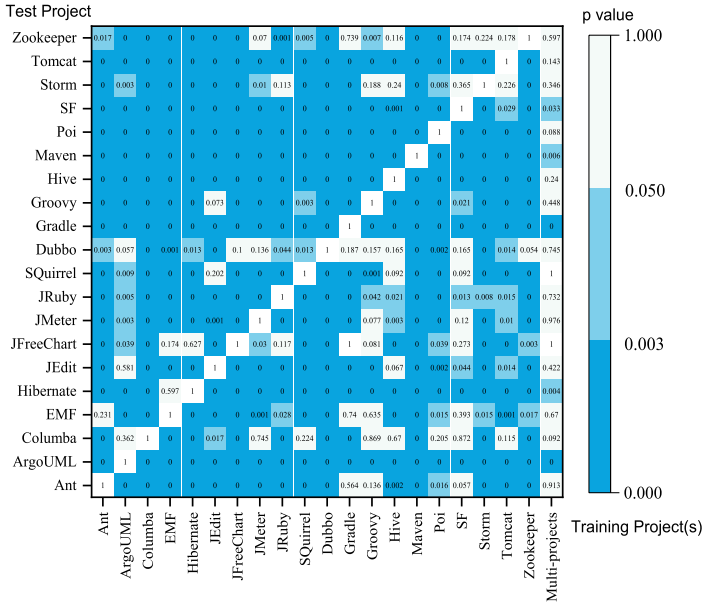


Fig. 12. The concept drift detection of *TM*.

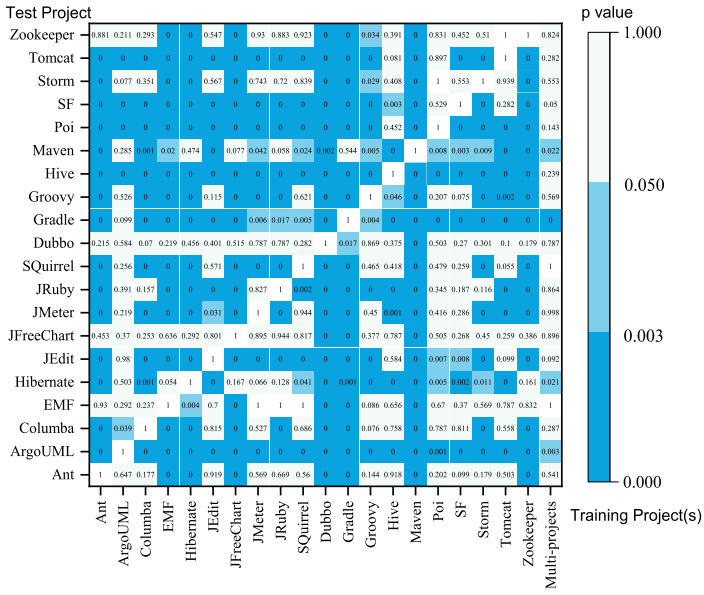


Fig. 13. The concept drift detection of *Easy*.

on the training data, but they do not achieve an equally good performance on the test data. Note that, for *TM* and *Easy*, the numbers of pairs that contain concept drift have decreased (see the last columns in Figure 12 and Figure 13) when multi-projects are used for training a prediction model. This can partly explain the reason why *TM* and *Easy* can perform better than *NLP* under MTO scenario (in RQ2). We can also conclude that the performances of *TM* and *Easy* under MTO scenario

Table 14. Example SATD Comments that Cannot Be Identified by the Supervised Approaches

Type of FN instances	Examples

The FNs contain task tags	// TODO Is this a valid operation on a timestamps cache? - *[from Hibernate by NLP]*
	// FIXME: need a locale as well as a timezone - *[from JFreeChart by NLP]*
	// XXX ENCODING - this only works if encoding is UTF8-compat -*[from Tomcat by NLP]*
	// Hack via a static since we can't pass an instance in the test. -*[from Tomcat by NLP]*
	// TODO: This shouldn't depend on the current project - *[from ArgoUML by TM]*
	// TODO Should this be OK ? - *[from JMeter by TM]*
	// TODO: This does not work correctly. None of the partitions is created - *[from Hive by TM]*
	// XXX Please check. - *[from Tomcat by TM]*

The FNs do not contain task tags	// stupid Swing - *[from JEdit by NLP]*
	// not yet handled - *[from JRuby by NLP]*
	// not thread safe - *[from Groovy by NLP]*
	// Dummy implementation - *[from Tomcat by NLP]*
	// not absolutely necessary - *[from Hibernate by TM]*
	// there's a risk - *[from Gradle by TM]*
	// I think this is wrong - *[from Hive by TM]*
	// Multiple bad arguments - *[from Zookeeper by TM]*

are better than the counterparts under OTO scenario according to Figure 12 and Figure 13. However, we cannot get the similar conclusion for *NLP* under different scenarios according to Figure 11, which is consistent with the result of *NLP* in RQ2. The above findings mean that using multiple projects rather than a single project as the training set is in favor of improving the robustness of the two supervised approaches. However, the unsupervised approach *MAT* is free of the concept drift challenge, since it does not depend on the training data.

Conclusion. *In summary, concept drift exists when applying all three supervised approaches to identify SATD comments, which can negatively impact their performance. This is one possible reason why the investigated supervised approaches do not exhibit outstanding performance compared with MAT.*

(2) ***Which SATD comments cannot be identified by the supervised approaches?***

According to Table 11 in RQ3, we can find that *MAT* can identify 23.5% and 16.9% more unique SATD comments compared with the number of TP instances classified by *NLP* and *TM*, respectively. In other words, there are many SATD comments with task tags misclassified by these supervised approaches. In the following, we investigate the characteristics of SATD comments (especially for those contain task tags) that cannot be identified by the supervised approaches (i.e., *NLP* and *TM*).

Table 14 lists example SATD comments that cannot be identified by the supervised approaches (i.e., FN). In particular, the FNs are divided into two types based on whether the comments contain task tags or not. As can be seen, the FNs that contain task tags and do not contain task tags both have very obvious semantic meanings that indicate SATD. One possible reason for these misclassifications is that the supervised approaches will combine the weights of each word in a target comment to determine the final category (SATD or non-SATD), which is easily affected by some unrelated words. For example, the SATD comment "TODO: Should this be OK?" in JMeter is misclassified as non-SATD comment by *TM*. In this example, the word "OK" may incorrectly affect the final decision of *TM*. However, for the comments that contain task tags (shown in Table 14), we can see that they can be easily identified by human experts or *MAT*.

Table 15 reports the distribution of false negative instances (i.e., comments) that have task tags. For each supervised approach (*NLP* or *TM*), the first column reports FN-tags, the number of instances having task tags in false negative instances. The second column reports N-tags (i.e., the

Table 15. The Distribution of False Negative Instances that Have Task Tags for *NLP* and *TM*

Approach	NLP			TM		
Project	**FN-tags**	**N-tags (FN-tags/N-tags)**	**FN (FN-tags/FN)**	**FN-tags**	**N-tags (FN-tags/N-tags)**	**FN (FN-tags/FN)**
Ant	9	14 (64.29%)	52 (17.31%)	15	18 (83.33%)	59 (25.42%)
ArgoUML	144	207 (69.57%)	192 (75.00%)	184	241 (76.35%)	239 (76.99%)
Columba	29	34 (85.29%)	43 (67.44%)	7	8 (87.50%)	23 (30.43%)
EMF	5	5 (100.00%)	48 (10.42%)	3	3 (100.00%)	42 (7.14%)
Hibernate	53	62 (85.48%)	141 (37.59%)	31	32 (96.88%)	118 (26.27%)
JEdit	14	19 (73.68%)	133 (10.53%)	9	14 (64.29%)	136 (6.62%)
JFreeChart	18	28 (64.29%)	37 (48.65%)	0	0 (0.00%)	24 (0.00%)
JMeter	30	36 (83.33%)	88 (34.09%)	20	23 (86.96%)	75 (26.67%)
JRuby	65	71 (91.55%)	100 (65.00%)	26	31 (83.87%)	63 (41.27%)
SQuirrel	32	36 (88.89%)	91 (35.16%)	17	20 (85.00%)	84 (20.24%)
Dubbo	11	21 (52.38%)	31 (35.48%)	5	8 (62.50%)	27 (18.52%)
Gradle	66	116 (56.90%)	119 (55.46%)	52	75 (69.33%)	118 (44.07%)
Groovy	33	73 (45.21%)	71 (46.48%)	20	36 (55.56%)	55 (36.36%)
Hive	152	223 (68.16%)	350 (43.43%)	123	167 (73.65%)	343 (35.86%)
Maven	18	32 (56.25%)	54 (33.33%)	11	12 (91.67%)	51 (21.57%)
Poi	146	200 (73.00%)	214 (68.22%)	46	61 (75.41%)	116 (39.66%)
SF	13	35 (37.14%)	38 (34.21%)	6	22 (27.27%)	32 (18.75%)
Storm	11	16 (68.75%)	39 (28.21%)	7	7 (100.00%)	38 (18.42%)
Tomcat	40	80 (50.00%)	92 (43.48%)	53	73 (72.60%)	110 (48.18%)
Zookeeper	7	22 (31.82%)	32 (21.88%)	7	16 (43.75%)	33 (21.21%)
Average	44.8	66.5 (67.37%)	98.25 (45.60%)	32.1	43.35 (74.05%)	89.3 (35.95%)
Median	29.5	35.5 (83.10%)	79.5 (37.11%)	16	21 (76.19%)	61 (26.23%)

number of instances having task tags in the predicted negative instances) as well as the ratio of FN-tags to N-tags (shown in parentheses). The ratio of FN-tags to N-tags indicates what percentage of instances with task tags in the predicted negative instances is actual SATD comments. The third column reports FN (i.e., the number of false negative instances) as well as the ratio of FN-tags to FN (shown in parentheses). The ratio of FN-tags to FN indicates what percentage of actual SATD instances in the false positive instances can be identified by matching task tags.

