

Identifying Self-Admitted Technical Debts with Jitterbug: A Two-Step Approach

*Zhe Yu, [†]Fahmid Morshed Fahid, [†]Huy Tu, and [†]Tim Menzies, *Fellow, IEEE*

*Department of Software Engineering, Rochester Institute of Technology, USA

[†]Department of Computer Science, North Carolina State University, USA

Abstract—Keeping track of and managing Self-Admitted Technical Debts (SATDs) are important to maintaining a healthy software project. This requires much time and effort from human experts to identify the SATDs manually. The current automated solutions do not have satisfactory precision and recall in identifying SATDs to fully automate the process. To solve the above problems, we propose a two-step framework called **Jitterbug** for identifying SATDs. **Jitterbug** first identifies the “easy to find” SATDs automatically with close to 100% precision using a novel pattern recognition technique. Subsequently, machine learning techniques are applied to assist human experts in manually identifying the remaining “hard to find” SATDs with reduced human effort. Our simulation studies on ten software projects show that **Jitterbug** can identify SATDs more efficiently (with less human effort) than the prior state-of-the-art methods.

Index Terms—Technical debt, software engineering, machine learning, pattern recognition.

1 INTRODUCTION

Recently, much research has been focused on identifying the “Self-Admitted Technical Debts” (SATDs) from source code comments. Keeping track of and managing these SATDs are important to maintaining a healthy software project as they (1) are diffused in the codebase; (2) increase over time and accumulate interests when not fixed in time; (3) even when fixed, it survives long time (over 1,000 commits on average) in the system [1]; and (4) make the code more difficult to change in the future [2]. What we found in this work is that there are two types of SATDs:

- **The “easy to find” SATDs**, which can be automatically identified without human verification in almost 100% precision. As an example, the comments containing keywords like “*fixme, todo*” are almost always related to SATDs.
- **The “hard to find” SATDs**, which only human experts can accurately decide whether they are SATDs or not. As an example, the comment “*Modify the system class loader instead - horrible! But it works!*” can be easily classified as an SATD by human experts but remains a hard problem for algorithms.

The most important message we want to convey is

Do not waste effort on finding the “easy to find” SATDs, focus more on identifying the “hard to find” SATDs.

Current solutions for identifying SATDs do not separate the two types of SATDs, and belong to either *pattern-based approaches* or *machine learning approaches*. Researchers exploring *pattern-based approaches* first manually inspect code comments and label each one as SATD or non-SATD, then manually analyze the labeled items and summarize patterns for SATDs, e.g. if a comment has keywords like “*hack, fixme, probably a bug*”, then it has a high chance of being related to a SATD. On the other hand,

machine learning approaches first train a classification model on the manually labeled comments, then predict for which comments are related to SATDs (usually on a “hold-out” test set so that performance metrics like precision and recall can be calculated). Limitations exist in both approaches:

- *Pattern-based approaches* require large amounts of human effort in analyzing and summarizing effective patterns.
- Since not all SATDs are “easy to find,” many of the patterns identified by the *pattern-based approaches* from some source projects could be ineffective in a new, unseen project.
- Even the state-of-the-art *machine learning approaches* can only reach around 78% F1 score [3] and 74% AUC [4] (to which the “easy to find” SATDs contribute greatly). That means the process cannot be fully automated without human experts checking the algorithms’ decisions and making the final call.

Acknowledging the existence of the two types of SATDs, we address the SATD identification problem in two steps:

- **Step 1: identify the “easy to find” SATDs automatically.** The comments containing keywords like “*fixme, todo*” are almost always related to SATDs. This suggests that there exist strong patterns that could be used to identify such “easy to find” SATDs automatically, with very high precision. The key challenge of this step is to automatically identify these strong patterns with close to 100% precision so that human experts do not need to verify the results.
- **Step 2: guide human experts to manually read the comments without strong patterns looking for the remaining “hard to find” SATDs.** The remaining “hard to find” SATDs cannot be accurately identified through machine learning algorithms. Human efforts are essential for identifying such SATDs. Therefore, the key challenge of this step is to (1) guide the human effort to the comments that most likely contain SATDs, and (2) provide information such as an estimation of the number of undiscovered SATDs to help human experts make trade-off choices, such as deciding on when to stop the process.

As shown in Figure 1, We designed **Jitterbug**, a two-step frame-

• E-mail: zxyvse@rit.edu

• E-mail: {ffahid, hqtu}@ncsu.edu, timm@ieee.org.

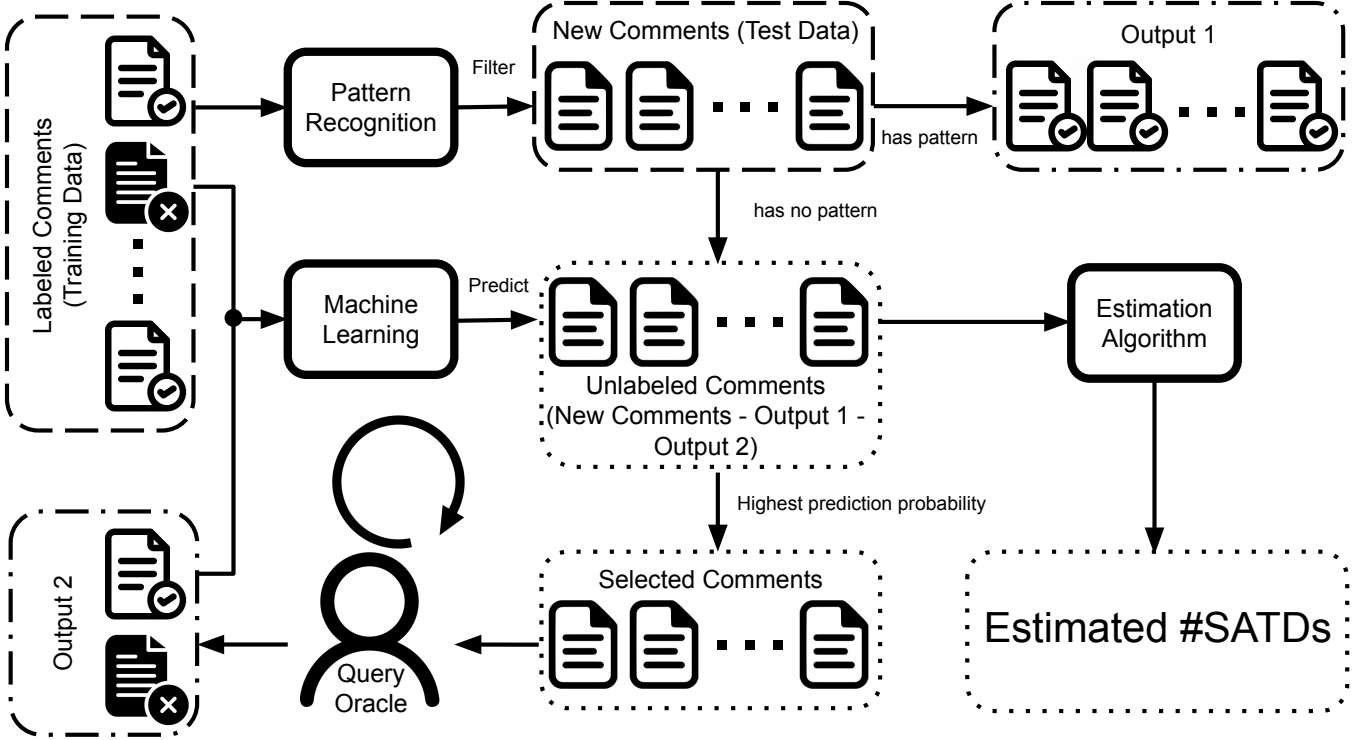


Fig. 1: Workflow of **Jitterbug**.

work. In **Step 1**, **Jitterbug** utilizes a novel pattern recognition technique to identify patterns that could yield very high precision (if one comment has the recorded patterns then there is a close to 100% chance it is related to SATDs). In **Step 2**, Then machine learning models are trained to guide humans to discover SATDs from comments that do not have high precision patterns, as well as to estimate the number of SATDs left in the comments. This idea of separating the SATD identification problem into two steps provides the following advantages: (1) Mining patterns in **Step 1** becomes easy since high recall is no longer a requirement. (2) Human efforts are only spent in **Step 2** on the “hard to find” SATDs—there is zero human effort and consequently negligible cost for finding the “easy to find” SATDs in **Step 1**.

Simulated on the latest SATD dataset from Maldonado and Shihab [5], we ask and answer the following research questions.

RQ1: How to find the strong patterns of the “easy to find” SATDs in Step 1? First, on 9 training projects, a pattern recognizer named **Easy**, with fitness function specifically designed to achieve high precision, is applied to identify patterns with precision higher than 80%. Then, the identified patterns are used on the holdout project to test the performance. We also conduct a validation study by manually analyzing the comments containing the strong patterns found with **Easy** but were labeled as Non-SATDs in the dataset. Interesting findings were discovered during our exploration of this step:

- 1) **Easy** detects the same set of strong patterns— “*todo, fixme, hack, workaround*” for every target project.
- 2) **Easy** achieves close to 100% precision (100% on eight projects and 99% on two projects) on identifying the “easy to find” SATDs. These results are higher than the human-derived set of patterns— “*todo, fixme, hack, xxx*” from Guo et al. [6] (MAT).
- 3) **Easy** is even more accurate than human experts in identify-

ing the “easy to find” SATDs since 98% of the conflicting comments, which were labeled as Non-SATDs by humans but contain the patterns from **Easy**, are identified as SATDs in our validation study.

- 4) Although **Easy** is an algorithm with close to 100% precision and barely any cost (training takes seconds), it alone can only identify 20% to 90% of the SATDs. Thus it is necessary for **Step 2** to find the remaining “hard to find” SATDs.

RQ2: How to find the “hard to find” SATDs efficiently with human experts? After all the “easy to find” SATDs are filtered out, only the “hard to find” SATDs persist in the dataset. First, we will show in RQ2.1 that the “hard to find” SATDs cannot be automatically detected without human oracles. Then the human-in-the-loop approach **Hard** is designed in RQ2.2 and RQ2.3. As shown in Figure 1, a machine learning model is trained to rank the remaining comments. Human oracles are queried for the top rank comments and then those oracles are used to update the machine learning model. This loop will iterate until the target level of recall has been reached in estimation. The advantage of this human-in-the-loop strategy is that the information in both source projects and newly labeled data in the target project can be utilized to better direct human effort towards comments that are more likely to contain SATDs. Our results in RQ2.2 show that this strategy finds more SATDs with fewer human oracles than the state-of-the-art supervised learning techniques where only information from the training datasets is utilized. Meanwhile, RQ2.3 shows that the number of undiscovered SATDs can be accurately estimated by **Hard**, thus helping the human experts make decisions on whether to spend more time looking for the “hard to find” SATDs or to stop at that point.

RQ3: Overall how does Jitterbug perform? We evaluate the overall performance of **Jitterbug** with three sub RQs. In

RQ3.1, it is shown that **Jitterbug** can always find more SATDs with less human effort compared to other state-of-the-art methods as well as **Easy** or **Hard** alone. RQ3.2 evaluates **Jitterbug** when it stops at 90% target recall. Results show that **Jitterbug** always achieves higher recall than another more complex baseline algorithm CNN [3], and **Jitterbug** binary dominates CNN [3] in terms of recall and cost on 2 of the 10 projects. This suggests that **Jitterbug** is a better framework than CNN [3] while the model of CNN [3] has the potential to further improve the performance of **Jitterbug**. In RQ3.3, the computational overhead of **Jitterbug** is similar to that of a traditional supervised learning model (34 seconds), and is much lower than training a deep learning model (3,548 seconds [3]). With a human reviewing 3 comments per minute (estimated with our own experience in classifying SATD comments), on a medium-sized project with 5,000 comments, **Jitterbug** finds in median 97% of the SATDs in 4.5 hours while reviewing all the comments would have cost 28 hours. Therefore, more than 23 hours of human work can be saved for each project when using **Jitterbug**.

1.1 Contributions of this Paper

- 1) In this paper, we show that there are two types of SATDs: the “easy to find” ones that can be identified without human verification, and the “hard to find” ones that only humans can make the final decisions on.
- 2) A novel two-step framework **Jitterbug** is proposed to identify the two types of SATDs. This framework first identifies the “easy to find” SATDs automatically with a novel pattern recognition technique, then applies machine learning techniques to assist human experts in manually identifying the remaining “hard to find” SATDs with reduced human effort.
- 3) A novel pattern recognition technique **Easy** is presented to find strong patterns with close to 100% precision for the “easy to find” SATDs. Results show that its precision is even higher than humans’, thus making it reliable to be applied automatically.
- 4) A continuous learning framework **Hard** is shown to outperform other supervised learning models in retrieving the “hard to find” SATDs with less human effort, and also to provide information on how many more SATDs there are to be found.
- 5) All code and data in this work are available¹, allowing other researchers to replicate, improve, or even refute our findings.

