SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool

Zhongxin Liu*, Qiao Huang*, Xin Xia†, Emad Shihab§, David Lo‡, and Shanping Li*
*College of Computer Science and Technology, Zhejiang University, China
†Faculty of Information Technology, Monash University, Australia
§Department of Computer Science and Software Engineering, Concordia University, Canada
‡School of Information Systems, Singapore Management University, Singapore

ABSTRACT

In software projects, technical debt metaphor is used to describe the situation where developers and managers have to accept compromises in long-term software quality to achieve short-term goals. There are many types of technical debt, and self-admitted technical debt (SATD) was proposed recently to consider debt that is introduced intentionally (e.g., through temporary fix) and admitted by developers themselves. Previous work on SATD has shown that SATD can be successfully detected using source code comments. However, most current state-of-the-art approaches identify SATD comments through pattern matching, which achieve high precision but very low recall. That means they may miss many SATD comments and are not practical enough. In this paper, we propose SATD Detector, a tool that is able to (i) automatically detect SATD comments using text mining and (ii) list and manage detected SATD comments in an integrated development environment (IDE). Specifically, this tool first leverages a pre-trained composite classifier to detect SATD comments, and then highlights and marks these SATD comments in the source code editor of an IDE. In addition, SATD Detector provides a view in IDE which collects all detected SATD comments for management.
Demo URL: https://youtu.be/sn4gU2qhGm0
Demo download: https://goo.gl/ZzjBzp

1 INTRODUCTION

In real-world software projects, developers and managers sometimes have to make tradeoffs between long-term code quality and short-term revenue due to various reasons (e.g., cost reduction, market pressure, and tight project schedule). Technical debt, which is introduced by Cunningham [2], is a metaphor used to describe this kind of situation. It has been shown by prior work that technical debt is common, unavoidable and may degrade code quality and increase software complexity in the future [4, 10]. Moreover, technical debt is not always visible, i.e., it may only be known to some specific people but not those who eventually pay for it. Therefore, many studies have been conducted to enable the detection and management of technical debt.

The concept of self-admitted technical debt (SATD) is proposed by Potdar and Shihab [7], which considers the technical debt that is intentionally introduced (e.g., in the form of temporary workaround) and admitted by developers themselves. In particular, SATD is used to describe the situation where developers know that current implementation is not optimal and record this in source code comments. For example, one comment in the open source project "JEdit" mentions that "Need some format checking here". This comment indicates that developers admitted that the corresponding code is defective and requires format checking. A previous study [10] shows that although the percentage of SATD in a project is not high, it can negatively impact the maintenance of a project. Detecting and managing SATD can remind developers and managers about the existence of SATD, help them plan to discharge SATD and hence result in software quality improvement.

Prior work on SATD also shows that SATD can be successfully detected using source code comments [7]. However, most of the previous studies detected SATD by manually classifying comments [7] or using the 62 SATD comment patterns [1, 10] which are manually derived by Potdar and Shihab [7]. Approaches that involve manual classification of comments require much human effort, and thus are not practical for real-world projects. Although pattern-based approaches can achieve high precision, their recall is often very low since they fail to detect SATD comments which do not match any known patterns. This is the case since it is difficult to extract all potential SATD comment patterns. Most recently, Maldonado et al. proposed an approach based on natural language processing (NLP) to automatically identify different types of SATD comments [5]. However, their work only focuses on certain types of SATD (i.e., SATD on design, SATD on requirement or non-SATD), while we care more about whether a comment contains SATD or not, which also includes other types of SATD (i.e., defect debt, documentation debt and test debt). Moreover, no prior work provides practical tools to help developers detect and manage SATD in an IDE.