From Table 15, we have the following two observations: On the one hand, of the instances with tags in the predicted negative instances, more than two-thirds of instances are actual SATD comments. On average, for *NLP*, 44.8 SATD comments with task tags are misclassified as non-SATD comments, which accounts for 67.4% of the predicated negative instances with task tags. Surprisingly, this proportion is higher (32.1/43.35 = 74.05%) for *TM*. This indicates that most of the comments with tags in the predicted negative instances are actual SATD comments with a small rate of exceptions. On the other hand, of the false negative instances, more than one-third of instances have already been marked by task tags. Specifically, on average, there are 45.60% such instances for *NLP* and 35.95% such instances for *TM*. This indicates that there are non-ignorable false negative instances that can be corrected by matching task tags. The above two observations reveal that: (1) many actual SATD comments marked with task tags cannot be identified by a supervised approach, thus hindering its effectiveness; and (2) matching task tags have a potential in reducing false negative instances, thus helping boost its effectiveness.

Conclusion. *In summary, quite a part of the false negative instances caused by NLP and TM contain the popular task tags, which can be easily identified by MAT. Meanwhile, it seems that the improvement of a supervised approach could be a challenging problem if the researchers do not use MAT to make up for the shortcomings, but only by adjusting their own mechanism.*

Table 16. The Performance Comparisons among *MAT, NLP, TM, NLP+MAT*, and *TM+MAT* Based on 3 Projects with Low Ratio of tags (<=50%) and 17 Projects with High Ratio (>50%) of Tags under the MTO Scenario

Ratio of tags in projects		Low (3)					High (17)				
Approach	Details	Precision	Recall	F_1	ER	RI	Precision	Recall	F_1	ER	RI
MAT	MAT	0.907	0.339	0.485	0.961	26.081	0.798	0.748	0.768	0.920	18.318
NLP	NLP	0.585	0.386	0.459	0.941	16.145	0.670	0.645	0.652	0.910	13.904
	NLP+MAT	0.608	0.462	0.519	0.943	16.893	0.663	0.799	0.719	0.908	13.782
	Impr.	**4.05%**	**19.67%**	**13.00%**	**0.21%**	**4.63%**	-0.97%	**23.86%**	**10.33%**	-0.13%	-0.88%
TM	TM	0.768	0.386	0.508	0.954	21.613	0.755	0.690	0.717	0.917	16.978
	TM+MAT	0.774	0.464	0.571	0.955	21.883	0.750	0.782	0.761	0.917	16.681
	Impr.	**0.78%**	**20.22%**	**12.54%**	**0.10%**	**1.25%**	-0.60%	**13.28%**	**6.18%**	-0.02%	-1.74%

(3) ***Can the effectiveness of supervised approaches be promoted by incorporating MAT?***

According to the result of RQ2, *MAT* can achieve a high precision for each project. This means that the majority of comments that contain task tags are indeed SATD comments, which can be identified by *MAT* accurately. Meanwhile, according to the Figure 9 of RQ3, we can summarize that there are 284 (or 211) SATD comments that correctly classified by *NLP* (or *TM*) cannot be identified by *MAT*, since some SATD comments are not marked by task tags. In other words, the advantages of *NLP*/*TM* and *MAT* are complementary. This motivates us to investigate whether we can achieve more effective SATD identification by combining *MAT* and *NLP* (or *TM*).

To this end, for each target project, we first apply *MAT* to identify the comments that contain task tags and predict them as SATD comments. Then, we apply *NLP* (or *TM*) to classify the remaining comments that do not contain any task tags into two categories: SATD and non-SATD. Finally, we combine the SATD comments predicted by two approaches (*NLP+MAT* or *TM+MAT*). To obtain correct conclusions, we divide the experimental projects into two groups (i.e., Low group and High group) according to the ratio of task tags in the SATD comments of a project and report them, respectively. Low group includes 3 projects (i.e., Ant, EMF, and JEdit) with a task tag ratio £ 50% and High group includes the remaining 17 projects.

Table 16 reports the average performance comparisons among *MAT, NLP, TM, NLP+MAT*, and *TM+MAT* in Low group and High group. As can be seen, the combined approaches (*NLP+MAT* or *TM+MAT*) have a higher performance than the original supervised approach (*NLP* or *TM*) in the Low group. Specifically, *NLP+MAT* leads to an improvement of 4.05% in precision, 19.67% in recall, 13.00% in F_1, 0.21% in ER, and 4.63% in RI compared with *NLP*. *TM+MAT* also achieves improvements in terms of five indicators, especially for recall (20.22%) and F_1 (12.54%) compared with *TM*. As for the High group, on average, two combined approaches both achieve improvements in terms of recall and F_1. The results indicate that *MAT* has a positive effect for recall and F_1 when combined with the supervised approaches (*NLP* or *TM*). At the same time, the values of the other three indicators (i.e., precision, ER, and RI) of the two combined approaches are almost not affected negatively. In other words, the supervised approaches *NLP* and *TM* can be boosted by *MAT*.

Compared with *MAT*, for the project in the Low group, the combined approaches also achieve a higher performance in recall and F_1 on average. More specifically, *NLP+MAT* (or *TM+MAT*) can lead to an improvement of 36.28% (or 36.87%) in recall and 7.01% (or 17.73%) in F_1. In particular, for JEdit, *TM+MAT* achieves a great improvement of 70.24% compared with *MAT* in terms of recall. These improvements indicate that the combined approaches can recall more SATD comments in the projects with few task tags. Nevertheless, for the projects in the High group, we find that the combined approaches do not perform better than *MAT* according to most indicators, since the *NLP*

(or *TM*) will misclassify many non-SATD comments as SATD comments. This means that many false positive instances are introduced. As a result, overall, it appears that the F_1 score does not benefit from the combination compared with *MAT*.

Conclusion. *In summary, the combined approaches (NLP+MAT and TM+MAT) have a better overall performance than a single supervised approach (NLP or TM). In addition, they can facilitate MAT to recall more SATD comments that are not labeled by task tags.*

6 DISCUSSION

In previous sections, we compare *MAT* with existing approaches to gain the understanding of the real progress in SATD identification. The used *MAT* is very simple but enough for our purpose in this study. However, in practice, it may be expected that *MAT* could achieve a higher effectiveness in SATD identification. In this section, we conduct additional experiments to provide a more comprehensive understanding on the characteristics of *MAT*, thus facilitating researchers and practitioners to improve and apply *MAT* in the future.

6.1 How Does Fuzzy Matching Strategy Affect the Classification Effectiveness of *MAT*?

As aforementioned, comment words may be connected together (e.g., "pleasefixme," "hackhere") due to carelessness. These typos will negatively increase the proportion of mismatching when using a strict matching strategy. To tackle this problem, *MAT* takes a fuzzy matching strategy (as described in Section 3) to identify SATD. To determine the usefulness of a fuzzy strategy, we next compare the effectiveness of a strict matching strategy versus a fuzzy matching strategy. To this end, we construct two *MAT* variants that, respectively, apply strict and fuzzy strategies to match task tags in comments for SATD identification. The only difference between the two variants is the selection of matching strategy. For each target project, *MAT* will identify the label (i.e., SATD or non-SATD) of each comment in the project. After that, we have 20 classification results for the models that apply strict strategy and fuzzy strategy, respectively. As such, we compute precision, recall, F_1, ER, and RI to evaluate the effectiveness of each matching strategy.