The rest of this paper is structured as follows. Background and related work are discussed in §2. Our methodology is described in §3. This is followed by the details of the SATD datasets in §4. Experiment (simulation) design and answers to the research questions are then presented in §5. In §6, **Jitterbug** is applied to identify SATDs from a real-world, unlabeled software project with a human reading the comments to test its generalizability. Threats to the validity of this work are analyzed in §7. Lastly, conclusion and future work are provided in §8.

2 BACKGROUND AND RELATED WORK

2.1 About Technical Debt

When developers cut corners and make haste to rush out code, that code often contains *technical debt* (TD), i.e. decisions that must be repaid, later on, with further work. Technical debt is like dirt in the gears of software production. As TD accumulates, development becomes harder and slower. Ever since the term technical debt

(TD) was first introduced by Cunningham in 1993 [7], it has been found to be a widespread problem in the software industry damaging many aspects of a system including *evolvability* (how fast we can add new functionality) and *maintainability* (how well we can keep bugs out of the code) [7], [8], [9]:

- In 2012, after interviewing 35 software developers from diverse projects in different companies, varying both in size and type, Lim et al. [10] found developers generate TD due to factors like increased workload, unrealistic deadline in projects, lack of knowledge, boredom, peer-pressure among developers, unawareness or short-term business goals of stakeholders, and reuse of legacy or third party or open-source code.
- After observing five large-scale projects, Wehaibi et al. [2] found that the number of technical debts in a project may be very low (only 3% on average), yet they create a significant amount of defects in the future (and fixing such technical debts are more difficult than regular defects).
- Another study on five software large-scale companies revealed that TDs contaminate other parts of a software system and most of the future interests are non-linear in nature with respect to time [11].
- According to the SIG (Software Improvement Group) study of Nugroho et al. [9], a regular mid-level project owes \$857,500 in TD and resolving TD has a Return On Investment (ROI) of 15% in seven years.
- Guo et al. [8] also found similar results and concluded that the cost of resolving TD in the future is twice as much as resolving immediately.
- As Ozkaya et al. [12] revealed, technical debt affects multiple aspects of the software development process and is mostly invisible.

Therefore, identifying TD has a large impact on software development. However, limited success has been achieved while much research tried to identify TD as part of Code Smells using static code analysis [13], [14], [15], [16], [17]. Static code analysis has a high rate of false alarms while imposing complex and heavy structures for identifying TD [18], [19], [20], [21].

2.2 Identifying Self-Admitted Technical Debt

Recently, much more success has been seen in work on the “self-admitted technical debt” (SATD). Technical debt is often “self-admitted” by the developer in code comments [22], thus making it much easier to find. Identifying and tracking these SATDs have large benefits:

- Removing the SATDs early reduces the maintenance cost of a software project. As reported by Wehaibi et al. [2] in 2016, these SATDs have negative implications on the software development process in particular by making it more difficult to change in the future.
- SATDs can provide cheap training data for learning to identify technical debts (TDs). SATDs are not a specific type of TDs, rather they are the TDs that have been “admitted” by the developers. SATDs also cover the different types of TDs such as code, defect, and requirement debts [1].

In 2014, after studying four large-scale open-source software projects, Potdar and Shihab [22] concluded that developers intentionally leave traces of TD in their comments (saying things like “*hack, fixme, is problematic, this isn’t very solid, probably a bug, hope everything will work, fix this crap*”).

1. <https://github.com/ai-se/Jitterbug>

TABLE 1: Differences between our approach and Guo et al. [6]

	How to find patterns for the “easy to find” SATDs	How to find the “hard to find” SATDs
Guo et al. [6]	Manually find patterns from the test set. Require large amounts of human effort. Performances are tested on the same data used for finding those patterns.	Train a supervised learning model on the training set and test its classification performance on a holdout test set. Users have little control of the recall and precision achieved.
Our approach	Automatically mine patterns from the training set. No human effort cost. Utilize a holdout set to validate the performance of the mined patterns.	Continuously train/update a model on both training set and labeled data from the test set, then use the model to select comments for human experts to read, these human decisions are then used as new labeled data for updating the model. Also apply another model to estimate the total number of SATDs in the comments, thus providing information for the user about what level of recall has been achieved.

2.2.1 Pattern-Based Approaches

Pattern-based approaches [22], [23], [24], [25], [5] consist of three steps: (1) manually inspect code comments and label each one as SATD or non-SATD; (2) manually analyze the labeled items and summarize patterns for SATDs, e.g. if a comment has keywords like “*hack, fixme, probably a bug*”, then it has a high chance of being related to a SATD; (3) apply the summarized patterns to unlabeled comments to identify SATDs.

Potdar and Shihar’s work is now considered the first *pattern-based* approach for identifying SATDs. They found 62 distinct keywords for identifying such TDs [22] (similar conclusions were made by Faris et al. [23], [24], [25]). In 2015, Maldonado et al. used five open-source projects to manually classify different types of SATDs [5] and found

- SATDs mostly contain Requirement Debt and Design Debt in source code comments;
- 75% of SATDs get removed, but the median lifetime of SATDs ranges from 18 to 173 days [26].

Another study tried to find the SATD-introducing commits in Github using different features on change level [27]. Instead of using the bag of word approach, a recent study also proposed word embedding as a vectorization technique for identifying SATD [28]. These *pattern-based* studies focused on identifying keywords in code comments indicating SATDs and then used those keywords to label comments as SATDs [29].

There are risks and problems to this approach. First, it requires much manual effort from human experts to find those keywords by reading thousands of comments. Second, it is natural to believe that such keywords can vary from projects to projects and will not produce 100% precision and recall but none of the studies used a holdout set to evaluate the precision and recall of using such keywords to identify SATDs.

2.2.2 Machine Learning Approaches

To solve the above mentioned problems, *machine learning* [30], [31], [32], [4] approaches are proposed for identifying SATDs. In these approaches, supervised learning models are trained on labeled SATD datasets to learn the underlying rules of comments admitting TDs. For example, Tan et al. [33], [34] analyzed source code comments using natural language processing to understand programming rules and documentations and indicates comment quality and inconsistency. A similar study was done by Khamis et al [35]. After analyzing and categorizing comments in source code, Steidl et al. [36] proposed a machine learning technique that can measure the comment quality according to category. Malik et al. [37] used a random forest classifier to understand the lifetime of code comments. A similar study on three open-source projects was also done by Fluri et al. [38]. In 2017,

Maldonado et al. [30] successfully identified two types of SATD in 10 open-source projects (average 63% F1 Score) using Natural Language Processing (Max Entropy Stanford Classifier) using only 23% training data. A different approach was introduced by Huang et al. [31] in 2018. Using eight datasets, Huang et al. build a Multinomial Naive Bayes sub-classifier for each training dataset using information gain as feature selection. By implementing a boosting technique using all those sub-classifiers, they have found an average of 73% F1 scores for all datasets [32]. A recent IDE for Eclipse was also released using this technique for identifying SATD in java projects [31]. More recently, Zampetti et al. [4] reported an average precision of 55%, recall of 57%, and AUC of 0.73 with a deep learning-based approach. Recently, some studies explore different feature engineering for identifying SATDs, e.g. Wattanakriengkrai et al. [39] applied N-gram IDF as features, and Flisar and Podgorelec [40] explored how feature selection with word embedding can help the prediction. The latest progress from Ren et al. [3] utilized a deep convolutional neural network with hyperparameter tuning to achieve a higher F1 score than all the previous solutions.

These machine learning models can be a good indicator for which comments are more likely to be related to SATDs. However, with precision ranging from 60% to 85%, it is not reliable to fully automate the process. Human experts are then required to verify every decision the machine learning model made and thus costs a large amount of time and labor but still finding only, say 57% of the SATDs.

2.2.3 Two-Step Approaches

As described in §1, we take a two-step approach to identify SATDs: (1) identify patterns for the “easy to find” SATDs with close to 100% precision and automatically classify comments with the patterns as SATDs (without human verification); (2) then apply machine learning techniques to guide human experts to find the remaining “hard to find” SATDs with least number of comments read. Interestingly, during the drafting of this paper, we found a preprint [6] that utilized a similar idea to our two-step approach. Guo et al. [6] used four keywords (“*fixme, todo, hack, xxx*”) to identify the “easy to find” SATDs and applied supervised learning models to find the remaining “hard to find” SATDs. Although Guo et al. consider their approach as just a strong baseline, it still demonstrates the effectiveness of such two-step approaches. The differences between our approach and Guo et al.’s are listed in Table 1. More detailed comparisons, along with other state-of-the-art machine learning algorithms will be presented in §5.

3 METHODOLOGY

As shown in Figure 1, **Jitterbug** consists of two operators— a pattern recognizer **Easy** and a continuous learning model **Hard**.

To find all possible strong patterns in Step 1, we featurize the data as a term frequency matrix without stemming or stop word removal. This section breakdowns the workflow as shown in Algorithm 1 and introduces the two operators in detail.

Algorithm 1: Psuedo Code for Jitterbug.

```

Input :  $X$ , set of training data.
          $Y$ , set of test data.
          $T_{rec}$ , target recall of the "hard to find" SATDs.
          $CL$ , the machine learning model applied.
Output :  $TD$ , set of SATDs identified from test data.

1 Function Jitterbug ( $X, Y, T_{rec}, CL$ )
   // Extract patterns from training data.
2    $patterns \leftarrow \text{Easy}(X)$ ;
   // Identify the "easy to find" SATDs.
3    $TD_{easy} \leftarrow \text{Has\_Pattern}(Y, patterns)$ ;
   // Remove "easy to find" SATDs from training and test
   data.
4    $Y_{hard} \leftarrow Y \setminus TD_{easy}$ ;
5    $X_{hard} \leftarrow X \setminus \text{Has\_Pattern}(X, patterns)$ ;
   // Identify the "hard to find" SATDs.
6    $TD_{hard} \leftarrow \text{Hard}(X_{hard}, Y_{hard}, T_{rec})$ ;
7    $TD \leftarrow TD_{easy} \cup TD_{hard}$ ;
8   return  $TD$ ;

```

3.1 Easy

Algorithm 2: Psuedo Code for Easy.

```

Input :  $X$ , set of training data.
Output :  $patterns$ , list of identified patterns.

1 Function Easy ( $X$ )
   // Set precision threshold as a stopping rule.
2    $thres \leftarrow 0.8$ ;
3    $patterns \leftarrow []$ ;
4   while True do
   // Find the pattern of highest fitness score.
5    $scores \leftarrow \{ p : \text{FitnessFunction}(X, p) \text{ foreach } p \in \text{All\_Patterns}(X) \}$ ;
6    $p \leftarrow \text{argmax}(scores)$ ;
   // Check if highest precision is below the
   threshold.
7   if  $\text{Precision}(X, p) < thres$  then
8     break;
   // Add p as one of the strong patterns.
9    $patterns.append(p)$ ;
   // Remove comments that contain p.
10   $X.remove(\text{Has\_Pattern}(X, p))$ ;
11  return  $patterns$ ;

12 Function FitnessFunction ( $X, p$ )
   // Calculate the fitness score of input pattern.
13   $P, TP \leftarrow \text{Metrics}(X, p)$ ;
14   $score \leftarrow TP^4 / P^3$ ;
15  return  $score$ ;

16 Function Precision ( $X, p$ )
   // Calculate the precision of input pattern.
17   $P, TP \leftarrow \text{Metrics}(X, p)$ ;
18   $prec \leftarrow TP / P$ ;
19  return  $prec$ ;

20 Function Metrics ( $X, p$ )
   // Calculate # Positives and # True Positives.
21   $Ps \leftarrow \text{Has\_Pattern}(X, p)$ ;
22   $TPs \leftarrow \text{Is\_SATD}(Ps)$ ;
23  return  $|Ps|, |TPs|$ ;

```

Pattern Recognition is an engineering application of Machine Learning. Machine Learning deals with the construction and study of systems that can learn from data, rather than follow only explicitly programmed instructions whereas Pattern recognition is the recognition of patterns and regularities in data [41]. Here

in **Jitterbug**, the task of the pattern recognizer **Easy** is to find the strong patterns of the "easy to find" SATDs (**RQ1**). For each potential pattern (a keyword in the comments in our case), we measure two metrics:

- $P(p)$: the number of comments containing the pattern p (positives).
- $TP(p)$: the number of SATD comments containing the pattern p (true positives).

Derived from the above two metrics, we also have

- $Prec(p) = TP(p)/P(p)$: precision of the pattern p .

To achieve high reliability and thus fully automate the process, we want to find those patterns with very high precision. On the other hand, we also want to avoid rare patterns, e.g. if a pattern only appears once, it is not useful even with 100% precision. As a result, we define our fitness function as

$$Fitness(p) = Prec(p)^N \cdot P(p) = TP(p)^N / P(p)^{N-1} \quad (1)$$

We set $N = 4$ to find patterns with close to 100% precision. Using our labeled training data, the pattern recognizer looks for the pattern with the highest fitness score, then removes comments containing that pattern from the training data and finds the next pattern with the highest fitness score (re-calculated). The detailed algorithm is shown in Algorithm 2.