In this paper, we present SATD Detector, a tool based on our previous work [3]. SATD Detector is able to (i) automatically detect SATD comments in source code through a text-mining-based approach and (ii) list and manage detected SATD comments inside an IDE. SATD Detector is an Eclipse plug-in, which leverages a pre-trained composite classifier to detect SATD comments after a project is imported into Eclipse. Specifically, whenever a developer opens Eclipse, our tool will automatically parse all source code files in the Eclipse workspace, detect and mark the comments which contains SATD. Once some source code files are modified, it will re-parse these files, re-detect SATD comments and update markers in these files immediately. A toolbar button is provided by this plug-in to trigger complete detection (i.e., detection of SATD comments in the *whole* workspace). Moreover, this plug-in also provides an Eclipse view in which all detected SATD comments are listed for

Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

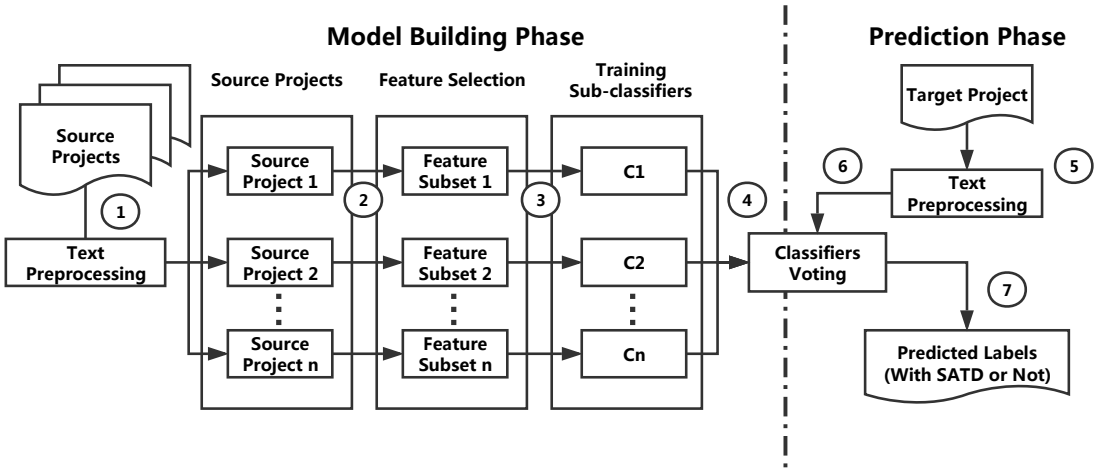

Figure 1: Overall Framework of Our Model

management. With the help of SATD Detector, it would be easy for developers and managers to manage SATD and pay back SATD in time. Also, the tool is easy to deploy and use.

To build and evaluate our tool, we use a manually classified dataset of source code comments from 8 open source projects with 212,413 comments, provided by Maldonado and Shihab [6]. The experimental results show that, on every target project, our tool outperforms Maldonado and Shihab's approach [6] by a substantial margin in terms of F1-score.

The remainder of the paper is organized as follows. In Section 2, we present the text-mining-based model used to detect SATD comments by our tool. The details of SATD Detector, including its workflow, life cycle and user interface, are described in Section 3. Section 4 shows the experimental results of our evaluation. We conclude our work and mention future work in Section 5.

2 APPROACH

2.1 Overall Framework

SATD Detector leverages a pre-trained text mining model to automatically predict whether a comment contains SATD or not. The pre-trained model is the composite classifier proposed in our previous work [3]. Figure 1 presents the overall framework of our model. It contains two phases: a model building phase and a prediction phase. We refer to the projects which are used to build the model as source projects, and the projects we want to detect as target projects. In the model building phase, our approach builds a sub-classifier for each individual source project, using data from the other source projects. In the prediction phase, all sub-classifiers are combined to jointly predict SATD comments in the target project.