Table 17 reports the performance comparison between strict and fuzzy matching strategies. The improvement percentages marked by bold fonts indicate that the performance of a fuzzy matching strategy is superior to that of a strict matching strategy. As can be seen, on average, a fuzzy strategy leads to a slightly higher recall and almost the same precision compared with the strict strategy. In other words, for most projects, a fuzzy matching strategy can identify more SATD comments and at the same time it does not introduce too many false positives. Note that, the recalls of more than half of the projects (13/20) have increased, indicating that there are typos in the comments of these projects and the occurrence of typos is not an accident. In particular, the recall of a fuzzy matching strategy has a great improvement of 18.5% on the project Gradle. Meanwhile, although few projects (except Zookeeper) introduce some additional false positives when applying the fuzzy strategy, the percentages of these false positives are very low and hence it should not waste much effort of developers to review these additional comments. Considering the fact that a fuzzy matching strategy can recall more SATD comments, we believe that it is worthwhile to use it to identify these SATD comments. In terms of F_1, a fuzzy matching strategy leads to a higher score (0.726) than strict matching (0.721). More specifically, fuzzy strategy slightly improves F_1 for 10 out of the 20 projects and leads to an inferior F_1 in only 3 projects. On average, fuzzy strategy leads to an improvement of 0.62% in terms of F_1. Considering the effort-aware indicators, there are no significant differences between two strategies for most projects. Note that, the performances between strict strategy and fuzzy strategy are the same on 6 projects, regardless of which indicators are considered. The result shows that the comments in these projects are well written by developers so there are no typos in these comments.

Table 17. Performance Comparisons between Strict and Fuzzy Matching Strategies

Indicator	Precision			Recall			F_1			ER			RI		
Approach	Strict	Fuzzy	Impr.%	Strict	Fuzzy	Impr.%	Strict	Fuzzy	Impr.%	Strict	Fuzzy	Impr.%	Strict	Fuzzy	Impr.%
Ant	0.865	0.870	**0.6%**	0.441	0.461	**4.5%**	0.584	0.603	**3.3%**	0.961	0.962	**0.1%**	24.894	25.043	**0.6%**
ArgoUML	0.838	0.823	-1.8%	0.934	0.934	0.0%	0.883	0.874	-1.0%	0.787	0.783	-0.5%	3.692	3.606	-2.3%
Columba	0.912	0.906	-0.7%	0.813	0.828	**1.8%**	0.860	0.865	**0.6%**	0.966	0.965	-0.1%	28.15	27.949	-0.7%
EMF	1.000	1.000	0.0%	0.338	0.351	**3.8%**	0.505	0.520	**3.0%**	0.971	0.971	0.0%	33.932	33.932	0.0%
Hibernate	0.944	0.945	**0.1%**	0.714	0.724	**1.4%**	0.813	0.820	**0.9%**	0.840	0.840	0.0%	5.239	5.244	**0.1%**
JEdit	0.844	0.851	**0.8%**	0.195	0.205	**5.1%**	0.317	0.331	**4.4%**	0.950	0.951	**0.1%**	19.111	19.268	**0.8%**
JFreeChart	0.723	0.723	0.0%	0.723	0.723	0.0%	0.723	0.723	0.0%	0.944	0.944	0.0%	16.847	16.847	0.0%
JMeter	0.924	0.924	0.0%	0.780	0.780	0.0%	0.846	0.846	0.0%	0.926	0.926	0.0%	12.597	12.597	0.0%
JRuby	0.911	0.911	0.0%	0.877	0.883	**0.7%**	0.894	0.897	**0.3%**	0.885	0.885	0.0%	7.683	7.687	**0.1%**
SQuirrel	0.925	0.925	0.0%	0.612	0.612	0.0%	0.737	0.737	0.0%	0.951	0.951	0.0%	19.581	19.581	0.0%
Dubbo	0.750	0.750	0.0%	0.741	0.741	0.0%	0.746	0.746	0.0%	0.931	0.931	0.0%	13.55	13.550	0.0%
Gradle	0.667	0.671	**0.6%**	0.623	0.738	**18.5%**	0.644	0.703	**9.2%**	0.855	0.856	**0.1%**	5.903	5.952	**0.8%**
Groovy	0.747	0.727	-2.7%	0.819	0.823	**0.5%**	0.782	0.772	-1.3%	0.925	0.923	-0.2%	12.31	11.948	-2.9%
Hive	0.785	0.783	-0.3%	0.748	0.761	**1.7%**	0.766	0.772	**0.8%**	0.955	0.954	-0.1%	21.023	20.976	-0.2%
Maven	0.746	0.746	0.0%	0.691	0.691	0.0%	0.718	0.718	0.0%	0.850	0.851	0.0%	5.687	5.687	0.0%
Poi	0.844	0.845	**0.1%**	0.848	0.854	**0.7%**	0.846	0.850	**0.5%**	0.951	0.951	0.0%	19.526	19.550	**0.1%**
SF	0.650	0.654	**0.6%**	0.684	0.694	**1.5%**	0.667	0.673	**0.9%**	0.980	0.981	**0.1%**	50.189	50.454	**0.5%**
Storm	0.848	0.848	0.0%	0.609	0.609	0.0%	0.709	0.709	0.0%	0.970	0.970	0.0%	32.561	32.561	0.0%
Tomcat	0.742	0.741	-0.1%	0.763	0.767	**0.5%**	0.753	0.753	0.0%	0.968	0.968	0.0%	30.604	30.534	-0.2%
Zookeeper	0.750	0.648	-13.6%	0.524	0.556	**6.1%**	0.617	0.598	-3.1%	0.969	0.964	-0.5%	31.036	26.685	-14.0%
Average	0.821	0.815	-0.8%	0.674	**0.687**	**1.9%**	0.721	**0.726**	**0.7%**	**0.927**	0.926	-0.1%	19.706	19.483	-1.1%
Median	0.841	0.834	-0.8%	0.719	**0.731**	**1.7%**	0.742	0.742	0.0%	0.951	0.951	0.0%	19.319	**19.409**	**0.5%**

In summary, by correctly dealing with the typos of task tags, a fuzzy matching strategy can improve the recall and maintain almost a similar performance in the other indicators compared with a strict matching strategy. This can facilitate developers to find more SATD comments by *MAT* in practice.

6.2 How Do Project-specific tags Affect the Classification Effectiveness of *MAT*?

In default, *MAT* matches four popular task tags to identify SATD comments. Although it can achieve a promising performance in terms of average F_1, we find that some projects have a low recall (e.g., 0.205 for JEdit and 0.351 for EMF) due to only a few comments of the corresponding projects containing the four task tags. In this context, an interesting problem is naturally raised: Is there a simple approach to extending *MAT* such that the recall in such projects can be improved?

To tackle the above problem, we manually read the comments in these projects that have a low recall to understand their characteristics. Consequently, we make the following observations: First, the developers of these projects did not always use the default task tags (e.g., "TODO") in their comments. For instance, there are only 40 comments that contain default tags in all SATD comments (195) in JEdit. Second, in addition to default task tags, there are project-specific task tags (defined by developers, usually highlighted in capitalized words). For example, the word "NOTUSED" often appears in the comments in JMeter. Table 18 summarizes the meanings and examples of project-specific tags used in the subject projects under study. Intuitively, for a given project, if we adapt *MAT* to incorporate such project-specific task tags, the performance in SATD identification would be improved. For the simplicity of presentation, we name *MAT* incorporating project-specific tags as *MAT-ext*.

Table 19 reports the performance comparison between *MAT* and *MAT-ext* on projects with project-specific task tags under the MTO scenario. As can be seen, on average, *MAT-ext* achieves an improvement of 13.37% and 8.77% compared with *MAT* in terms of recall and F_1, respectively. This indicates that, the incorporation of project-specific task tags in *MAT* has a positive effect in

Table 18. The Specified Tags Used in the Subject Projects

Specified Task Tags	Projects	The meanings of tags	Examples
WORKAROUND	Columba Hibernate JEdit SQuirrel Poi	Alternatives, the problem itself has not been solved	//**WORKAROUND**: we simply append URLs to the existing global class loader and use the same as parent - *[from Columba]* **Workaround** for backwards compatibility. Previously this case would unintentionally cause the method to be invoked on the owner continue below - *[from Gradle]*
TBD	EMF Hibernate	The abbreviation of "To Be Determined"	// **TBD** filter out volatile and other inappropriate links? - *[from EMF]*
REVISIT	EMF Tomcat	There is something that need to be revisit	// **REVISIT**: Remove this code. // Store port value as string instead of integer. - *[from EMF]*
Note	JEdit	There is something that need to be noted	//"/* **Note**: This class is messy. The method and field resolution need to be rewritten. - *[from JEdit]*
NOTUSED	JMeter	The statements that not used	// **NOTUSED** private String chosenFile; - *[from JMeter]*
REMIND	JMeter Tomcat	A weak code that should pay attention to later	// **REMIND**: convert arg list Vectors here? - *[from JMeter]*
UNDONE	Hive	Something has not be done	//**UNDONE**: Haven't finished isRepeated -*[from Hive]*
DEPRECATED	Hive	Current code is deprecated	//**@deprecated** in favour of {@link HCatTable.#collectionItemsTerminatedBy()}. To be removed in Hive 0.16. -*[from Hive]*