3.2 Hard

As shown in Algorithm 3, **Hard** utilizes a machine learner to continuously learn from both labeled data in the source projects and human decisions of comments in the target project. This machine learner in **Hard** can be any supervised learner in theory. However, since it will be updated/re-trained frequently, we only consider models that can be trained within seconds (users will not wait for more than a few seconds every time they finish a batch of comments). For this reason, we only test the following fast and simple learners listed below.

Logistic Regression: Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable [42]. A standard logistic function is a common "S" shape with Equation (2):

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}} \quad (2)$$

where $p(x) \in (0, 1)$ for all t . Through fitting on the training data, logistic regression looks for the best parameter β to classify input data x into two target classes $\{0, 1\}$.

Decision Tree: Decision tree learning is a method commonly used in data mining which uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). Algorithms for constructing decision trees usually work top-down, by choosing a variable at each step that best splits the set of items [43]. Two metrics are commonly applied to determine the best split:

- **Gini impurity:** $I_G(p) = \sum_{i=1}^J p_i(1 - p_i)$.

- **Entropy:** $I_E(p) = \sum_{i=1}^J p_i \log_2 p_i$.

Where J is the number of classes and p_i is the fraction of items labeled with class i in the training dataset. The algorithm will find the best split after which the value of $I_G(p)$ or $I_E(p)$ decreases the most. In this paper, we use Gini impurity.

Algorithm 3: Psuedo Code for Hard

Input : X_{hard} , labeled training data containing “hard to find” SATDs.
 Y_{hard} , unlabeled test data containing “hard to find” SATDs.
 T_{rec} , target recall (as stopping rule).

Output : $T D_{hard}$, “hard to find” SATDs identified.

```

1 Function Hard ( $X_{hard}, Y_{hard}, T_{rec}$ )
2    $Y_{labeled} \leftarrow \emptyset$ ;
3    $T D_{hard} \leftarrow \emptyset$ ;
4   // Each time query the oracle for 10 comments.
5    $K \leftarrow 10$ ;
6   while True do
7     // Train the machine learning model.
8      $CL.fit(X_{hard} \cup Y_{labeled})$ ;
9     // Estimate # “hard to find” SATDs.
10     $|R_E| \leftarrow \text{Estimate}(CL, Y_{hard}, Y_{labeled})$ ;
11    // Check if target recall has been reached.
12    if  $|T D_{hard}| / (|T D_{hard}| + |R_E|) \geq T_{rec}$  then
13      break;
14    // Select comments with top K prediction
15    probability.
16     $Q \leftarrow \text{argsort}(CL.decision\_function(Y_{hard} \setminus Y_{labeled}))[:K]$ ;
17     $Y_{labeled} \leftarrow Y_{labeled} \cup Q$ ;
18    // Query oracles for the selected comments.
19     $T D_{hard} \leftarrow T D_{hard} \cup \text{ls\_SATD}(Q)$ ;
20  return  $T D_{hard}$ ;

14 Function Estimate ( $CL, Y_{hard}, Y_{labeled}$ )
15 if  $|Y_{labeled}| == 0$  then
16   return NaN;
17  $|R_E|_{last} \leftarrow 0$ ;
18  $Y_{unlabeled} \leftarrow Y_{hard} \setminus Y_{labeled}$ ;
19 foreach  $x \in Y_{hard}$  do
20    $D(x) \leftarrow CL.decision\_function(x)$ ;
21   if  $x \in Y_{labeled}$  and  $\text{ls\_SATD}(x)$  then
22      $L(x) \leftarrow 1$ ;
23   else
24      $L(x) \leftarrow 0$ ;
25  $|R_E| \leftarrow \sum_{x \in Y_{hard}} L(x)$ ;
26 while  $|R_E| \neq |R_E|_{last}$  do
27   // Fit and transform Logistic Regression
28    $\text{LogisticRegression}.fit(D(Y_{hard}), L(Y_{hard}))$ ;
29    $LReg(Y_{unlabeled}) \leftarrow \text{LogisticRegression}.predict\_proba(D(Y_{unlabeled}))$ ;
30    $L \leftarrow \text{TemporaryLabel}(LReg, L)$ ;
31    $|R_E|_{last} \leftarrow |R_E|$ ;
32   // Estimation based on temporary labels
33    $|R_E| \leftarrow \sum_{x \in Y_{hard}} L(x)$ ;
34 return  $|R_E|$ ;

33 Function TemporaryLabel ( $LReg, L$ )
34  $count \leftarrow 0$ ;
35  $target \leftarrow 1$ ;
36  $can \leftarrow []$ ;
37 // Sort  $Y_{unlabeled}$  by descending order of  $LReg$ 
38  $Y_{unlabeled} \leftarrow \text{argsort}(LReg)[::-1]$ ;
39 foreach  $x \in Y_{unlabeled}$  do
40    $count \leftarrow count + LReg(x)$ ;
41    $can.append(x)$ ;
42   if  $count \geq target$  then
43      $L(can[0]) \leftarrow 1$ ;
44      $target \leftarrow target + 1$ ;
45      $can \leftarrow []$ ;
46 return  $L$ ;

```

Random Forest: Random forest classifier is an ensemble learning method that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes of the individual trees [44]. Each decision tree from the random forest model is independently trained on all the training data but with only a subset of the features. In this way, these decision trees are 100% accurate on training data and yet have different generalization errors. When used together for inference, these decision trees correct for each other’s generalization errors and thus avoid overfitting on the training data.

Naive Bayes: Naive Bayes classifiers are a family of simple “probabilistic classifiers” based on Bayes’ theorem with strong (naive) independence assumptions between the features [45]. With the strong assumption that all features are mutually independent, a Naive Bayes classifier predicts the conditional probability of data x belonging to class C_i to be

$$p(C_k | x_1, \dots, x_n) \propto p(C_k) \prod_{i=1}^n p(x_i | C_k) \quad (3)$$

where $p(C_k)$ and $p(x_i | C_k)$ are counted from the training data. Multinomial Naive Bayes model assumes that each $p(x_i | C_k)$ is a multinomial distribution, which works well for text data.

Support Vector Machine: A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane [46]. Soft-margin linear SVMs are commonly used in text classification given the high dimensionality of the feature space. A soft-margin linear SVM looks for the decision hyperplane that maximizes the margin between training data of two classes while minimizing the training error (hinge loss):

$$\min \lambda \|w\|^2 + \left[\frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i(w \cdot x_i - b)) \right] \quad (4)$$

where the class of x is predicted as $\text{sgn}(w \cdot x - b)$.

Hard also utilizes an estimator to estimate the number of “hard to find” SATDs and thus determine when to stop. This estimator, also described in Algorithm 3, is adopted from our previous work [47] where it was shown to outperform any other state-of-the-art estimators. The idea behind this estimator is that it (1) assigns temporary labels to unlabeled data points following the probability prediction from a logistic regression model, (2) then updates that logistic regression model on the temporary labeled data, (3) iterates the above two steps until convergence (when the number of temporarily assigned labels stays unchanged).

4 DATASETS

While §3 shows how **Jitterbug** should be applied in practice with human reading source code comments looking for the “hard to find” SATDs, it is too expensive for humans to test different treatments and answer all the research questions. As a result, the performance of **Jitterbug** is tested through simulations on a publicly available SATD dataset originally collected by Maldonado and Shihab [5]. This dataset contains ten open-source java projects on different application domains (five of these projects were added by the same authors later after its first release), varying in size and the number of developers and most importantly, in the number of comments in source code. All of these ten projects, namely Apache-Ant-1.7.0, Apache-Jmeter-2.10, ArgoUML, Columba-1.4-src, EMF-2.4.1, Hibernate-Distribution-3.3.2.GA, jEdit-4.2, jFreeChart-1.0.19, jRuby-1.4.0, SQL12 were

collected from GitHub. The provided dataset contains project names, classification type (if any) with actual comments. Note that, our problem does not concern with the type of SATD, rather we care about a binary problem of being a SATD or not. So, we have changed the final label into a binary problem by defining *WITHOUT_CLASSIFICATION* as *no* and the rest (for example *DESIGN*) as *yes*. A few examples from the dataset are given in Table 2 for readers’ ease.

TABLE 2: Examples from Dataset

project	classification	commenttext	label
Apache Ant	DEFECT	// FIXME formatters are not thread-safe	yes
EMF	IMPLEMENTATION	// TODO Binary incompatibility; an old override must override putAll.	yes
JFreeChart	DESIGN	// calculate the adjusted data area taking into account the 3D effect... this assumes that there is a 3D renderer, all this 3D effect is a bit of an ugly hack...	yes
JRuby	WITHOUT CLASSIFICATION	// build first node (and ignore its result) and then second node	no
Columba	WITHOUT CLASSIFICATION	// get message header	no
JMeter	WITHOUT CLASSIFICATION	// parameters to pass to script file (or script)	no

4.1 Independent Variables

When Maldonado and Shihab [5] created this dataset, jDeodrant [48] was applied, which is an Eclipse plugin for extracting comments from the source code of java files. After that, Maldonado and Shihab [5] used four filtering heuristics to the comments. A short description of the filtering heuristics is given below.

- Removed licensed comments, auto-generated comments, etc. because according to the dataset authors, they do not contain SATD by developers.
- Removed commented source codes as commented source codes do not contain any SATD.
- Removed Javadoc comments that do not contain the words such as “todo”, “fixme”, “xxx” etc. because according to the dataset authors, the rest of the comments rarely contain any SATDs.
- Multiple single-line comments are grouped into a single comment because they all convey a single message and it is easy to consider them as a group.

After Maldonado and Shihab [5] applied these heuristics, the number of comments in each project reduced significantly (for example, the number of comments in Apache Ant reduced from 21,587 to 4140, almost 19% of the original size).

4.2 Dependent Variables

In the work of Maldonado and Shihab [5], two humans then manually classified each comment according to the six different types of TD mentioned by Alves et al. [49] if they contained any SATD at all, else marked them *WITHOUT_CLASSIFICATION*. Stratified sampling of the dataset was applied to check personal bias and found a 99% confidence level with a confidence interval of 5%. A third human verified the agreement between the two using stratified sampling and reported a high level of agreement, using Cohen’s Kapp [50] coefficient of +0.81. Such a high confidence level, as well as a higher level of agreement indicates that the dataset is unbiased and reliable. A detailed description of the dataset is given in in Table 3.

TABLE 3: Dataset Details

Project	Release / Year	Domain	Comments	SATDs	Ratio
Apache Ant	1.7.0 / 2006	Automating Build	4098	131	3.2%
JMeter	2.10 / 2013	Testing	8057	374	4.64%
ArgoUML	-	UML Diagram	9452	1413	14.95%
Columba	1.4 / 2007	Email Client	6468	204	3.15%
EMF	2.4.1 / 2008	Model Framework	4390	104	2.37%
Hibernate	3.3.2 / 2009	Object Mapping Tool	2968	472	15.90%
JEdit	4.2 / 2004	Java Text Editor	10322	256	2.48%
JFreeChart	1.0.19 / 2014	Java Framework	4408	209	4.74%
JRuby	1.4.0 / 2009	Ruby for Java	4897	622	12.70%
Squirrel	-	Database	7215	286	3.96%
SUM			62275	4071	6.54%
MEDIAN			5682.5	271	4.77%

5 EXPERIMENTS AND RESULTS

Experiments are conducted on the SATD dataset with 10 projects described in §4. Each time, one project is selected as a target project (with labels unknown) and the rest 9 datasets are treated as source projects (with labels known). In Step 2, when oracles are queried for the target project, the ground truth labels are applied to label the queried comments, thus simulating the human-in-the-loop process without a real human in the loop. The rest of this section will provide details on the experiments and results for answering the research questions listed in §1.

5.1 RQ1: How to find the strong patterns of the “easy to find” SATDs in Step 1?

In this experiment, we compare the performance of the following two treatments:

- **Easy:** The pattern recognizer of **Jitterbug** described in Algorithm 2, which iteratively selects the pattern with the highest fitness score in (1) until the selected pattern has lower than 80% precision on the training data.
- **MAT:** a baseline approach from Guo et al. [6] where a set of human-derived patterns— “*todo*, *fixme*, *hack*, *xxx*” is applied to find the “easy to find” SATDs.

on three different performance metrics

- **Precision:** $Precision = TP / (TP + FP)$.
 - **Recall:** $Recall = TP / (TP + FN)$.
 - **F1 score:** $F_1 = 2 \cdot Precision \cdot Recall / (Precision + Recall)$.
- where TP is the number of true positives (SATD comments predicted as SATDs), FP is the number of false positives (non-SATD comments predicted as SATDs), and FN is the number of false negatives (SATD comments predicted as non-SATDs).

Table 4 (**Original**) shows the results of the experiment on the original dataset. Our observation from the results are

- 1) Choosing any project as the holdout set, the strong patterns discovered by the pattern recognizer are always the same— “*todo*, *fixme*, *hack*, *workaround*”, except for *JRuby* where the strong patterns are “*fixme*, *hack*”.