Our framework takes as input training comments with known labels from different source projects. It first preprocesses the text descriptions of comments and extracts features (i.e. words) to represent each comment (Step 1). Then, for each source project, feature selection is applied to select features that are useful for classification and remove useless features (Step 2). Next, we use the selected features to train a sub-classifier for each source project (Step 3). Suppose there are n source projects, we end up with n classifiers which are combined to form a composite classifier for prediction (Step 4). For each new comment in the target project, we first preprocess the comment to extract features (Step 5) and input them to the composite classifier (Step 6). Finally, each sub-classifier will predict the label of the comment according to its features, and the label with the largest number of "votes" will be chosen as the final prediction result of the composite classifier (Step 7).

2.2 Model Details

Our model mainly contains four steps: text preprocessing, feature selection, sub-classifiers training and classifiers voting. The following paragraphs elaborate the details of the four steps:

Text Preprocessing: We preprocess the text description of comments to extract features (i.e., words) in 3 steps: tokenization, stop-word removal, and stemming. While tokenizing, we only keep English letters in a token and convert all words to lowercase. As for stop-word removal, since some stop words are useful for classification (e.g., "should"), we manually build a list of stop-words to filter stop-words. Words whose lengths are no more than 2 or no less than 20 are also treated as stop-words. Finally, each token is stemmed (i.e., reduced to its root form) using the well-known Porter stemmer [1].

Feature Selection: After preprocessing and tokenizing the comments, we use the Vector Space Model (VSM) [9] to represent each comment with a word vector. In total, we have a large number of features for each source project (e.g., there are 3,661 features in ArgoUML project). Feature selection is applied to identify a subset of features that are most useful in differentiating different classes (i.e., comments with or without SATD). In this model, we employ Information Gain (IG) [8] to select useful features. Only the features whose feature selection scores are in the top 10% of the ranked list are retained, and the other features are removed.

[1] http://tartarus.org/martin/PorterStemmer

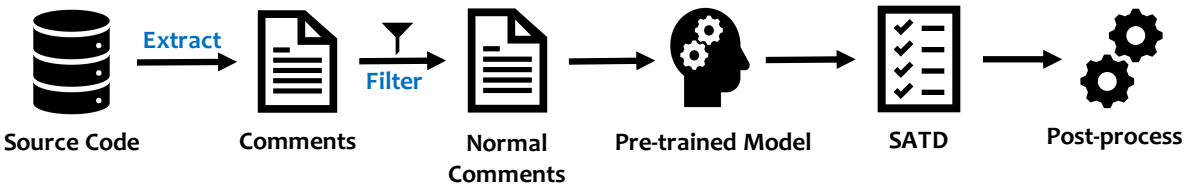

Figure 2: Workflow of SATD Detector

Sub-classifiers Training: In our tool, we train each sub-classifier using Naive Bayes Multinomial (NBM), which is widely used to analyze text data. Note that our tool can also work with other classifiers.

Classifiers Voting: In our model, the composite classifier is built from all the sub-classifiers and it is responsible for predicting the label of a new comment in the target project. The prediction process is just like an election, and the prediction result of each sub-classifier is regarded as a "vote". The comment label which gets the largest number of "votes" will be the final prediction result of the composite classifier.

3 SATD DETECTOR

3.1 Workflow

SATD Detector is an Eclipse plug-in which can automatically detect and manage SATD comments in Eclipse. Figure 2 shows its workflow. First of all, SATD Detector parses the source code files in the workspace, and extracts comments from them. Then, it leverages regular expressions to remove irrelevant comments which mainly include the following two types:

(1) Automatically generated comments with fixed format (i.e., Auto-generated constructor stubs, auto-generated method stubs and auto-generated catch blocks), which are inserted as part of code snippets by Eclipse to generate constructors, methods, and try catch blocks.
(2) Javadoc and license comments which do not contain any task annotation (i.e., "TODO", "FIXME", or "XXX") [8].

Next, the rest of comments are inputted to the pre-trained text mining model described in Section 2, and each comment will be classified by the model. Finally, for comments which are predicted to contain SATD, SATD Detector will post-process them in the source code editor of Eclipse, e.g., highlight them and add markers for them.