Table 19. The Performance Comparison between *MAT* and *MAT-ext* on Projects with Project-specific Task Tags under the MTO Scenario

Indicator	Precision			Recall			F_1			ER			RI		
Approach	MAT	MAT-ext	Impr.%	MAT	MAT-ext	Impr.%	MAT	MAT-ext	Impr.%	MAT	MAT-ext	Impr.%	MAT	MAT-ext	Impr.%
Columba	0.906	0.910	**0.44%**	0.828	0.867	**4.71%**	0.865	0.888	**2.66%**	0.965	0.966	**0.10%**	27.949	28.072	**0.44%**
EMF	1.000	0.898	-10.20%	0.351	0.595	**69.52%**	0.520	0.715	**37.50%**	0.971	0.968	-0.31%	33.932	30.369	-10.50%
Hibernate	0.945	0.943	-0.21%	0.724	0.743	**2.62%**	0.820	0.831	**1.34%**	0.840	0.839	-0.12%	5.244	5.232	-0.23%
JEdit	0.851	0.683	-19.74%	0.205	0.441	**115.12%**	0.331	0.536	**61.93%**	0.951	0.938	-1.37%	19.268	15.255	-20.83%
JMeter	0.924	0.911	-1.41%	0.780	0.798	**2.31%**	0.846	0.851	**0.59%**	0.926	0.925	-0.11%	12.597	12.399	-1.57%
SQuirrel	0.925	0.929	**0.43%**	0.612	0.652	**6.54%**	0.737	0.766	**3.93%**	0.951	0.952	**0.11%**	19.581	19.675	**0.48%**
Hive	0.783	0.759	-3.07%	0.761	0.820	**7.75%**	0.772	0.789	**2.20%**	0.954	0.953	-0.10%	20.976	20.298	-3.23%
Tomcat	0.741	0.743	**0.27%**	0.767	0.784	**2.22%**	0.753	0.763	**1.33%**	0.968	0.969	**0.10%**	30.534	30.612	**0.26%**
Average	0.884	0.847	-4.23%	0.629	**0.713**	**13.37%**	0.706	**0.767**	**8.77%**	0.941	0.939	-0.21%	21.260	20.239	-4.80%
Median	0.915	0.904	-1.20%	0.743	**0.764**	**2.83%**	0.763	**0.778**	**1.97%**	0.953	0.953	0.00%	20.279	19.987	-1.44%

recalling more SATD comments. In particular, in JEdit, the improvement in recall is 61.93%, since a large number of the tag "Note" exists in the comments. Considering the other indicators, the differences between *MAT* and *MAT-ext* are not large. Therefore, in practice, it is recommended that developers incorporate project-specific tags to *MAT*, which can lead to a more accurate SATD identification.

Table 20 reports the performance comparison between *CNN* and *MAT-ext* under the MTO scenario in Dataset-M. Note that, the gray rows denote the performance comparison on projects with project-specific task tags. We can see that, after incorporating project-specific task tags, *MAT-ext* achieves a more competitive performance compared with *CNN*. On average, *MAT-ext* performs well in precision while *CNN* has an advantage in recall. The reason for this is that *MAT-ext* can only identify the comments marked by tags, while *CNN* can find SATD comments without tags. However, the precision of *CNN* is relatively low, since more irrelevant comments are misclassified

Table 20. The Performance Comparison between *CNN* and *MAT-ext* on Dataset-M under the MTO Scenario

Indicator	Precision			Recall			F_1			ER			RI		
Approach	CNN	MAT-ext	Impr.%	CNN	MAT-ext	Impr.%	CNN	MAT-ext	Impr.%	CNN	MAT-ext	Impr.%	CNN	MAT-ext	Impr.%
Ant	0.584	0.870	**48.97%**	0.758	0.461	-39.18%	0.660	0.603	-8.64%	0.943	0.962	**2.01%**	16.474	25.043	**52.02%**
ArgoUML	0.816	0.823	**0.86%**	0.950	0.934	-1.68%	0.878	0.874	-0.46%	0.781	0.783	**0.26%**	3.569	3.606	**1.04%**
Columba	0.830	0.910	**9.64%**	0.875	0.867	-0.91%	0.852	0.888	**4.23%**	0.962	0.966	**0.42%**	25.521	28.072	**10.00%**
EMF	0.793	0.898	**13.24%**	0.594	0.595	**0.17%**	0.679	0.715	**5.30%**	0.964	0.968	**0.41%**	26.701	30.369	**13.74%**
Hibernate	0.930	0.943	**1.40%**	0.743	0.743	0.00%	0.826	0.831	**0.61%**	0.837	0.839	**0.24%**	5.147	5.232	**1.65%**
JEdit	0.773	0.683	-11.64%	0.489	0.441	-9.82%	0.599	0.536	-10.52%	0.946	0.938	-0.85%	17.409	15.255	-12.37%
JFreeChart	0.686	0.723	**5.39%**	0.802	0.723	-9.85%	0.739	0.723	-2.17%	0.941	0.944	**0.32%**	15.939	16.847	**5.70%**
JMeter	0.873	0.911	**4.35%**	0.787	0.798	**1.40%**	0.828	0.851	**2.78%**	0.922	0.925	**0.33%**	11.841	12.399	**4.71%**
JRuby	0.805	0.911	**13.17%**	0.930	0.883	-5.05%	0.836	0.897	**7.30%**	0.870	0.885	**1.72%**	6.676	7.687	**15.14%**
SQuirrel	0.794	0.929	**17.00%**	0.692	0.652	-5.78%	0.739	0.766	**3.65%**	0.943	0.952	**0.95%**	16.669	19.675	**18.03%**
Average	0.788	**0.860**	**9.09%**	**0.762**	0.710	-6.86%	0.764	**0.768**	**0.63%**	0.911	**0.916**	**0.58%**	14.595	**16.419**	**12.50%**
Median	0.800	**0.904**	**13.07%**	**0.773**	0.733	-5.11%	0.783	**0.799**	**2.04%**	0.942	0.941	-0.11%	16.207	16.051	-0.96%

as SATD. In terms of F_1, *MAT-ext* is better than *CNN* on most projects (i.e., better on 6 projects, worse on 4 projects). In particular, we can find that the average F_1 values of *CNN* and *MAT-ext* are very close (0.764 for *CNN* vs. 0.768 for *MAT-ext*). If we take into account the cost of modeling building and application, it is clear that *MAT-ext* is preferred in practice.

For researchers, they do not know what tags are in a specific project in advance. However, for the developers responsible for the project, it is easy to acquire project-specific task tags, because developers in the project usually specify the used tags clearly at the stage of design to better maintain the project later. Therefore, it is meaningful to apply *MAT-ext* to improve the practicability of *MAT* in real projects.

6.3 What Are the Causes of Misclassification by *MAT*?

Although an excellent prediction performance of *MAT* has been shown in Section 5, there are some inherent limitations when applying *MAT* to identify SATD comments due to the simple design of *MAT*, i.e., *MAT* can output some misclassified comments. According to our thorough analysis, we conclude that the misclassification of *MAT* can be classified into four kinds of situations, shown in Table 21.

We report the summary of misclassification of *MAT* in Table 21 to illustrate the main causes:

(1) **False positive instances caused by auto-generated tag comments.** For the IDEs (e.g., Eclipse) that integrate the function of tagging comments, developers may introduce an auto-generated comment, which consists of only a task tag, due to their negligence. For example, there are some comments like "/* XXX*/" in JRuby (in Table 21) generated by IDE. However, it is actually unclear whether a developer tends to use such a comment to indicate SATD or they are just auto-generated by IDEs. In the dataset provided by Maldonado et al. [17], this kind of comment is manually classified as non-SATD. In this situation, *MAT* will introduce false positive instances. To reduce such FPs, one can filter a comment if it consists of only one tag (e.g., "//TODO") token after preprocessing the original comment.

(2) **False positive instances caused by the components of a sentence.** According to our observation, there is another kind of false positive instance: some non-SATD comments contain the same word (e.g., "hack") as a task tag but it is not tended to be used as a tag. For the example of "/* Owner related todo items: */" from the project ArgoUML, the word "todo" is not a task tag but a component word of the comment sentence. Note that

Table 21. The Summary of Misclassification of *MAT*

Type	Reasons of mis-classification	Examples	Possible solution
False Positive	FPs caused by the auto-generated tag comments.	// **TODO**: - *[from ArgoUML, Storm, Poi, Maven]* //**TODO**!!! - *[from Hibernate]* // **FIXME** - *[from JFreeChart, Tomcat, Groovy]* /* **XXX***/ - *[from JRuby, Tomcat]*	Filtering the comments that only contain one task tag token.

	FPs caused by the components of sentences.	/* Owner related **todo** items: */ - *[from ArgoUML]*
		// Copy the **todo** items after the model - *[from ArgoUML]*
		// no item exists in table // → nothing **todo** - *[from Columba]*
		// **Hack** to ensure charset is set correctly at start-up - *[from Columba]*
			Filtering the comments whose task tag does not appear at the top of the sentence.