TABLE 4: Experimental results for Step 1 on every targeting project. **Easy** represents the pattern recognizer in **Jitterbug** while **MAT** is a baseline approach from Guo et al. [6]

that uses human-derived patterns—“*todo, fixme, hack, xxx*” to find SATDs. The column **Better** summarizes how many times one treatment is better than the other on the given metric. This table presents results on two sets of ground truth labels: (1) **Original**: the ground truth labels provided by Maldonado and Shihab [5], and (2) **Corrected**: labels after validating the conflicts between **Easy** and original ground truth, as shown in Figure 2.

Ground Truth	Metrics	Treatment	Squirrel	JMeter	EMF	Apache Ant	ArgoUML	Hibernate	JUnit	JFreeChart	Columba	JRuby	Better
Original	Precision	Easy	0.85	0.87	0.69	0.89	0.85	0.94	0.95	0.72	0.91	0.93	3
		MAT	0.85	0.87	0.67	0.90	0.85	0.94	0.81	0.72	0.91	0.92	1
	Recall	Easy	0.54	0.75	0.33	0.24	0.88	0.74	0.21	0.47	0.87	0.52	5
		MAT	0.54	0.75	0.29	0.47	0.88	0.73	0.19	0.46	0.86	0.90	2
	F1	Easy	0.66	0.80	0.44	0.38	0.87	0.83	0.35	0.57	0.89	0.67	6
		MAT	0.66	0.80	0.40	0.62	0.86	0.82	0.30	0.56	0.88	0.91	2
Corrected	Precision	Easy	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	0.99	1.00	6
		MAT	1.00	0.99	1.00	0.96	0.99	0.99	0.85	1.00	0.99	0.99	0
	Recall	Easy	0.58	0.77	0.41	0.27	0.90	0.75	0.22	0.55	0.88	0.90	4
		MAT	0.58	0.77	0.38	0.49	0.90	0.74	0.19	0.55	0.87	0.91	2
	F1	Easy	0.74	0.87	0.58	0.42	0.94	0.86	0.37	0.71	0.93	0.95	4
		MAT	0.73	0.87	0.55	0.65	0.94	0.85	0.31	0.71	0.93	0.95	1

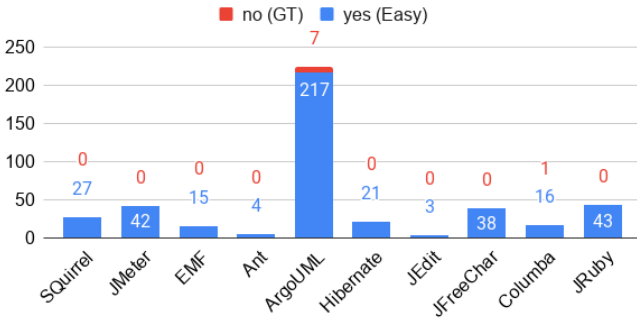


Fig. 2: Validation results for double-checking false positives of **Easy**. GT means the original ground truth label and DC means the double-checking result. The values of “yes (Easy)” show the number of comments that the double-checking result agrees with the **Easy** results (Easy=yes AND GT=no AND DC=yes) while the values of “no (GT)” show the number of comments that the double-checking result agrees with the original ground truth labels (Easy=yes AND GT=no AND DC=no). This graph shows that most (426 out of 434) of the false positives are actually true positives that were previously wrongly labeled in the original dataset.

- 2) Compared with manually discovered patterns—“*todo, fixme, hack, xxx*” from Guo et al. [6] (MAT), the patterns automatically learned by **Easy** showed higher or similar precision and recall on 8 out of 10 target projects.

The results above suggest that our automated pattern recognizer **Easy** performed better than the human-derived patterns from Guo et al. [6] (MAT). However, it did not reach close to 100% precision on many target projects as we expected. One possible reason for this is human errors—labels in the original dataset may not always be correct. Therefore, we manually analyzed the false positives (comments containing the strong patterns but were labeled as Non-SATDs) of **Easy** to double-check their labels. Two graduate students were employed to classify the 434 (out of 62,275, 7%) comments where the original ground truth labels (GT) are **no** but the **Easy** predictions are **yes**. Surprisingly, the two graduate students found the comments very easy to classify and both made the same classification. Table 5 shows some example comments whose labels were flipped. As shown in Figure 2, most

of the false positives (98%) were wrongly labeled in the original dataset. That means these strong patterns identified by **Jitterbug** are even more accurate than human experts in finding the “easy to find”.

After the ground truth labels were corrected, we reran the experiments and collected results in Table 4 (**Corrected**). This time, we observe:

- 1) **Easy** detects the same set of strong patterns—“*todo, fixme, hack, workaround*” for every target project including **JRuby** after correcting the human errors. This also greatly increases the recall on **JRuby**.
- 2) **Easy** achieves close to 100% precision (100% on eight projects and 99% on two projects) on identifying the “easy to find” SATDs. These results are higher than the human-derived set of patterns—“*todo, fixme, hack, xxx*” from Guo et al. [6] (MAT).
- 3) **Easy** achieves much lower recall and F1 score than **MAT** on **Apache Ant**. This is because only on the **Apache Ant** project, “xxx” is a strong pattern of fitness score 25. On the other projects, fitness scores for “xxx” range from 0 to 2. Therefore, the pattern of “xxx” can help only on **Apache Ant** and will damage the precision when used on other projects. This is exactly the advantage of **Easy** over **MAT**—to avoid such “trap” patterns like “xxx” by training on a collection of projects.
- 4) **Easy** is even more accurate than human experts in identifying the “easy to find” SATDs since 98% of the conflicting comments, which were labeled as Non-SATDs by humans but contain the patterns from **Easy**, are identified as SATDs in our validation study.
- 5) Although **Easy** is an algorithm with close to 100% precision and barely any cost (training takes seconds), it alone can only identify 20% to 90% of the SATDs. Thus it is necessary for **Step 2**.

5.2 RQ2: How to find the “hard to find” SATDs efficiently with human experts?

After all the “easy to find” SATDs are filtered out from the datasets, it is now a problem to find the remaining “hard to find” SATDs (which are 10-80% of all the SATDs). To solve this problem, we first ask the following question.

TABLE 5: Examples of Corrected Labels

Project	Comment Text	GT	Easy
Apache Ant	//TODO Test on other versions of weblogic //TODO add more attributes to the task, to take care of all jspc options //TODO Test on Unix	no	yes
ArgoUML	// skip backup files. This is actually a workaround for the cpp generator, which al- ways creates backup files (it's a bug).	no	yes
JFreeChart	// FIXME: we've cloned the chart, but the dataset(s) aren't cloned and we should do that	no	yes
JRuby	// All errors to sysread should be SystemCallEr- rors, but on a closed stream Ruby returns an IOError. Java throws same exception for all errors so we resort to this hack...	no	yes
Columba	// FIXME r.setPos();	no	yes

5.2.1 RQ2.1: Can the “hard to find” SATDs be automatically detected without human oracles?

To answer this question, we trained supervised learning models on source projects and tested them on the target project (both source projects and target projects do not contain comments that have the patterns identified by Step 1). The following models are evaluated in this experiment:

- **LR**: logistic regression model described in §3.2. Implemented with scikit-learn² package *LogisticRegression* in Python with balanced class weight.
- **DT**: decision tree model described in §3.2. Implemented with scikit-learn package *DecisionTreeClassifier* in Python with balanced class weight.
- **RF**: random forest model described in §3.2. Implemented with scikit-learn package *RandomForestClassifier* in Python with class_weight = balanced_subsample.
- **SVM**: linear soft-margin support vector machine model described in §3.2. Implemented with scikit-learn package *SGDClassifier* in Python with balanced class weight.
- **NB**: Multinomial Naive Bayes model described in §3.2. Implemented with scikit-learn package *MultinomialNB* in Python.
- **TM**: ensemble model from Huang et al. [32] where a Multinomial Naive Bayes model is trained on selected features with information gain on each source project and the majority vote of the predictions from each model on the target project is utilized to make the final prediction. **TM** is considered as one of the state-of-the-art solutions for identifying SATDs so here we apply it as a baseline algorithm.

To assess whether the above supervised learning models can identify the “hard to find” SATDs precisely without human oracles, we apply the following **P@10** metric:

- **P@10**: precision for the top 10 predictions. For example, if there are 6 SATDs amongst the 10 comments with the highest prediction probability, $P@10 = 0.6$. We choose this metric since a model is definitely imprecise if the precision for its top 10 confident predictions is already low.

Figure 3 shows that at most three out of ten projects have **P@10** higher than 0.5. That means all of the machine learning models predict wrongly with higher than 50% probability even for the 10 most confident predictions on most of the projects. Therefore, we conclude that directly using the model predictions as classification results will result in a large number of false positives. As a result, human experts have to verify each prediction result to decide whether it is SATD or not. This leads to our next research question.

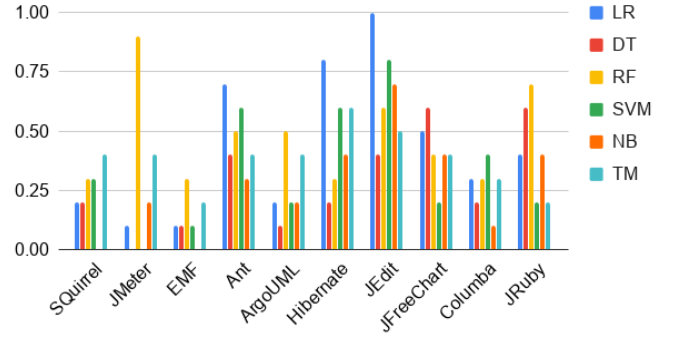


Fig. 3: P@10 results for supervised learning models on the “hard to find” SATDs. At most three out of ten projects have P@10 higher than 0.5. This suggests that all these supervised learners are not precise enough to fully automate the process of identifying the “hard to find” SATDs. Human oracles have to be queried to make the final decisions.

5.2.2 RQ2.2: How to more efficiently utilize human oracles to find the “hard to find” SATDs?

Since it is inevitable to spend human effort on verifying the prediction results in Step 2, the **Hard** strategy is applied to learn from this incrementally acquired information and update its model for better predictions, as described in Algorithm 3. In this experiment, we record the recall and its corresponding cost (of human effort) for each algorithm as the cost increases.

$$Recall = \frac{|\{\text{SATDs}\} \cap \{\text{human verified comments}\}|}{|\{\text{SATDs}\}|} \quad (5)$$

$$Cost = \frac{|\{\text{human verified comments}\}|}{|\{\text{comments}\}|} \quad (6)$$

To simplify the comparison between different algorithms, we calculate the area under the recall-cost curve as a performance metrics:

- **APFD**: first proposed in test case prioritization [51]. APFD calculates the area under the recall-cost curve. Ranging from 0.0 to 1.0, a larger APFD means higher recall can be achieved at a lower cost, thus the better. An APFD of 0.5 can be achieved by randomly select the next item each time. We choose to evaluate the overall performance with **APFD** since it is a single metric that evaluates the entire recall-cost curve, while other metrics (e.g. precision, recall, F1 score) only evaluate a single point of the curve. In this way, we do not need to decide the stopping point before comparing different treatments.

Table 6 shows the APFD results for different models with or without the **Hard** strategy. Given that most of the results are deterministic and are close to each other, we applied Cohen’s effect size test to determine which results are similar. To that end, we calculated:

$$Small_{step2} = 0.2 \cdot StdDev(\text{All APFD results}) = 0.02. \quad (7)$$

We then consider all the results that are higher than the best result minus the $Small_{step2}$ as the best results on each target project (colored in gray in Table 6). From these results we can see

- Continuously updating the model (**Hard**) helps improve the performance on 4 out of 5 models (except for the decision tree model).

2. <https://scikit-learn.org/>

TABLE 6: APFD (higher the better) results for different models with or without the **Hard** strategy on the “hard to find” SATDs. Medians and IQRs (delta between 75th percentile and 25th percentile, lower the better) are calculated for easy comparisons. If **Hard** = no, human oracles on the target project are not utilized, the model is just a one-time trained supervised learning model. On the other hand, if **Hard** = yes, human oracles on the queried comments are utilized to update the model before it is applied to find its next highest predictions for humans to verify. **Jitterbug** utilizes **Hard** = yes. A threshold of Cohen’s small effect size (0.02) is applied to determine which treatment performs best in each target project and color them in **gray**. The column **#Best** shows the number of projects each treatment performs the best in.