3.2 Life Cycle

After installation, SATD Detector will start with Eclipse and then parse source code files in the whole workspace in background. Since in most cases, users only care about SATD in their own projects, our tool ignores the files in the third party libraries. In our current tool implementation, SATD Detector only supports Java projects, and it will not parse non-Java source code files. Once source code files are modified, they will be re-parsed and our tool will detect SATD in these files immediately. The markers created by our tool for SATD comments will not be persisted; hence while users are exiting Eclipse, these markers will be deleted and SATD Detector will then stop.

3.3 User Interface

Figure 3 presents the user interface (UI) of SATD Detector. SATD Detector follows the workflow shown in Figure 2 to detect SATD comments. Once it identifies one comment with SATD, it will highlight this comment (① in Figure 3) and add a marker for this comment (② in Figure 3) in the editor. At the same time, we can check currently detected SATD comments in an Eclipse view (③ in Figure 3). This Eclipse view displays details of each SATD comment, which includes *Description* (i.e., the text description of a comment), *Resource* (i.e., in which file a comment is located), *Path* (i.e., the path of a comment's corresponding file), *Location* (i.e., at which line(s) a comment is located) and *Type* (i.e., the type of a comment's marker). The marker type of SATD comments is set to "Technical Debt" by default. If a user double clicks some SATD comment in the view, Eclipse will open the file in which this comment is located and focus on this comment in the editor (④ in Figure 3). This Eclipse view also provides some basic features for SATD management, e.g., filtering and sorting.

In addition, SATD Detector provides a toolbar button (⑤ in Figure 3), which is used to trigger complete SATD detection. This tool will re-analyze the comments in the whole workspace if a user clicks this button. The time spent by this detection process depends on the size of the target project. In order to improve user experience, detection process always runs in background and the real-time detection progress will be displayed in the Progress view (①② in Figure 4).

4 EVALUATION

To evaluate the performance of SATD Detector, we used the dataset provided by Maldonado and Shihab [6], which involves source code comments from 8 open source projects with 212,413 comments. We compare our tool with 4 kinds of baseline approaches:

(1) Pattern: In this approach, a comment is regarded as SATD comment if and only it matches one of the 62 patterns published by Potdar and Shihab [7].
(2) NBM, SVM and kNN: We build simple classifiers using different text mining techniques (i.e., NBM, SVM and kNN), and classify comments with these classifiers respectively.
(3) BestSub: For each target project, we choose the sub-classifier with best performance as our baseline.
(4) NLP: We follow Maldonado et al.'s work [5] and build a maximum entropy classifier to predict whether a comment contains SATD or not.