False Negative	FNs caused by the untagged comments.	// Check it out; also ugly. - *[from Ant]*
		// this is ugly - *[from Zookeeper]*
		// Not implemented - *[from SQuirrel, SpringFramework]*
		// not thread safe - *[from Hive]*
		// there is a risk - *[from Gradle]*
			Combining supervised approaches and MAT.

	FNs caused by the forms of tags.	// **TO DO**: these annotations only work with XYPlot - *[from JFreeChart]* //**TOOD**: get a real example file ... to actual test the FBSE entry not sure where the foDelay block is -*[from Poi]*	Setting a matching dictionary of one-of-a-kind tags.

these comments are not marked by task tags. The reason of the misclassification of *MAT* is that task tags (e.g., "TODO") will be transformed into its original form (e.g., "todo") in the preprocessing stage before matching. This will be confused with the components of sentences and hence may cause false positive instances. According to our observation, a task tag usually is the first token of one comment. Therefore, this kind of FPs can be reduced by filtering out the comments in which the first token is not a task tag.

(3) **False negative instances caused by the untagged comments.** This is the main kind of misclassification of *MAT*. The rationale of *MAT* assumes that developers will use tags to mark the technical debts admitted by themselves. However, this is not the habit of some developers and hence *MAT* will perform poorly in the projects for which these developers are responsible. For these comments, we can use the state-of-the-art supervised approaches (e.g., *TM* [10] or *CNN* [49]) to identify the SATD comments by extracting useful semantic information from comments.

(4) **False negative instances caused by the formats of tags.** This kind of misclassification is uncommon but also shows the weakness of *MAT*. In the application of *MAT*, we have considered using fuzzy matching strategy to avoid some misclassifications. However, fuzzy matching we used cannot handle all forms of tags. For example, for the SATD comment "//TO DO: delete the file if it is not a valid file" in Ant, *MAT* will classify it as non-SATD, because there is no task tag after preprocessing ("TO DO " will be transformed as two words "to" and "do" but not "todo"). In our studied projects, the situation is extremely rare and there is no universal rule to capture this kind of FNs. One possible solution to this problem is to add these one-of-a-kind tags into the matching dictionary of *MAT* once it is found by developers.

It should be pointed out that the above misclassifications do not have a large influence on the overall effectiveness of *MAT*. The reason is that the percentage of these misclassifications is low

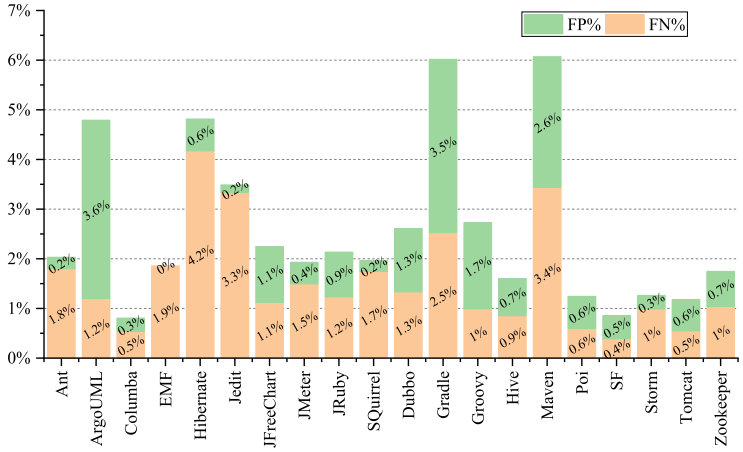


Fig. 14. The statistics of misclassifications by *MAT* in terms of FP% and FN%.

when considering the large number of comments. Figure 14 reports the percentages of misclassification (i.e., FP % and FN %) by *MAT*. We can see that the proportion of the misclassification cases of the majority projects is very low (less than 2%).

In summary, there are only a small part of comments that are wrongly classified by *MAT*. Moreover, most types of FPs and FNs of *MAT* can be corrected easily by applying more refined matching strategies.

6.4 How Well Does *Jitterbug* Perform Compared with *MAT*?

In the experiments of Section 5, all approaches we compared are automatic technologies that do not need manual labeling by human experts' effort (i.e., human intervention). As a result, only the "*Easy*" component in Yu et al.'s *Jitterbug* is compared with *MAT*. In Yu et al.'s study [95], the "Hard" component was designed to identify SATD from the comments not matched by the "*Easy*" component. The purpose of the "Hard" component is to increase the recall of the classification results of *Easy* via human experts' manual labelling. To this end, the "Hard" component uses a recommendation model to iteratively recommend top 10 comments with the highest probabilities of being SATD to human experts for labeling and uses the labeled data to update the recommending model. This process is repeated until the estimated recall reaches a target recall. In this section, we analyze how well *MAT* can perform compared with *Jitterbug*. Following the settings in their experiment, we run Yu et al.'s script to obtain the classification result of *Jitterbug* when its estimated recall reaches 0.9.

Figure 15 reports the performance comparison among *Easy, Jitterbug*, and *MAT*. It can be seen that there is almost no difference between *MAT* and *Easy* in terms of all performance indicators. However, a large difference can be found between *MAT* and *Jitterbug*. Because the purpose of *Jitterbug* is to identify more SATD comments (not just the comments that contain keywords), the recalls of *Jitterbug* (with an average of 0.958) are significantly higher than that of *MAT* (with an average of 0.687). The results show that *Jitterbug* has indeed achieved its goal. However, a large decrease occurs for the precision of *Jitterbug*, which only achieves 0.153 on average. The reason is that many non-SATD comments are recommended to human experts for labelling. These recommended candidate comments are considered by *Jitterbug* to be the most probable ones that indicate SATD. However, most of them are false alarms, leading to a low precision. As a result, the F_1 scores are pulled from 0.709 (*Easy*) down to 0.251 (*Jitterbug*). In addition, the effort-aware indicators of

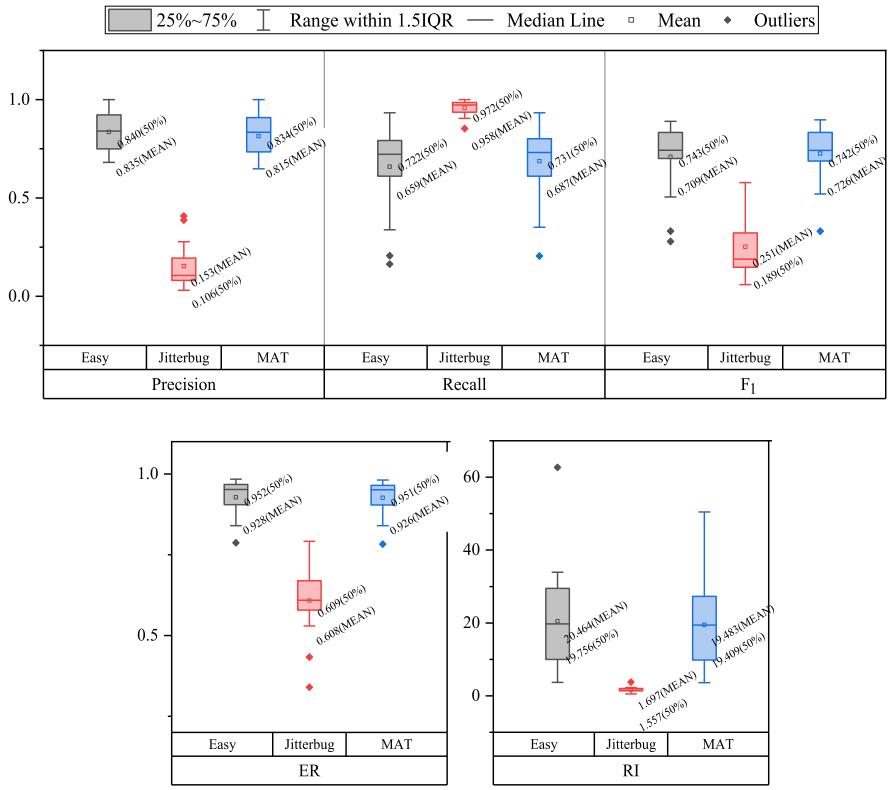


Fig. 15. The performance comparison among *Easy*, *Jitterbug*, and *MAT* based on Dataset-M and Dataset-G under the MTO scenario.

Jitterbug are considerably lower than *MAT* or *Easy*. In particular, *Jitterbug* only achieves 1.697 in terms of RI on average. This indicates that *Jitterbug* can only recall about 69.7% more SATD comments than a random approach.

According to the above results, *Jitterbug* achieves a quite high recall by leveraging human expert's effort while it introduces many false positives to the final classification result, which may not be economical in practice. Although *MAT* does not achieve such a high recall, its overall classification performance is competitive, and it does not need any human expert's effort. Therefore, it should be more useful in practice.