Model	Hard	Squirrel	JMeter	EMF	Apache Ant	ArgoUML	Hibernate	JUnit	JFreeChart	Columna	JRuby	Median	IQR	#Best
LR	no	0.69	0.85	0.80	0.88	0.93	0.80	0.80	0.80	0.90	0.83	0.82	0.07	2
	yes	0.74	0.84	0.83	0.89	0.93	0.81	0.85	0.81	0.92	0.83	0.84	0.07	3
DT	no	0.78	0.78	0.84	0.78	0.88	0.76	0.86	0.71	0.87	0.86	0.81	0.08	0
	yes	0.71	0.64	0.71	0.77	0.8	0.77	0.75	0.53	0.78	0.81	0.76	0.07	0
RF	no	0.82	0.79	0.81	0.83	0.88	0.78	0.82	0.73	0.91	0.83	0.82	0.03	0
	yes	0.91	0.85	0.93	0.90	0.96	0.83	0.92	0.89	0.98	0.91	0.91	0.04	9
SVM	no	0.58	0.83	0.75	0.83	0.90	0.75	0.71	0.74	0.84	0.74	0.75	0.09	0
	yes	0.91	0.86	0.91	0.90	0.92	0.83	0.95	0.79	0.97	0.92	0.91	0.05	8
NB	no	0.41	0.74	0.67	0.62	0.75	0.57	0.37	0.63	0.65	0.67	0.64	0.09	0
	yes	0.47	0.75	0.67	0.66	0.79	0.59	0.42	0.64	0.66	0.68	0.66	0.08	0
TM	no	0.73	0.7	0.72	0.80	0.69	0.75	0.77	0.69	0.77	0.89	0.74	0.07	0

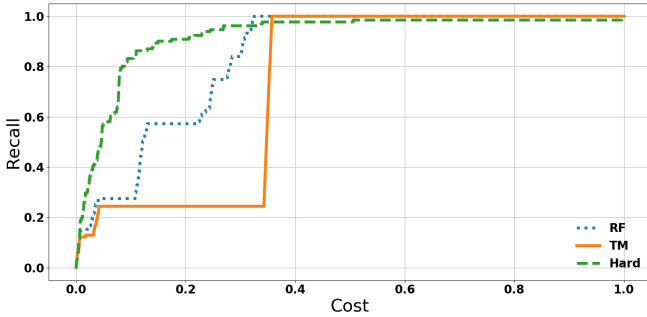


Fig. 4: Recall-cost curves for three different treatments on finding the “hard to find” SATDs on target project Squirrel. **Hard** represents **RF** with **Hard**=yes while **RF** represents **RF** with **Hard**=no. APFD results in Table 6 were calculated as the area under these curves. Figures on other target projects are shown in the Appendix as Figure A.1.

- Random forest and support vector machine models with the **Hard** strategy achieved the highest median APFD of 0.91. Given that the **#Best** of random forest is higher, we choose random forest as the internal model of **Hard** for the rest of the experiments.
- Random forest model with the **Hard** strategy outperformed the baseline algorithm **TM** on finding the “hard to find” SATDs.

For a more intuitive comparison, Figure 4 shows the recall-cost curves of three different treatments on target project Squirrel. The APFD results in Table 6 were calculated as the area under these curves. As we can see, **Hard** (**RF** with **Hard**=yes) has the highest APFD score of 0.91. Also, in Figure 4 it almost always reaches the same recall at a lower cost than **RF** (**RF** with **Hard**=no) with APFD score of 0.82 and **TM** with APFD score of 0.73. Recall-cost curves on other target projects can be found in Figure A.1 from the Appendix.

5.2.3 RQ2.3: When to stop Hard in Step 2?

RQ2.2 shows that overall, **Hard** achieves higher recall at a lower cost than other methods. However, it is still necessary to decide a stopping point of **Hard** in real-world applications. Our solution to

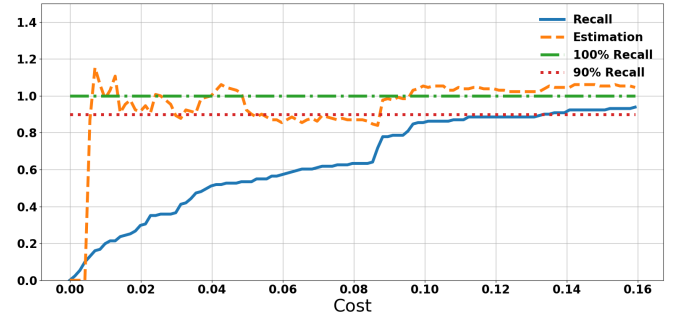


Fig. 5: Recall-cost and estimation-cost curves for finding 90% of the “hard to find” SATDs with **Hard** on target project Squirrel. Results shown in Table 7 were derived from these curves. Figures on other target projects are shown in the Appendix as Figure A.2.

this problem is that, with an accurate estimation of the number of undiscovered SATDs, human experts are easier to make decisions on whether to spend more time looking for the “hard to find” SATDs or to stop at that point. To assess the accuracy of the estimation, we plot out the recall-cost curves with the following estimation:

$$Estimation = \frac{\text{estimated number of SATDs}}{|\{\text{SATDs}\}|} \quad (8)$$

As we can see from Figure 5, **Hard** tends to overestimate in the beginning but converges to the actual value after around 10% cost and it does help to determine when has 90% recall been reached. These recall-cost and estimation-cost curves on other target projects can be found in Figure A.2 from the Appendix. Table 7 shows the final recall and cost when **Hard** stops at 90% recall through estimations. Overall, **Hard** could stop close to the target 90% recall with the estimations.

5.3 RQ3: Overall how does Jitterbug perform?

In this research question, we evaluate the overall performance of **Jitterbug** in three aspects: (1) APFD measures its overall efficiency without stopping rules; (2) Precision, Recall, F1 score, and Cost measures its performance when finding 90% “hard

TABLE 7: Performance of **Hard** aiming to find 90% of the “hard to find” SATDs with the estimator.

Targeting 90% Recall	Squirrel	JMeter	EMF	Apache Ant	ArgoUML	Hibernate	JEdit	JFreeChart	Columba	JRuby	Median	IQR
Recall	0.92	0.93	0.99	0.87	0.88	0.81	0.94	0.99	1.00	0.92	0.92	0.08
Cost	0.18	0.35	0.28	0.24	0.12	0.25	0.17	0.19	0.23	0.27	0.24	0.08

TABLE 8: APFD (higher the better) results for different treatments on finding all the SATDs. Medians and IQRs (delta between 75th and 25th percentile, lower the better) are calculated for easy comparisons. The proposed treatment **Jitterbug=Easy+Hard**. A threshold of Cohen’s small effect size (0.01) is applied to determine which treatment performs best in each target project and color them in gray. The column **#Best** shows the number of projects each treatment performs best in.

Treatment	Squirrel	JMeter	EMF	Apache Ant	ArgoUML	Hibernate	JEdit	JFreeChart	Columba	JRuby	Median	IQR	#Best
Jitterbug	0.97	0.97	0.95	0.93	1.00	0.96	0.94	0.97	1.00	0.99	0.97	0.03	10
Easy+RF	0.93	0.95	0.91	0.87	0.99	0.95	0.87	0.88	0.99	0.99	0.94	0.09	4
Hard	0.95	0.95	0.95	0.91	0.91	0.89	0.95	0.90	0.98	0.92	0.93	0.04	2
MAT+RF	0.91	0.97	0.88	0.92	0.99	0.96	0.86	0.87	0.98	0.98	0.94	0.09	4
TM	0.83	0.94	0.90	0.88	0.91	0.88	0.92	0.84	0.98	0.91	0.90	0.04	0
RF	0.91	0.93	0.84	0.88	0.90	0.88	0.88	0.84	0.97	0.92	0.89	0.04	0

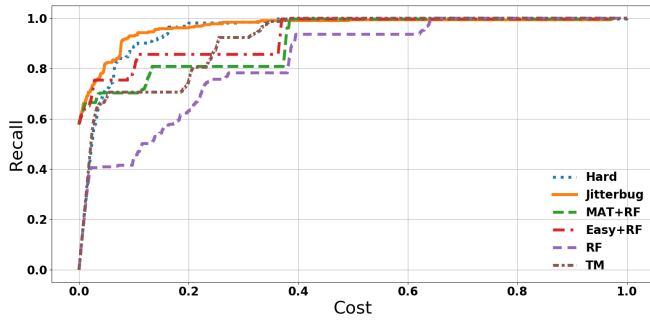


Fig. 6: Recall-cost curves for finding all SATDs with different treatments on target project Squirrel. APFD results in Table 8 were calculated as the area under these curves. Figures on other target projects are shown in the Appendix as Figure A.3.

to find” SATDs based on estimation; (3) runtime measures the additional computation cost of **Jitterbug** besides the human effort cost.

5.3.1 RQ3.1: How does **Jitterbug** perform in terms of APFD?

Table 8 shows the overall APFD scores for finding all the SATDs in the target project. The following treatments are tested in this experiment:

- **Jitterbug**: First apply **Easy** to automatically identify the “easy to find” SATDs with zero human effort, then apply **Hard** to guide humans in identifying the “hard to find” SATDs.
- **Easy+RF**: First apply **Easy** to automatically identify the “easy to find” SATDs with zero human effort, then apply a supervised learner random forest to rank the comments for humans to identify the “hard to find” SATDs.
- **Hard**: Directly apply **Hard** to guide humans in identifying both “easy to find” and “hard to find” SATDs.
- **MAT+RF**: First apply **MAT** to automatically identify the “easy to find” SATDs with zero human effort, then apply a supervised learner random forest to rank the comments for human to identify the “hard to find” SATDs.
- **TM**: Apply **TM** to rank all comments for humans to identify SATDs.

- **RF**: Apply random forest classifier to rank all comments for humans to identify SATDs.

Similarly to **RQ2.2**, we applied the following threshold from Cohen’s small effect size to determine the best treatments in each target project:

$$Small_{overall} = 0.2 \cdot StdDev(\text{All APFD results}) = 0.01. \quad (9)$$

From these results, we observed

- **Jitterbug** outperforms the state-of-the-art solutions **TM**, **RF**, and **MAT+RF**.
- **Jitterbug (Easy+Hard)** outperforms **Easy+RF**, which means the **Hard** strategy outperforms **RF** in Step 2 and this contributes to the overall performance.
- **Jitterbug (Easy+Hard)** outperforms **Hard**, which means applying **Easy** in Step 1 does contribute to the overall performance.

For a more intuitive comparison, Figure 6 shows the recall-cost curves of the compared treatments on target project Squirrel. The APFD results in Table 8 were calculated as the area under these curves. As we can see, **Jitterbug** has the highest APFD score of 0.97. Also, in Figure 6 it almost always reaches the same recall at a lower cost than other treatments. Recall-cost curves on other target projects can be found in Figure A.3 from the Appendix.

In conclusion, **Jitterbug** outperforms the state-of-the-art solutions (**TM**, **RF**, **MAT+RF**) in identifying SATDs in terms of APFD, and both of the two components **Easy** and **Hard** contribute to its good performance.

5.3.2 RQ3.2: How does **Jitterbug** perform overall when targeting at finding 90% of the “hard to find” SATDs?

Table 9 shows the overall performance scores for finding all the SATDs in the target project. To compare against the latest state-of-the-art deep convolutional neural network-based approach [3], precision, recall, F1 score, and cost are applied to evaluate each treatment on the original (uncorrected) dataset. The following treatments are tested in this experiment:

- **Easy**: Apply **Easy** to automatically identify the “easy to find” SATDs with zero human effort, then stop.
- **Jitterbug**: First apply **Easy** to automatically identify the “easy to find” SATDs with zero human effort, then apply **Hard** to guide humans until 90% of the “hard to find” SATDs are identified (according to estimation).

TABLE 9: Comparison between **Easy**, **Jitterbug**, and the best performing state-of-the-art supervised learning approach— **CNN** [3] on the original datasets in terms of precision, recall, F1 score, and cost. Here, **Jitterbug=Easy+Hard** targets at finding 90% of the “hard to find” SATDs with its estimator and its **Easy** part costs zero human effort.