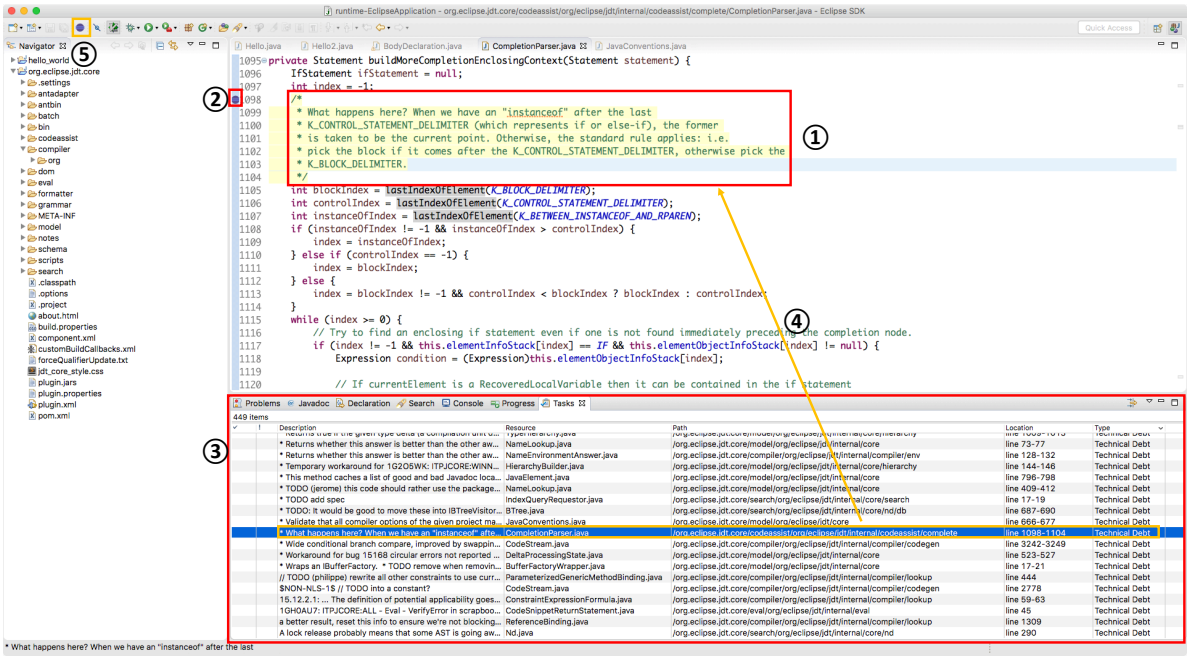

Figure 3: User Interface of SATD Detector

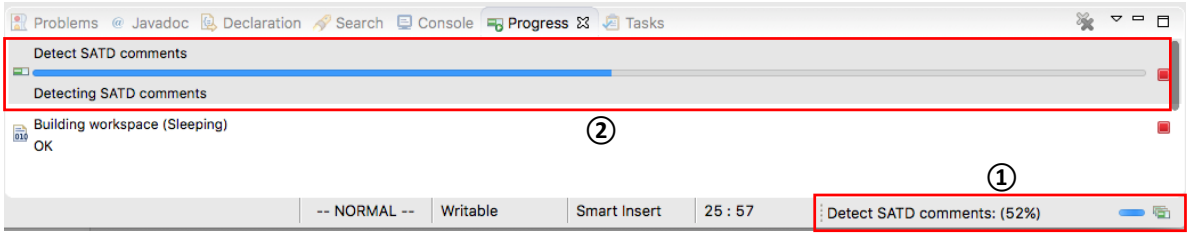

Figure 4: Re-detecting SATD in Background

The experimental results show that on every target project our approach achieves the best performance in terms of F1-score. The F1-score achieved by our approach ranges between 0.518 - 0.841, with an average of 0.737, which is a substantial improvement over the baseline approaches. On average, our approach improves the F1-scores over Pattern-based approach, Naive Bayes Multinomial (NBM) baseline, Support Vector Machine (SVM) baseline, k-Nearest Neighbor (kNN) baseline, BestSub approach and Maldonado et al.'s NLP-based approach by 499.19%, 58.49%, 882.67% 205.81%, 24.70% and 27.95% respectively.

5 CONCLUSION & FUTURE WORK

In this paper, we present SATD Detector, a tool that is able to automatically detect and help developers manage SATD comments in an IDE. This tool is implemented as an Eclipse plug-in with the aim of reminding developers and managers of existing SATD comments and helping them pay for SATD in time. In future, we plan to improve our tool to predict the priority of each SATD comment. We are also interested in providing visualization tools to help developers further analyze SATD comments in different kinds of software projects.

REFERENCES

[1] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *MSR*. ACM, 315–326.
[2] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), 29–30.
[3] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2017. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* (2017), 1–34.
[4] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A balancing act: what software practitioners have to say about technical debt. *IEEE software* 29, 6 (2012), 22–27.
[5] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* (2017).
[6] Everton da S Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE, 9–15.
[7] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *ICSME*. IEEE, 91–100.
[8] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.
[9] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.
[10] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *SANER*, Vol. 1. IEEE, 179–188.