7 IMPLICATIONS FOR TOOL PROVIDERS, PRACTITIONERS, AND RESEARCHERS

This study has important implications for tool providers, practitioners, and researchers.

7.1 For Tool Providers

This work gives a simple and intuitive solution for SATD identification, which can be easily added into the popular tools by tool providers to contribute more assistance for users. The detailed implications are as follows:

(1) **This work can motivate tool providers to enhance the role of task tags in existing integrated development environments (IDEs).** In our investigation, although the tool providers of many popular IDEs (e.g., Eclipse, Visual Studio, and NetBeans) have supported

task tags to facilitate the evolution activities of a target project developed in the IDE, this work highlights another important usage scenario of task tag, i.e., SATD identification and management. This finding may motivate tool providers to improve an IDE so it can be used for handling SATD-related task more efficiently. For example, the IDE can maintain a checklist to manage the unremoved SATD comments marked by task tags. To ensure the correctness of code, the checklist can automatically remind developers to review and pay for these unremoved SATDs before a new version of code is released.

(2) **The findings of this work can be integrated into static analysis tools (SATs).** The static analysis tools (e.g., FindBugs, PMD, and Checkstyle) are very useful in the analysis of software defects, which have been widely used in both industry and academia. Therefore, supporting the proposed *MAT* or similar matching solutions in these tools will provide a lot of convenience. For industry, developers can use these lightweight auxiliary tools to detect the potential SATD related static bugs easily. Compared with the supervised approaches, *MAT* is easier to be integrated into SATs. For academia, researchers can use such static analysis tools to support their research in many fields (e.g., defect prediction). For example, incorporating SATD detected by these SATs as a component to boost their proposed models.

7.2 For Practitioners

In practice, *MAT* can be easily used by practitioners for SATD identification when there are many task tags in comments of a target project. The detailed implications are as follows:

(1) **This work reminds practitioners to recognize the value of a simple matching approach in SATD identification again.** In the literature, many recently proposed approaches perform better than *Pattern* [14] in terms of classification accuracy. However, they are often complex, so there is a challenge for practitioners to use them. The proposed *MAT* has great advantages both in classification performance and actual application. Therefore, putting attention to *MAT* will benefit the recognition of SATD in practice. Because *MAT* is a training-free approach, practitioners can easily use it without collecting a lot of training data. This will save practitioners' time and effort greatly. Additionally, in many popular IDEs, these tools have built-in task annotation tags module in their environments, so it is convenient for practitioners to add task tags or scan the existing unsolved task tags in their daily routine.

(2) **We give practitioners useful guides for further improving and applying *MAT* in practice.** In the actual activities, practitioners can use *MAT* very flexibly instead of using it exactly according to the default settings introduced in this work. In other words, *MAT* provides a feasible way instead of a fixed approach for identifying SATD comments. More specifically, as stated in section 6, practitioners may use a more specific fuzzy matching strategy when the characteristics of comments in a target project are unique, add or use important project-specific task tags in *MAT* when these important tags are defined in the corresponding project, or combine existing supervised approaches with *MAT* for identifying SATD comments more accurately when seldom task tags are marked in the target project. Meanwhile, we analyze the limitations of *MAT* so practitioners can be aware of these adverse situations and take actions accordingly to apply *MAT* more reasonably.

(3) **The more meaningful enlightenment to practitioners of our work is that good coding habits and standard specifications will achieve twofold results with half the effort.** Task tags are an important element in comments that is designed as reminders

for a work or an action that needs to be done in the process of software development. For this reason, many popular IDEs (e.g., Eclipse) have integrated the task tags plugin in their environments to assist practitioners to develop and maintain software more easily. Meanwhile, according to our experimental results in Section 5, task tags show an outstanding performance than compared approaches in indicating SATD comments. Therefore, we strongly recommend practitioners to pay more attention to task tags, including the meaning and usage scenario of each task tag. When one developer needs to add a short-term solution (i.e., SATD) in a project due to some inevitable objective reasons, he (or she) had better use a proper task tag provided by the development platform (or a user-defined tag) to mark the SATD in the comment explicitly. If possible, making standard specifications for managing and applying these task tags can not only improve the robustness of software artifacts but also facilitate the identification of SATD comments later.

7.3 For Researchers

We contribute a simple yet strong baseline in SATD identification. In the future, if a new SATD identification approach is proposed, it should be compared against *MAT* to demonstrate its practical value. This will help our community develop more effective approaches for SATD identification. The detailed implications are as follows:

(1) **Our work highlights the importance of task tags in SATD identification.** If there are many task tags in the comments of a target project, then researchers should not build a prediction model by completely ignoring these tags or treating them as ordinary words. Otherwise, the performance of such prediction models (e.g., *TM, NLP*, and *CNN*) may not be better than *MAT*. Indeed, these task tags are predefined by developers or managers in the purpose of indicating problems (e.g., SATD) in software artifacts [22]. Although these task tags may be misused by developers in the process of practical application, the comments with task tags have a high probability of indicating SATD. Furthermore, these task tags are prior knowledge, which can be obtained without learning. Therefore, we strongly recommend that future studies take the task tags into consideration when building new SATD identification approaches.

(2) **Our work gives useful suggestions for improving SATD identification.** At the model constructing phase, we recommend combining the prior knowledge of task tags in his (or her) (supervised or unsupervised) identification approach. At the same time, the newly proposed approach had better automatically identify SATD comments and avoid labor cost as soon as possible. Otherwise, according to Section 6.4, it may not be practical, since the precision of a new approach is very low when increasing the recall rate. At the model evaluation phase, we recommend researchers add a group of test sets that the comments containing tasks from this test set are removed. Such test sets can be regarded as a control group to validate the real effectiveness of the proposed approach without the confounding effects caused by task tags.

(3) **Our work demonstrates an important software engineering scenario that a simple solution could work well compared with complex solutions.** Indeed, in software engineering, it is not uncommon to observe similar phenomena [53, 54, 57, 64, 75-77]. For example, in cross-project defect prediction, we found that, ManualDown, a very simple module-size model, performed similarly to or even better than almost all the state-of-the-art (complex) supervised models [64]; in the Stack Overflow text mining task, Majumder et al. reported that a simple tuned local model performed similarly to the state-of-the-art *CNN* model but was 500+ times faster [77]; in the automatic generation of commit messages, Liu et al. showed that a simple nearest neighbor generator outperformed the

complex neural machine translation algorithm but was 2,600 times faster [75]. The above facts caution that when faced with a software engineering problem, researchers should first seek simple rather than intricate and complex solutions. This is in accordance with the idea of "less, but better" advocated by Menzies,[10] which aims to use simple approaches to solve complex problems. At least, "before researchers release research results, they compare their supposedly sophisticated methods against simpler alternatives" [77]. This will help avoid wasting research effort and find practical solutions.

8 THREATS TO VALIDITY

We consider the most important threats to construct, internal, and external validity of our study.

8.1 Construct Validity

Construct validity is the degree to which the dependent and independent variables accurately measure the concept they purport to measure. In this study, the used dependent variable is a binary variable that indicates whether a code comment is a SATD comment. We used the dataset shared online by Huang et al., which was collected by Maldonado et al. [17] (Dataset-M) and the dataset we collected (Dataset-G). In the data collection, both the authors of the two datasets manually read comments to obtain the dependent variable. During this process, there were ambiguous comments that could not be accurately labeled. For example, a comment only consisting of a task tag (e.g., "// TODO:") cannot be determined whether it indicates a SATD. Therefore, this is a threat to the construct validity of the dependent variable that needs to be reduced in the future work. Furthermore, there is another risk for the Dataset-G that was collected by ourselves, because we have learned the prior knowledge (i.e., task tags may indicate SATD) to label each comment. To mitigate this risk, our group has spent a long period of time (about 200 hours) to label these comments and asked others to verify the labels. Currently, there are no more published datasets except for the ones used in this work. Although *MAT* performs well in 20 of the projects and *MAT* should also work on other datasets intuitively, it is necessary to conduct more experiments to validate the effectiveness of *MAT* on new datasets published by other research groups in the future.

8.2 Internal Validity

Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. There are two main internal threats in this article. The first internal threat is from the selection of existing SATD identification approaches. Because this articleis devoted to the current research progress of SATD identification, it is important to select the most representative works. To this end, the approaches used in this articleare all from the top international conferences and journals. To the best of our knowledge, these approaches are the most widely cited works in the field of SATD identification, and they have attracted much attention. Therefore, in our opinion, this threat has been minimized.