Metrics	Treatment	Squirrel	JMeter	EMF	Apache Ant	ArgoUML	Hibernate	JUnit	JFreeChart	Columba	JRuby	Median	IQR
Precision	Easy	0.85	0.87	0.69	0.89	0.85	0.94	0.95	0.72	0.91	0.93	0.88	0.11
	Jitterbug	0.21	0.13	0.10	0.12	0.62	0.48	0.16	0.23	0.33	0.65	0.22	0.39
	CNN [3]	0.79	0.87	0.79	0.58	0.82	0.93	0.77	0.69	0.83	0.81	0.8	0.09
Recall	Easy	0.54	0.75	0.33	0.24	0.88	0.74	0.21	0.47	0.87	0.52	0.53	0.47
	Jitterbug	0.97	0.98	0.96	0.93	0.99	0.95	0.96	0.98	0.99	0.96	0.97	0.02
	CNN [3]	0.69	0.79	0.59	0.76	0.95	0.74	0.49	0.80	0.88	0.93	0.77	0.22
F1	Easy	0.66	0.80	0.44	0.38	0.87	0.83	0.35	0.57	0.89	0.67	0.66	0.41
	Jitterbug	0.35	0.23	0.19	0.21	0.76	0.64	0.27	0.37	0.49	0.77	0.36	0.45
	CNN [3]	0.74	0.83	0.68	0.66	0.88	0.83	0.60	0.74	0.85	0.86	0.78	0.18
Cost	Easy	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Jitterbug	0.16	0.31	0.21	0.24	0.08	0.19	0.15	0.17	0.06	0.12	0.16	0.11
	CNN [3]	0.03	0.04	0.02	0.04	0.17	0.13	0.02	0.06	0.03	0.15	0.04	0.1

- **CNN**: Apply a convolutional neural network (on word2vec features and with hyperparameter tuning) to classify each comment into SATD or non-SATD [3]. Due to the difficulty of reproducing a deep learning solution, we used the same precision, recall, F1 scores reported in Ren et al. [3], and the cost metric for CNN is calculated as

$$Cost = \frac{|SATDs| \times recall}{|comments| \times precision}$$

From the results in Table 9, we observed

- **Precision**: **Easy** always achieves the highest precision except for the **EMF** project. We know from Table 4 that once the labels are corrected, the precision of **Easy** on the **EMF** project will be close to 100%. Therefore, **Easy** can reach higher precision than **CNN** on every project.
- **Recall**: **Jitterbug** achieves the highest recall on every project. Also, since the target of **Jitterbug** is to stop at finding 90% of the “hard to find” SATDs, its final recalls should all be higher than 90%, which is consistent with the results shown in Table 4.
- **F1**: **CNN** always achieves the highest F1 score except for the **Columba** project. This suggests that the cutoff point of **CNN** is more balanced (between precision and recall) than **Easy** and **Jitterbug**.
- **Cost**: While **Easy** always cost zero human effort, **CNN** costs less human effort than **Jitterbug** in 8 out of 10 projects due to its higher precision than **Jitterbug**.
- **Overall**: there is no clear winner except for three projects: (1) on **ArgoUML** and **JRuby**, **Jitterbug** achieves both higher recall and a lower cost than **CNN**; (2) on **Hibernate**, **Easy** achieves the same recall at a lower cost than **CNN**.

In conclusion, there is no clear win of **Jitterbug** over **CNN** except on the **ArgoUML** and **JRuby** projects where **Jitterbug** achieves higher recall at a lower cost than **CNN**. On the rest eight projects, **Jitterbug** always achieves higher recall but also at a higher cost than **CNN**.

Besides the fact that **Jitterbug** outperforms **CNN** on two out of ten projects. The advantage of **Jitterbug** is that (1) it separates the “easy to find” SATDs from the “hard to find” ones so that the “easy to find” SATDs can be automatically identified with zero human cost; and (2) it can guarantee a high level of recall on the “hard to find” SATDs with accurate estimation.

On the other hand, **CNN** also shows its strong prediction capability with its complex model and hyperparameter tuning. It is promising that replacing the random forest classifier in **Hard**

with **CNN** can further improves the performance of **Jitterbug** in the future.

Therefore, we believe that **CNN** has the potential to further improve **Jitterbug** in the future but **Jitterbug** is a better solution in terms of framework. Imagine using **CNN** for SATD identification on a new project, the user will be told that on average 80% of the predicted comments will contain SATDs. If the user check all the predicted comments, they will be checking on average 4% of the comments and finding 78% of the SATDs and this is the end of it. On the other hand, when using **Jitterbug**, the user will first be told that the selected comments by **Easy** are 100% related to SATDs. On average, those comments cover 53% of the total SATDs. If the user wants to find more SATDs, **Hard** will guide the user to check the comments most likely to be SATDs among the remaining “hard to find” ones. During this process, a recall will be estimated to inform the user what percentage of the “hard to find” SATDs have been identified. Thus **Jitterbug** can guarantee to reach the user-specified recall.

5.3.3 RQ3.3: How much computation cost does **Jitterbug** add to the process besides the human effort cost?

The runtime for **Jitterbug** is split into two parts:

- **Easy** takes 12 seconds to train on 62,275 comments and to detect the “easy to find” SATDs on one target project.
- **Hard** takes on average 20.5 seconds to train the random forest model and 1.5 seconds to estimate recall in each iteration.

As a result, the total runtime for **Jitterbug** would be $12 + 22 \times N$ seconds where N is the number of iterations (10 comments per iteration). For projects that require many iterations of **Hard**, the total runtime for **Jitterbug** can be hours. However, given the fact that human needs more than 22 seconds to classify 10 comments in each iteration, in practice, the training time of **Hard** can hide behind the human classification time. That is to say, we can train the model once, provide the first 20 comments to human, then update the model for every 10 labeled comments and utilize the updated model to provide the next 10 comments for human classification³. In this way, the additional computation cost is 34 seconds for **Jitterbug** which is similar to training a traditional supervised learning model (**TM**, **RF**) and is much lower than training a deep learning model (3,548 seconds for training a **CNN** model in Ren et al. [3]).

3. Note that the current design of the **Jitterbug** does not implement this. We plan to develop a more interactive and user-friendly tool of **Jitterbug** utilizing this query strategy in our future work.

In summary:

Jitterbug outperforms the state-of-the-art SATD identification solutions by reaching higher recall at a lower cost and negligible additional computation time (34 seconds). This is attributed to two factors of **Jitterbug**: (1) it first identifies the “easy to find” SATDs with close to 100% precision, thus 20 to 90% of the SATDs can be found with zero human effort; (2) for the remaining “hard to find” SATDs, it utilizes the human classification results to update its prediction model incrementally and thus can make better guidance to which comments are more likely to be SATDs.

6 APPLY JITTERBUG TO A NEW PROJECT

To test the generalizability of **Jitterbug** while also demonstrating how researchers and developers can use **Jitterbug**, we applied **Jitterbug** to identify SATDs from a new, unlabeled project Apache httpd-2.4.6⁴.

Data collection: we applied srcML⁵ and extracted 17,208 code comments from the Apache httpd-2.4.6 project, similar to what Potdar and Shiha [22] did.

Easy: the **Easy** algorithm trained on the 10 labeled datasets (the four patterns of “*fixme*, *todo*, *hack*, *workaround*”) was applied to extract the “easy to find” SATDs from the 17,208 code comments. As a result, 148 comments were extracted as the “easy to find” SATDs.

Validation of Easy: given the close to 100% precision of **Easy**, the 148 comments should be automatically treated as SATDs without human verification. However, to test the generalizability of **Easy**, we validated these 148 comments by manually inspecting and classifying each one. We found 4 false-positives from these 148 comments, as listed in Table 10. Therefore, the precision of **Easy** on the Apache httpd-2.4.6 project is 144/148=97%. The full validation results of all 148 comments are available on the GitHub repository⁶.

TABLE 10: False-Positives of Easy on Apache Httpd

Comment Text	Easy	Human
/ See TODO in ap_queue_info_set_idle() /	yes	no
/ See TODO in ap_queue_info_set_idle() /	yes	no
/ basedir is either "", or "%2f" for the "squid %2f hack" /	yes	no
/ Add a link to the root directory (if %2f hack was used) /	yes	no

Hard: the remaining 17,060 comments contain only the “hard to find” SATDs. To identify such “hard to find” SATDs, the **Hard** model trained on the 10 labeled datasets was applied to sample 10 most informative comments from the 17,060 comments. Then the first author (acting as a user) manually inspected and labeled the sampled 10 comments. Next, the **Hard** model was re-trained with the newly added 10 labeled data. These processes should iterate until the number of “hard to find” SATDs identified meets the desired recall (based on the estimation of the total number of “hard to find” SATDs in the 17,060 comments). However, with limited human effort available, we only inspected and labeled 100

comments with the help of **Hard**, interactively. Among these 100 inspected comments, 87 were found to be SATDs. Therefore, the precision@100 for **Hard** on Apache httpd-2.4.6 is 87%. The full inspection results are available at the GitHub repository⁷.

Summary: on Apache httpd-2.4.6, the precision of **Easy** is 97% and the precision@100 for **Hard** is 87%. These results suggest that **Jitterbug** can be successfully applied to unlabeled software projects. As an existing work, Potdar and Shiha [22] applied their keyword-based method to identify SATDs on Apache httpd-2.4.6. They found 112 SATDs in total. The **Jitterbug** result is better than Potdar and Shiha [22] since by manually inspecting 100 comments, **Jitterbug** identified 144+87 = 231 SATDs, of which the 144 “easy to find” SATDs did not require any human effort. More details on how to apply **Jitterbug** to extract SATDs from an unlabeled project are available on our GitHub repository⁸.

7 THREATS TO VALIDITY

There are several validity threats [52] to the design of this study. Any conclusion made from this work must be considered with the following issues in mind:

Conclusion validity focuses on the significance of the treatment. To enhance conclusion validity, we ran experiments on 10 different target projects and found that our proposed method always performed better than the state-of-the-art approaches.

Internal validity focuses on how sure we can be that the treatment caused the outcome. To enhance internal validity, we heavily constrained our experiments to the same dataset, with the same settings, except for the treatments being compared.

Construct validity focuses on the relation between the theory behind the experiment and the observation. To enhance construct validity, we compared solutions with and without our strategies in Table 8 and showed that both components (**Easy** and **Hard**) improve the overall performance. However, we only showed that with our setting of featurization and default parameters of each learner, random forest learner is the best choice. What we have not shown is that whether the performance can get even better by tuning the parameters or using a different set of features. We plan to explore these in our future work.

External validity concerns how widely our conclusions can be applied. In order to test the generalizability of our approach, we always kept a project as the holdout test set and never used any information from it in training. In addition, we have applied **Jitterbug** to identify SATDs from an unlabeled software project Apache httpd-2.4.6 with a real human inspecting the comments. The results shown in §6 demonstrated the success of **Jitterbug** in this real-world application.

8 CONCLUSION AND FUTURE WORK

Identifying self-admitted technical debts (SATDs) from source code comments is important to maintain a healthy software project. Current solutions cannot automate this process due to the lack of precision of machine learning models in predicting the SATDs. To reduce the human effort required in identifying SATDs, this paper first showed that there are two types of SATDs— (1) the “easy to find” SATDs that can be automatically identified with close to 100% precision; and (2) the “hard to find” SATDs that

4. <https://archive.apache.org/dist/httpd/httpd-2.4.6.tar.gz>

5. <https://www.srcml.org/>

6. https://github.com/ai-se/Jitterbug/blob/master/httpd/httpd_easy_validated.csv

7. https://github.com/ai-se/Jitterbug/blob/master/httpd/httpd_rest_coded.csv

8. <https://github.com/ai-se/Jitterbug/blob/master/README.md>

only human experts can make the final decisions on. Then a half-automated two-step approach was proposed— **Step 1**: apply a novel pattern recognition technique to learn and utilize strong patterns of the “easy to find” SATDs to identify them automatically; **Step 2**: (a) train/update a continuous learning model incrementally on both historically labeled data and new human decisions, and (b) guide the human experts to screen the comments that most likely contain the “hard to find” SATDs according to the model’s prediction, iterate (a) and (b) until a target recall has been reached according to the model’s estimation. Based on simulation results on ten software projects, we conclude that

- **Step 1** solution **Easy** can find 20-90% of the SATDs with close to 100% precision automatically.
- **Step 2** solution **Hard** outperforms the state-of-the-art methods in finding the remaining “hard to find” SATDs (**Hard** finds more SATDs with less human effort).
- **Step 2** solution **Hard** can also provide an accurate estimation of the number of “hard to find” SATDs undiscovered, thus offers a practical way to stop at the target recall.
- Overall, the proposed two-step solution **Jitterbug** is most efficient in identifying SATDs.

That said, **Jitterbug** still suffers from the validity threats discussed in §7. To further reduce those threats and to move forward with this research, we propose the following future work:

- 1) Apply hyper-parameter tuning on data preprocessing and model configuration to see if our current conclusions still hold and whether tuning can further improve the performance.
- 2) Prototype **Jitterbug** as a tool to make it more user-friendly.
- 3) Explore more complex patterns (other than just single word patterns **Easy** has explored) in Step 1.
- 4) Explore more advanced feature engineering in Step 2 for finding the “hard to find” SATDs. E.g. explore N-gram patterns [39] and word embeddings with deep neural networks [40].
- 5) Explore whether replacing the random forest model in **Jitterbug** with a deep learning model (CNN [3]) will further improve its performance.
- 6) Extend the work to other types of technical debts and compare it with other state-of-the-art methods which continue to appear.

One important message this paper tries to convey is that— do not waste effort on finding the “easy to find” SATDs, future research on identifying SATDs should mostly focus on the “hard to find” SATDs.

APPENDIX A

This appendix shows the recall-cost curves on every target project in Figure A.1, Figure A.2, and Figure A.3.

ACKNOWLEDGEMENTS

This research was partially funded by a National Science Foundation Grant #1703487. The authors would like to thank Maldonado and Shihab for making their SATD data available to the public, which makes this research possible.