The second internal threat is from the comparison between *MAT* and *CNN*. Because the implementation of *CNN* is not open source, we can only compare the performance of *MAT* and *CNN* (the results of each project are all from their paper [49]) in the Dataset-M in RQ2. The result shows that *MAT* is a competitive identification approach compared with *CNN*. Unfortunately, we cannot compare the classification difference between the two approaches in RQ3. According to Table 15 in their paper [49], we summarize the top 10 important patterns (i.e., tokens) of each project identified by *CNN* and rank them in descending order according to the number of occurrences in

[10] http://menzies.us/.

these projects. We find an interesting phenomenon that the number of times that a pattern appears is high if it can be considered as a task tag. More specifically, "todo" ranked first (occurring 10 times), "hack" ranked second (occurring 9 times), "fixme" ranked third (occurring 6 times), and "xxx" ranked fifth (occurring 4 times). This indicates that *CNN* regards these words (i.e., task tags especially for "todo," "hack," and "fixme") as the most important features in SATD comments. This partly explains why *MAT* and *CNN* have a competitive performance on Dataset-M. However, it is clear that these task tags are prior knowledge, that one does not need to use *CNN* to acquire this information through a complex learning process and then use them to build a SATD identification model. This means that *CNN* may have complicated the SATD identification problem. In addition, because the four task tags are listed as the important features by *CNN*, it is reasonable to expect that the true positive instances (SATD) and true negative instances (non-SATD) recognized by *CNN* may overlap with those identified by *MAT* greatly. Since this is our conjecture, it needs to be validated in future work.

8.3 External Validity

External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. The most important threat is that our findings may not be generalized to non-Java projects or commercial projects. In our experiment, all subject projects are open-source Java projects. In particular, *MAT* uses four tags (i.e., "TODO," "FIXME," "HACK," and "XXX") to identify SATD. For non-Java or commercial projects, if the tags for marking SATD are different from these four tags, then *MAT* will not work. One possible solution is to adapt *MAT* by replacing the four tags with the popular tags or user-defined tags in those projects. In this context, we believe that the adapted *MAT* should still work. Nonetheless, to mitigate this threat, there is a need to reproduce our study across a wider variety of projects in the future.

9 CONCLUSION AND FUTURE WORK

In this article, we conduct a comprehensive empirical study to investigate the real progress in the field of SATD identification. We compare existing approaches with a simple heuristic approach *MAT* in terms of classification performance and classification differences. In nature, *MAT* is an unsupervised approach, which does not need any data to train a prediction model and only matches task tags in comments to identify SATD. The used task tags are predefined for indicating SATD comments and have been supported by many popular IDEs. It is prior that comments with these tags have a high probability of being SATD even if tag misusing is considered. However, existing SATD identification approaches neglect this fact and learn their relationships with SATD from labeled training data. We use 20 different open-source Java projects to conduct the comparison experiment. Our experimental results surprisingly show that *MAT* is very competitive or even superior to all the investigated approaches, regardless of whether effort-unaware or effort-aware performance indicators are considered. Furthermore, for the investigated approaches, the resulting true positive instances (i.e., SATD) and true negative instances (i.e., non-SATD) are highly overlapped with those identified by *MAT*. This result indicates that the real progress in SATD identification is not being achieved as it might have been envisaged. Due to a low computation cost and a low memory requirement, *MAT* can be efficiently applied in practice. Therefore, we strongly recommend that future SATD identification studies consider *MAT* as an easy-to-implement baseline, when many task tags are used in the comments of a target project. Indeed, in light of the fact that task tags are usually signals of SATD, there is no reason to neglect such a natural baseline. In practice, using *MAT* as a baseline will enable us to determine whether a new identification approach is practically useful.

In the future, we will investigate developers' habit of writing comments to develop a more accurate task tag matching for *MAT*. What is more, we will study how to effectively combine *MAT* with existing supervised approaches to boost the performance of SATD identification.

ACKNOWLEDGMENTS

We are very grateful to Maldonado, Shihab, and Tsantalis for sharing their datasets and collection methodologies online. We are also very grateful to Huang, Shihab, Xia, Lo, and Li for sharing their comment datasets and source code of *TM* online, to Yu, Fahid, Tu and Menzies for sharing their code for *Easy* online, and to Ren for providing the details on the datasets and the basic code used in their *CNN*.

REFERENCES

[1] Python Developer's Guide. 2020. Retrieved from https://www.python.org/dev/peps/pep-0350/.
[2] Tips about Eclipse. 2020. Retrieved from http://www.javaperspective.com/how-to-use-todo-and-fixme-task-tags-in-eclipse.html.++6.
[3] Tasklist of NetBeans. 2020. Retrieved from https://ui.netbeans.org/docs/hi/promoB/tasklist.html.
[4] To-Do List plugin of CodeBlocks. 2020. Retrieved from http://wiki.codeblocks.org/index.php/To-Do_List_plugin.
[5] Task Tags Preferences of Eclipse. 2020. Retrieved from https://www.eclipse.org/pdt/help/html/task_tags.htm.
[6] TODO comments of IntelliJ IDEA. 2020. Retrieved from https://www.jetbrains.com/help/idea/using-todo.html.
[7] Task List of Visual Studio. 2020. Retrieved from https://docs.microsoft.com/zh-cn/visualstudio/ide/using-the-task-list?view=vs-2015.
[8] Code Climate. 2020. Retrieved from https://codeclimate.com/.
[9] Android Studio. 2020. Retrieved from https://developer.android.com/.
[10] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empir. Softw. Eng.* 23, 1 (2018), 418–451.
[11] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Mess.* 4, 2 (1993), 29–30.
[12] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD detector: A text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, 9–12.
[13] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Automating change-level self-admitted technical debt determination. *IEEE Trans. Softw. Eng.* 45, 12 (2018), 1211–1229.
[14] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 91–100.
[15] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. IEEE, 179–188.
[16] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR'16). IEEE*, 315–326.
[17] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans. Softw. Eng.* 43, 11 (2017), 1044–1062.
[18] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD'15)*. IEEE, 9–15.
[19] Nico Zazworka, Antonio Vetrò, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn B. Seaman, and Forrest Shull. 2014. Comparing four approaches for technical debt identification. *Softw. Qual. J.* 22, 3 (2014), 403–426.
[20] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *J. Syst. Softw.* 101 (2015), 193–220.
[21] Jiawei Han and Micheline Kamber. 2006. *Data Mining: Concepts and Techniques, Second Edition (The Morgan Kaufmann Series in Data Management Systems)*. Elsevier.
[22] Margaret-Anne D. Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. 251–260.
[23] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetrò, Forrest Shull, and Carolyn B. Seaman. 2013. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE'13)*. 42–47.

[24] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn B. Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the IEEE 2nd International Workshop on Managing Technical Debt (MTD'11)*. 17–23.

[25] Clemente Izurieta, Antonio Vetrò, Nico Zazworka, Yuanfang Cai, Carolyn B. Seaman, and Forrest Shull. 2012. Organizing the technical debt landscape. In *Proceedings of the IEEE 3rd International Workshop on Managing Technical Debt (MTD'12)*. 23–26.

[26] Nicolli S. R. Alves, Leilane Ferreira Ribeiro, Vivyane Caires, Thiago Souto Mendes, and Rodrigo O. Spínola. 2014. Towards an ontology of terms on technical debt. In *Proceedings of the 6th International Workshop on Managing Technical Debt (MTD'14)*. 1–7.

[27] Beat Fluri, Michael Würsch, and Harald C. Gall. 2007. Do code and comments coevolve? On the relation between source code and comment changes. In *Proceedings of 14th Working Conference on Reverse Engineering (WCRE'07)*. 70–79.

[28] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E. Hassan. 2008. Understanding the rationale for updating a function's comment. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'08)*. 167–176.

[29] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Carolyn B. Seaman, Kevin J. Sullivan, and Nico Zazworka. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10)*. 47–52.

[30] Frank Buschmann. 2011. To pay or not to pay technical debt. *IEEE Softw.* 28, 6 (2011), 29–31.

[31] Erin Lim, Nitin Taksande, and Carolyn B. Seaman. 2012. A balancing act: What software practitioners have to say about technical debt. *IEEE Softw.* 29, 6 (2012). 22–27.

[32] Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, and Davide Falessi. 2013. Technical debt: Towards a crisper definition report on the 4th International Workshop on Managing Technical Debt. *ACM SIGSOFT Softw. Eng. Notes* 38, 5 (2013), 51–54.

[33] Gerard Salton, A. Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[34] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation (ICST'12)*. IEEE, 260–269.

[35] Ninus Khamis, René Witte, and Juergen Rilling. 2010. Automatic quality assessment of source code comments: The JavadocMiner. In *Proceedings of the International Conference on Application of Natural Language to Information Systems (NLDB'10)*. 68–79.

[36] Matthew J. Howard, Samir Gupta, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*. 377–386.

[37] Daniela Steidl, Benjamin Hummel, and Elmar Jürgens. 2013. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13)*. IEEE, 83–92.

[38] Bradley L. Vinz and Letha H. Etzkorn. 2008. Improving program comprehension by combining code understanding with comment understanding. *Knowl.-based Syst.* 21, 8 (2008), 813–825.

[39] Hirohisa Aman, and Hirokazu Okazaki. 2008. Impact of comment statement on code stability in open source development. In *Proceedings of the 8th Joint Conference on Knowledge-based Software Engineering (JCKBSE'08)*. 415–419.

[40] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167.

[41] Xiaobing Sun, Qiang Geng, David Lo, Yucong Duan, Xiangyue Liu, and Bin Li. 2016. Code comment quality analysis and improvement recommendation: An automated approach. *Int. J. Softw. Eng. Knowl. Eng.* 26, 6 (2016), 981–1000.

[42] Paul W. Mcburney and Collin Mcmillan. 2016. An empirical study of the textual similarity between source code and source code summaries. *Empir. Softw. Eng.* 21, 1 (2016), 17–42.

[43] Fabrizio Sebastiani. 2002. Machine learning in automated text categorization. *ACM Comput. Surv.* 34, 1 (2002), 1–47.

[44] Haibo He and Edwardo A. Garcia. 2009. Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284.

[45] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. 2017. The evolution of technical debt in the apache ecosystem. In *Proceedings of the European Conference on Software Architecture (ECSA'17)*. 51–66.

[46] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John C. Grundy, and Xiaohu Yang. 2020. On the replicability and reproducibility of deep learning in software engineering. CoRR abs/2006.14244 (2020).

[47] Yoav Benjamini and Daniel Yekutieli. 2001. The control of false discovery rate in multiple testing under dependency. *Ann. Statist.* 29, 4 (2001), 1165–1188.

[48] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*. 329–331.

[49] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural network based detection of self-admitted technical debt: From performance to explainability. *ACM Trans. Softw. Eng. Methodol.* 28, 3 (2019).

[50] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems (NIPS'13)*. 3111–3119.

[51] Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. 2014. Learning sentiment-specific word embedding for Twitter sentiment classification. In *Proceedings of the 52nd Meeting of the Association for Computational Linguistics (ACL'14)*. 1555–1565.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems (NIPS'12)*. 1106–1114.

[53] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 157–168.

[54] Wei Fu and Tim Menzies. 2017. *Easy* over hard: A case study on deep learning. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE'17)*. 49–60.

[55] Qiao Huang, Xin Xia, and David Lo. 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empir. Softw. Eng.* 24, 5 (2019), 2823–2862.

[56] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs. unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, 159–170.

[57] Bowen Xu, Amirreza Shirani, David Lo, and Mohammad Amin Alipour. 2018. Prediction of relatedness in stack overflow: Deep learning vs. SVM: A reproducibility study. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'18)*. 1–10.

[58] Margaret-Anne D. Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael J. Muller. 2009. How software developers use tagging to support reminding and refinding. *IEEE Trans. Softw. Eng.* 35, 4 (2009), 470–483.

[59] Annie T. T. Ying, James L. Wright, and Steven Abrams. 2005. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *Proceedings of the International Workshop on Mining Software Repositories (MSR'05)*. 1–5.

[60] Christopher D. Manning, and Dan Klein. 2003. Optimization, maxent models, and conditional estimation without magic. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL'03)*. 8–8.

[61] Rahul, Krishna and Tim Menzies. 2017. Bellwethers: A baseline method for transfer learning. *IEEE Trans. Softw. Eng.* 45, 11 (2017), 1081–1105.

[62] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2019. "Sampling" as a baseline optimizer for search-based software engineering. *IEEE Trans. Softw. Eng.* 45, 6 (2019), 597–614.

[63] Peter A. Whigham, Caitlin A. Owen, and Stephen G. MacDonell. 2015. A baseline model for software effort estimation. *ACM Trans. Softw. Eng. Methodol.* 24, 3 (2015).

[64] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How far we have progressed in the journey? An examination of cross-project defect prediction. *ACM Trans. Softw. Eng. Methodol.* 27, 1 (2018).

[65] Roberto Souto Maior de Barros and Silas Garrido Teixeira de Carvalho Santos. 2018. A large-scale comparison of concept drift detectors. *Inf. Sci.* 451-452, (2018), 348–370.

[66] Tsung-Han Chan, Kui Jia, Shenghua Gao, Jiwen Lu, Zinan Zeng, and Yi Ma. 2015. PCANet: A simple deep learning baseline for image classification. *IEEE Trans. Image Proc.* 24, 12 (2015), 5017–5032.

[67] Zhiguang Wang, Weizhong Yan, and Tim Oates. 2017. Time series classification from scratch with deep neural networks: A strong baseline. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'17)*. 1578–1585.

[68] Hongjian Wang, Xianfeng Tang, Yu-Hsuan Kuo, Daniel Kifer, and Zhenhui Li. 2019. A simple baseline for travel time estimation using large-scale trip data. *ACM Trans. Intell. Syst. Technol.* 10, 2 (2019).

[69] Bin Xiao, Haiping Wu, and Yichen Wei. 2018. Simple baselines for human pose estimation and tracking. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 472–487.

[70] Kawin Ethayarajh. 2018. Unsupervised random walk sentence embeddings: A strong but simple baseline. In *Proceedings of the 3rd Workshop on Representation Learning for NLP (Rep4NLP@ACL'18)*. 91–100.
[71] Zheng Xu, Xitong Yang, Xue Li, and Xiaoshuai Sun. 2018. The effectiveness of instance normalization: A strong baseline for single image dehazing. CoRR abs/1805.03305 (2018).
[72] Yaming Tang, Fei Zhao, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2015. Predicting vulnerable components via text mining or software metrics? An effort-aware perspective. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'15)*. 27–36.
[73] Roberto Souto Maior de Barros, Juan Isidro González Hidalgo, and Danilo Rafael de Lima Cabral. 2018. Wilcoxon rank sum test drift detector. *Neurocomputing* 275 (2018), 1954–1963.
[74] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2014. Cross-project defect prediction models: L'Union fait la force. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*. 164–173.
[75] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 373–384.
[76] Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. 2014. An in-depth study of the potentially confounding effect of class size in fault prediction. *ACM Trans. Softw. Eng. Methodol.* 23, 1 (2014).
[77] Suvodeep Majumder, Nikhila Balaji, Katie Brey, Wei Fu, and Tim Menzies. 2018. 500+ times faster than deep learning: A case study exploring faster methods for text mining stackoverflow. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. 554-563.
[78] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educ. Psychol. Meas.* XX, 1 (1960), 37–46.
[79] Joseph L. Fleiss. 1981. The measurement of interrater agreement. In *Statistical Methods Rates Proportions*. John Wiley, NewYork.
[80] Retrieved from https://github.com/Naplues/*MAT*.
[81] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *J. Syst. Softw.* 152, (2019), 70–82.
[82] Supatsara Wattanakriengkrai, Rungroj Maipradit, Hideaki Hata, Morakot Choetkiertikul, Thanwadee Sunetnanta, and Kenichi Matsumoto. 2018. Identifying design and requirement self-admitted technical debt using n-gram IDF. In *Proceedings of the International Workshop on Empirical Software Engineering in Practice (IWESEP'18)*. 7–12.
[83] Maleknaz Nayebi, Yuanfang Cai, Rick Kazman, Guenther Ruhe, Qiong Feng, Chris Carlson, and Francis Chew. 2018. A longitudinal study of identifying and paying down architectural debt. CoRR abs/1811.12904 (2018).
[84] Nathalie Japkowicz and Mohak Shah. 2011. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press.
[85] N. E. Breslow and N. E. Day. 1980. Statistical methods in cancer research: The analysis of case-control studies. *IARC Sci. Public.* 1, 32 (1980), 5–338.
[86] Matthew Hutson. 2018. Artificial intelligence faces reproducibility crisis. *Comput. Sci.* 359, 6377 (2018), 725–726.
[87] Andrew Lane Beam, Arjun K. Manrai, and Marzyeh Ghassemi. 2020. Challenges to the reproducibility of machine learning models in health care. *J. Amer. Med. Assoc.* 323, 4 (2020).
[88] Feng Zhang, Iman Keivanloo, and Ying Zou. 2017. Data transformation in cross-project defect prediction. *Empir. Softw. Eng.* 22, 6 (2017), 3186–3218.
[89] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 631–642.
[90] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. 45–54.
[91] Alex Graves, M. Abdel-rahman, and Geoffrey E. Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. 6645–6649.
[92] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 51–62.
[93] Lin Ma, Zhengdong Lu, and Hang Li. 2016. Learning to answer questions from image using convolutional neural network. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*. 3567–3573.
[94] Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*. Association for Computational Linguistics, 1746–1751.

[95] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. 2020. Identifying self-admitted technical debts with *Jitterbug*: A two-step approach. *IEEE Trans. Softw. Eng.* (2020).

[96] Fei Zhao, Yaming Tang, Yibiao Yang, Hongmin Lu, Yuming Zhou and Baowen Xu. 2015. Is learning-to-rank cost-effective in recommending relevant files for bug localization? In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'15)*. 298–303.

Received June 2020; revised December 2020; accepted January 2021