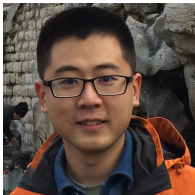
REFERENCES

- [1] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” in *MSR*, 2016.
- [2] S. Wehaibi, E. Shihab, and L. Guerrouj, “Examining the impact of self-admitted technical debt on software quality,” in *SANER*, 2016.
- [3] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, “Neural network-based detection of self-admitted technical debt: From performance to explainability,” *TOSEM*, 2019.
- [4] F. Zampetti, A. Serebrenik, and M. Di Penta, “Automatically learning patterns for self-admitted technical debt removal,” in *SANER*, 2019.
- [5] E. d. S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical debt,” in *MTD*, 2015.
- [6] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, and B. Xu, “Mat: A simple yet strong baseline for identifying self-admitted technical debt,” 2019.
- [7] W. Cunningham, “The wycash portfolio management system,” *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.
- [8] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, and C. Siebra, “Tracking technical debt—an exploratory case study,” in *ICSME*, 2011.
- [9] A. Nugroho, J. Visser, and T. Kuipers, “An empirical model of technical debt and interest,” in *MTD*, 2011.
- [10] E. Lim, N. Taksande, and C. Seaman, “A balancing act: What software practitioners have to say about technical debt,” *IEEE Software*, 2012.
- [11] A. Martini and J. Bosch, “The danger of architectural technical debt: Contagious debt and vicious circles,” in *12th ICSA*, 2015.
- [12] I. Ozkaya, R. L. Nord, and P. Kruchten, “Technical debt: From metaphor to theory and practice,” *Software*, 2012.
- [13] R. Marinescu, G. Ganea, and I. Verebi, “Incode: Continuous quality assessment and improvement,” in *CSMR*, 2010.
- [14] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *ICSME*, 2004.
- [15] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, “A case study on effectively identifying technical debt,” in *EASE*, 2013.
- [16] R. Marinescu, “Assessing technical debt by identifying design flaws in software systems,” *IBM Journal of Research and Development*, 2012.
- [17] F. A. Fontana, V. Ferme, and S. Spinelli, “Investigating the impact of code smells debt on quality code evaluation,” in *MTD*, 2012.
- [18] N. Tsantalis and A. Chatzigeorgiou, “Identification of extract method refactoring opportunities for the decomposition of methods,” *JSS*, 2011.
- [19] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, “Assessing the refactorability of software clones,” *TSE*, 2015.
- [20] J. Graf, “Speeding up context-, object-and field-sensitive sdg generation,” in *SCAM*, 2010.
- [21] K. Ali and O. Lhoták, “Application-only call graph construction,” in *ECOOP*, 2012.
- [22] A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” in *ICSME*, 2014.
- [23] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, and R. O. Spínola, “A contextualized vocabulary model for identifying technical debt on code comments,” in *MTD*, 2015.
- [24] M. A. de Freitas Farias, J. A. Santos, M. Kalinowski, M. Mendonça, and R. O. Spínola, “Investigating the identification of technical debt through code comment analysis,” in *ICEIS*, 2016.
- [25] M. A. de Freitas Farias, M. G. de Mendonça Neto, M. Kalinowski, and R. O. Spínola, “Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary,” *JST*, 2020.
- [26] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, “An empirical study on the removal of self-admitted technical debt,” in *ICSME*, 2017.
- [27] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, “Automating change-level self-admitted technical debt determination,” *TSE*, 2018.
- [28] J. Flisar and V. Podgorelec, “Enhanced feature selection using word embeddings for self-admitted technical debt identification,” in *SEAA*, 2018.
- [29] G. Sierra, E. Shihab, and Y. Kamei, “A survey of self-admitted technical debt,” *JSS*, 2019.
- [30] E. d. S. Maldonado, E. Shihab, and N. Tsantalis, “Using natural language processing to automatically detect self-admitted technical debt,” *TSE*, 2017.
- [31] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Satd detector: a text-mining-based self-admitted technical debt detection tool,” in *ICSE*, 2018.
- [32] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” *EMSE*, 2018.
- [33] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, “/* icomment: Bugs or bad comments?*,” in *ACM SIGOPS Operating Systems Review*. ACM, 2007.
- [34] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, “@ tcomment: Testing javadoc comments to detect comment-code inconsistencies,” in *ICST*, 2012.
- [35] N. Khamis, R. Witte, and J. Rilling, “Automatic quality assessment of source code comments: the javadocminer,” in *NLDB*, 2010.

- [36] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *ICPC*, 2013.
- [37] H. Malik, I. Chowdhury, H. Tsou, Z. M. Jiang, and A. E. Hassan, "Understanding the rationale for updating a function's comment," in *ICSME*, 2008.
- [38] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *WCRE*, 2007.
- [39] S. Wattanakriengkrai, N. Srisermphoak, S. Sintoplertchaikul, M. Choetkiertikul, C. Raghitwetsagul, T. Sunetnanta, H. Hata, and K. Matsumoto, "Automatic classifying self-admitted technical debt using n-gram idf," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 316–322.
- [40] J. Flisar and V. Podgorelec, "Identification of self-admitted technical debt using enhanced feature selection based on word embedding," *IEEE Access*, vol. 7, pp. 106 475–106 494, 2019.
- [41] C. Siva, "Machine learning and pattern recognition," <https://dzone.com/articles/machine-learning-and-pattern-recognition>, 2018.
- [42] R. E. Wright, "Logistic regression." 1995.
- [43] L. Rokach and O. Z. Maimon, *Data mining with decision trees: theory and applications*. World scientific, 2008.
- [44] T. K. Ho, "Random decision forests," in *ICDAR*, 1995.
- [45] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 2001.
- [46] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, 1999.
- [47] Z. Yu and T. Menzies, "Fast2: An intelligent assistant for finding relevant papers," *ESA*, 2019.
- [48] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *ICSE*, 2011.
- [49] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *TechDebt*, 2014.
- [50] J. Cohen, "Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit." *Psychological Bulletin*, 1968.
- [51] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *TSE*, 2001.
- [52] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research-an initial survey." in *SEKE*, 2010.



Tim Menzies (IEEE Fellow) is a Professor in CS at NcState His research interests include software engineering (SE), data mining, artificial intelligence, search-based SE, and open access science. <http://menzies.us>



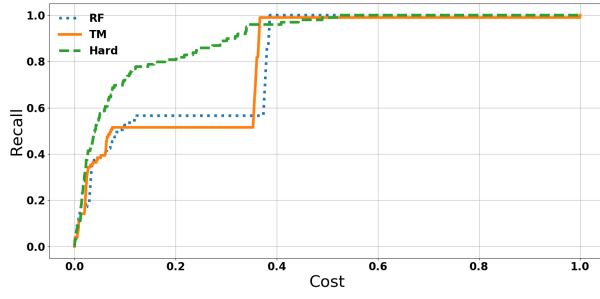
Zhe Yu (Ph.D. NC State University, 2020) is an assistant professor in the Department of Software Engineering at Rochester Institute of Technology, where he teaches data mining and software engineering. His research explores collaborations of human and machine learning algorithms that leads to better performance and higher efficiency. For more information, please visit <http://zhe-yu.github.io/>.



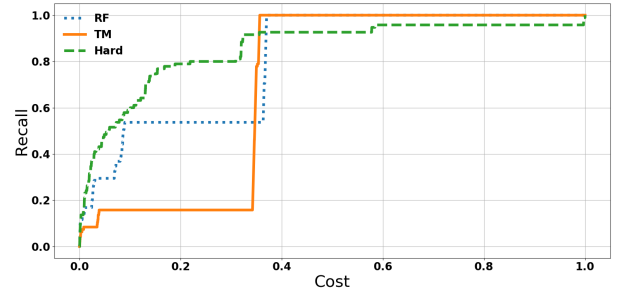
Fahmid Morshed Fahid is a Ph.D. student in the IntelliMedia Lab at North Carolina State University, under the supervision of Dr. James Lester. His research interest is Intelligent Learning Environment. Before joining NC State, He worked as a Software Engineer in Research and Development department of Reve System Ltd. and received his Bachelors Degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET) . <http://fahmidmorshed.github.io/>



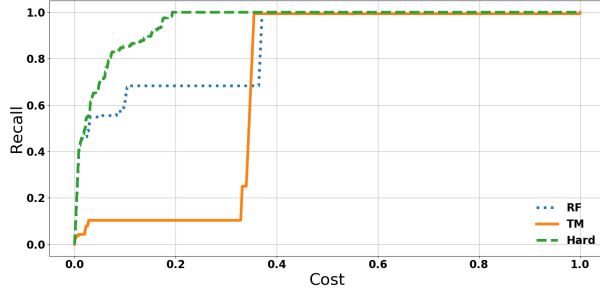
Huy Tu is a Ph.D. student in the department of Computer Science at North Carolina State University. He explores machine learning models that support and leverage from the human experience to solve real-world problems in software engineering. For more information, please visit <http://kentuu.us>.



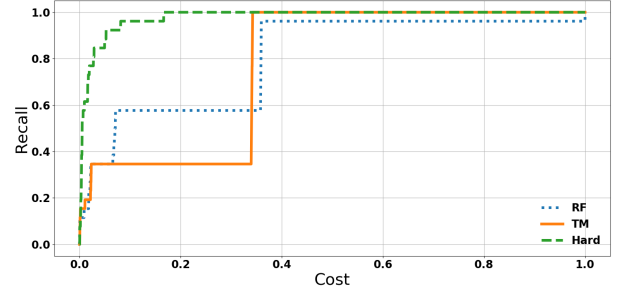
(a) Apache Ant



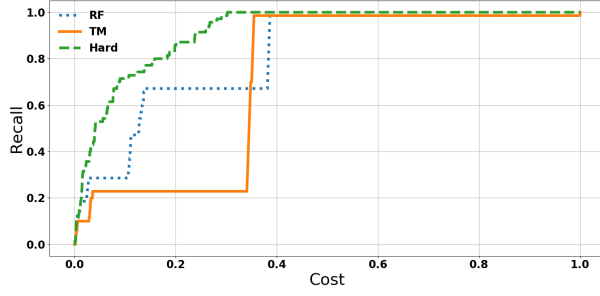
(b) JMeter



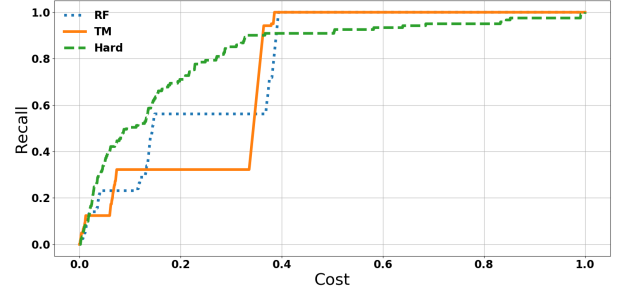
(c) ArgoUML



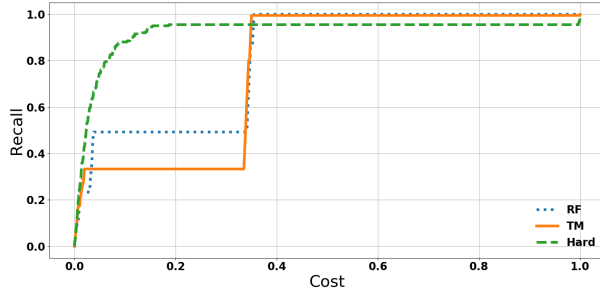
(d) Columba



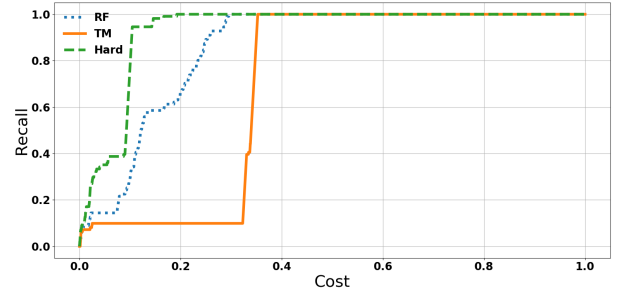
(e) EMF



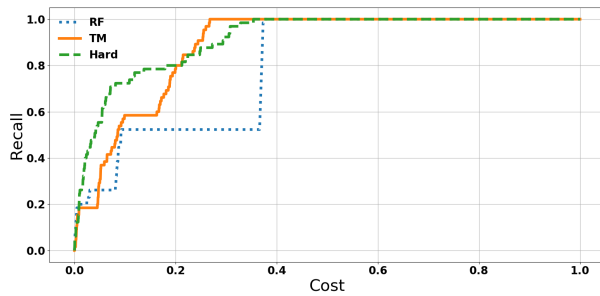
(f) Hibernate



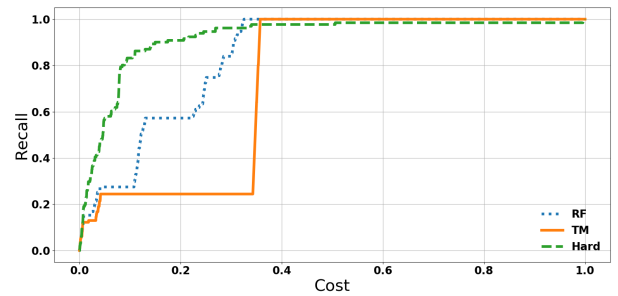
(g) JEdit



(h) JFreeChart

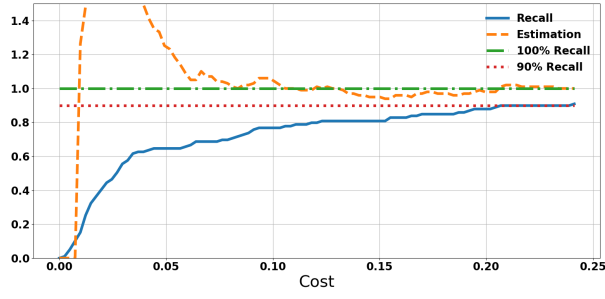


(i) JRuby

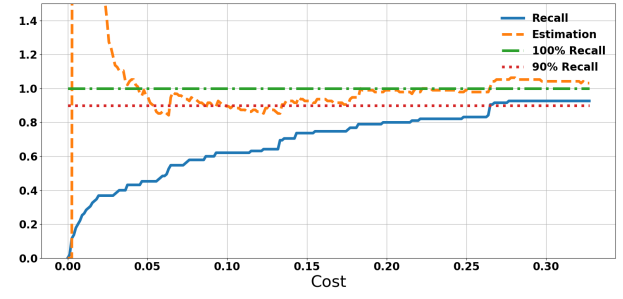


(j) Squirrel

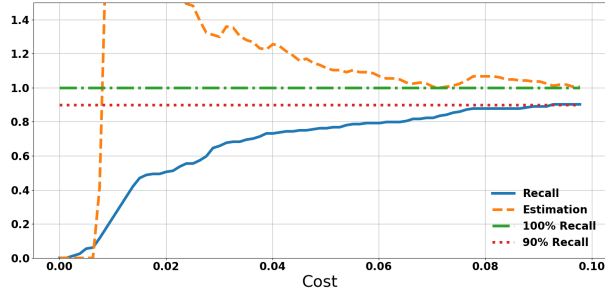
Fig. A.1: Recall-cost curves for finding “hard to find” SATDs on every target project.



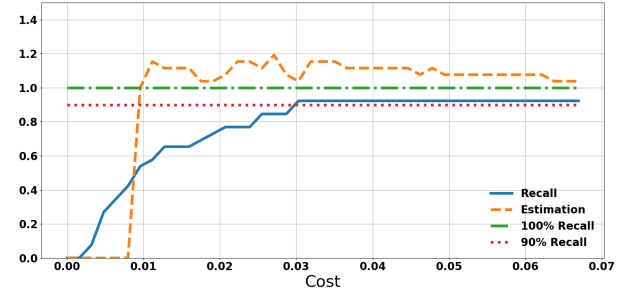
(a) Apache Ant



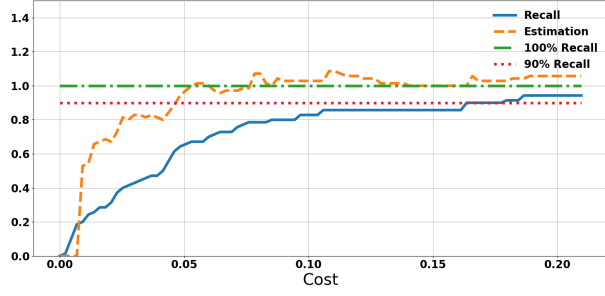
(b) JMeter



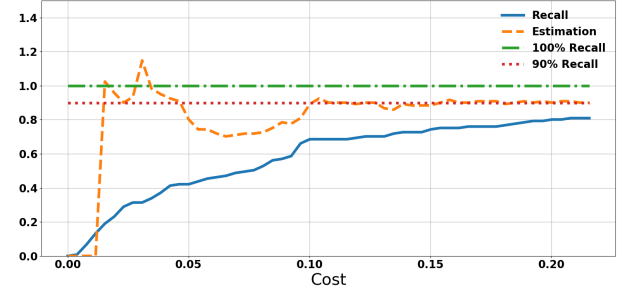
(c) ArgoUML



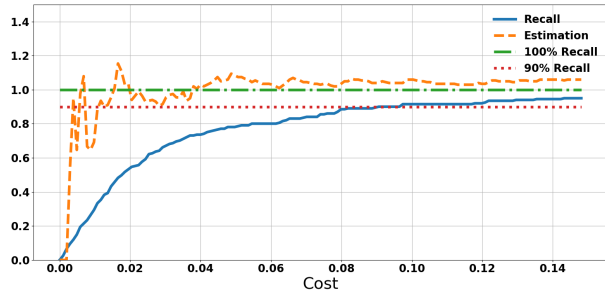
(d) Columba



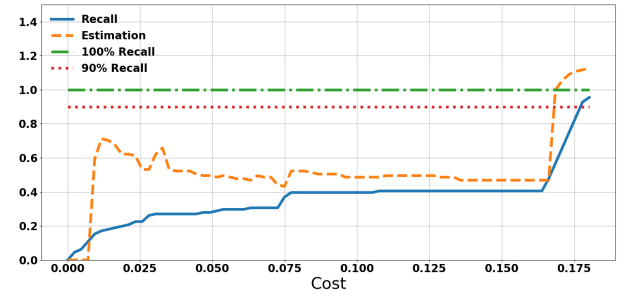
(e) EMF



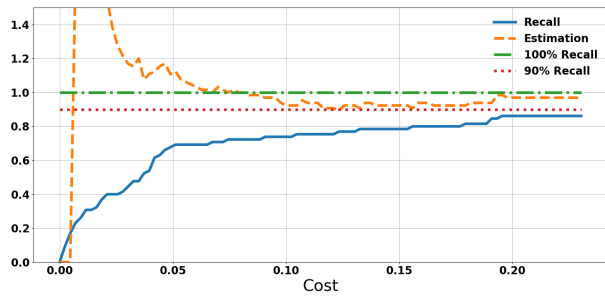
(f) Hibernate



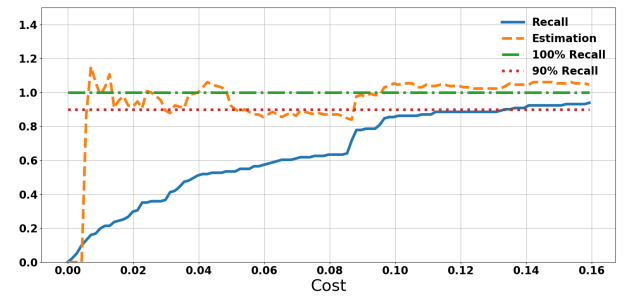
(g) JEdit



(h) JFreeChart

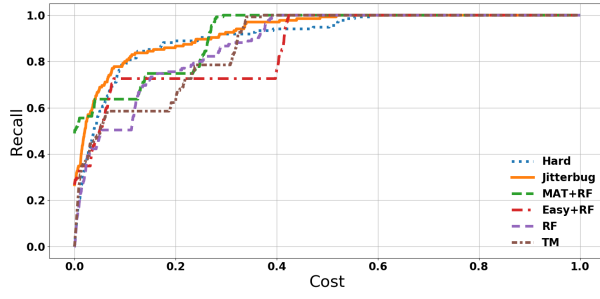


(i) JRuby

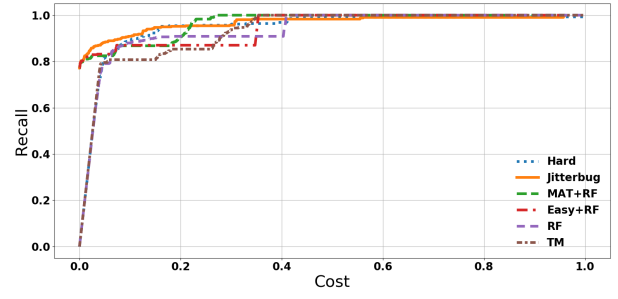


(j) Squirrel

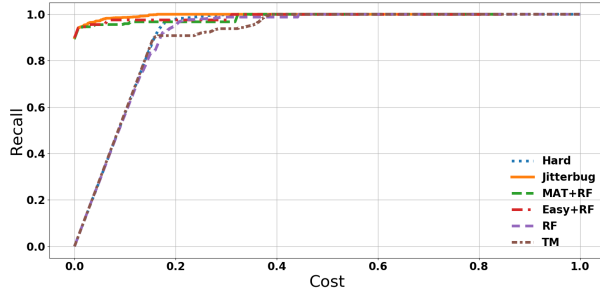
Fig. A.2: Recall-cost and estimation-cost curves for finding 90% of the “hard to find” SATDs with **Hard** on every target project.



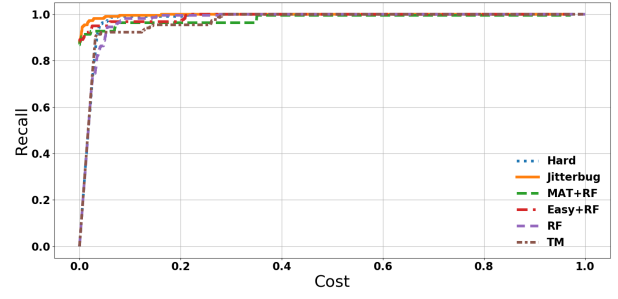
(a) Apache Ant



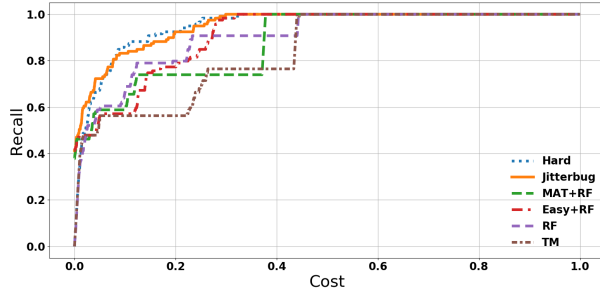
(b) JMeter



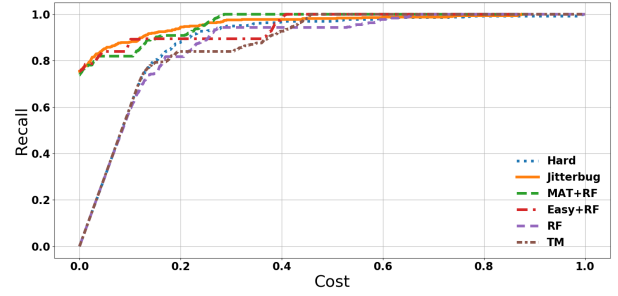
(c) ArgoUML



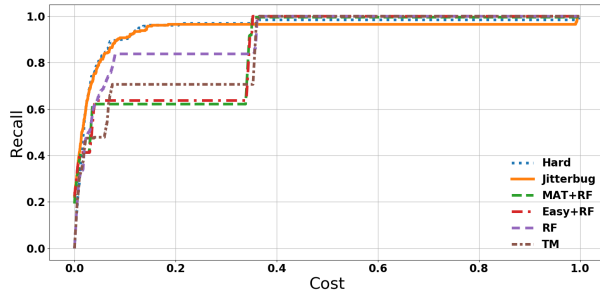
(d) Columba



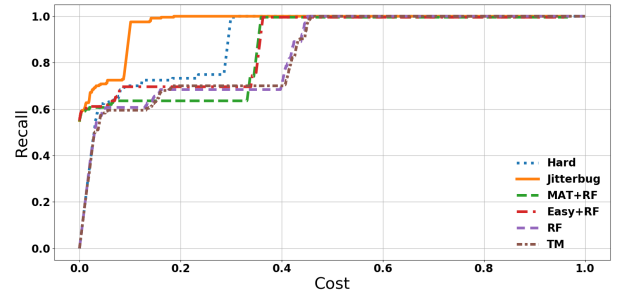
(e) EMF



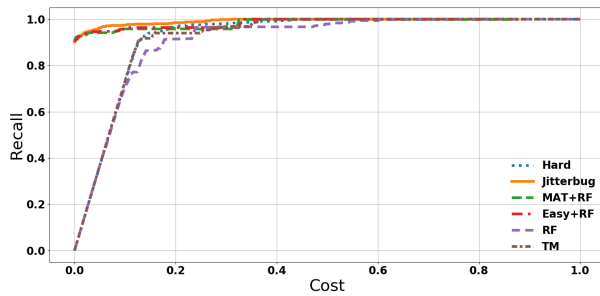
(f) Hibernate



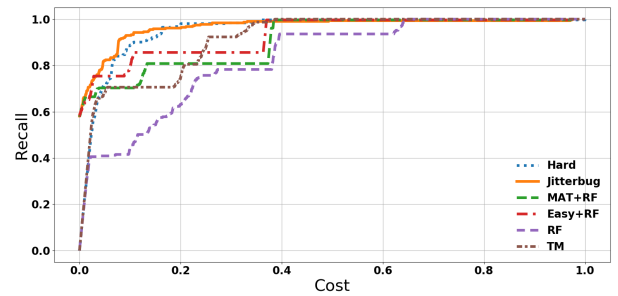
(g) JEdit



(h) JFreeChart



(i) JRuby



(j) Squirrel

Fig. A.3: Recall-cost curves for finding all SATDs on every target project.