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

Zhongxin Liu
Zhejiang University, China
liu_zx@zju.edu.cn

Qiao Huang
Zhejiang University, China
tkdsheep@zju.edu.cn

Xin Xia
Monash University, Australia
xin.xia@monash.edu

Emad Shihab
Concordia University, Canada
eshihab@encs.concordia.ca

David Lo
Singapore Management University, Singapore
davidlo@smu.edu.sg

Shanping Li
Zhejiang University, China
shan@zju.edu.cn

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 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) highlight, list and manage detected comments in an integrated development environment (IDE). This tool consists of a Java library and an Eclipse plug-in. The Java library is the back-end, which provides command-line interfaces and Java APIs to re-train the text mining model using users' data and automatically detect SATD comments using either the build-in model or a user-specified model. The Eclipse plug-in, which is the front-end, first leverages our pre-trained composite classifier to detect SATD comments, and then highlights and marks these detected comments in the source code editor of Eclipse. In addition, the Eclipse plug-in provides a view in IDE which collects all detected comments for management.
Demo URL: https://youtu.be/sn4gU2qhGm0
Java library download: https://git.io/vNdnY
Eclipse plug-in download: https://goo.gl/ZzjBzp

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Self-admitted technical debt, SATD detection, Eclipse plug-in

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183478

ACM Reference Format:
Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183478

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 [5, 11]. 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 [8], which considers the technical debt that is intentionally introduced (e.g., in the form of temporary workaround) and admitted by developers themselves. In particular, self-admitted technical debt 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 [11] 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 it and hence result in software quality improvement.

Prior work also shows that SATD can be successfully detected using source code comments [8]. However, most of the previous studies detected SATD by manually classifying comments [8] or using the 62 SATD comment patterns [1, 11] which are manually derived by Potdar and Shihab [8]. Approaches that involve manual classification of comments require much human effort, and thus

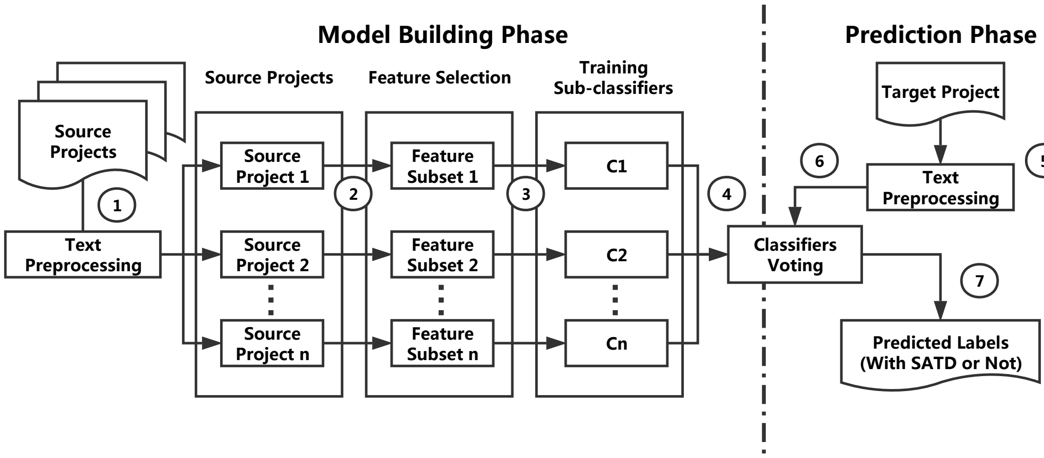

Figure 1: Overall Framework of Our Model

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 [6]. However, their work only focuses on certain types of SATD (i.e., design debt, requirement debt 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 [4]. This tool is able to (i) automatically detect SATD comments in source code through a text-mining-based approach and (ii) list and manage detected comments inside an IDE. It contains two parts: a Java library and an Eclipse plug-in. The Java library provides command-line interfaces and Java APIs. Through these interfaces, users can train the text mining model using their own data and leverage either the build-in model or their own models to identify SATD comments. The Eclipse plug-in, which is the front-end of our tool, uses our pre-trained composite classifier to make detection after a project is imported into Eclipse. Specifically, whenever a developer opens Eclipse, our plug-in will automatically parse all source code files, detect and mark the comments which contains SATD. Once some files are modified, it will re-detect SATD comments and update markers in these files immediately. Moreover, our plug-in also provides an Eclipse view in which all detected SATD comments are listed for management. Our tool is easy to deploy and use. With the help of the Eclipse plug-in, it would be easy for developers and managers to manage SATD and pay back it in time. In addition, using our Java library, users can train and leverage their own model, and integrate SATD Detector into their development tools (e.g., other IDEs or continuous integration tools).

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 [7]. The experimental results show that, on every target project, our tool outperforms Maldonado and Shihab's approach [7] 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 the usage of the Java library, the workflow, life cycle and user interface of the Eclipse plug-in, 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

In general, 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 [4]. 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 using data from each source project. 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. For each source project, we first preprocesses the text descriptions of comments and extracts features (i.e., words) to represent each comment (Step 1). Then, feature selection is applied to select features that are useful for classification and useless features are removed (Step 2). Next, we use the selected features to train a sub-classifier for the target 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 then input features 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) [10] 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., SATD comment or not). In this model, we employ Information Gain (IG) [9, 13] 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.

Sub-classifiers Training: In our tool, we train each sub-classifier using Naive Bayes Multinomial (NBM), which is widely used to analyze text data in software engineering [12, 14–16]. We use the implementation of NBM in Weka [3] with default settings. Note that our approach 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.

2.3 Dataset

We use the dataset provided by the authors of [7] to build our model and evaluate its performance. The dataset contains comments extracted from 8 open source projects, which are ArgoUML, Columba, Jmeter, JFreeChart, Hibernate, JEdit, JRuby, SQuirrel, and the label of each comment, i.e., SATD comment or not. All the labels are manually labeled by the authors of [7], who reported a high level of agreement on the classification results. Therefore, we are confident in the quality of the provided dataset. More information about the dataset can be found in our previous work [4].

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

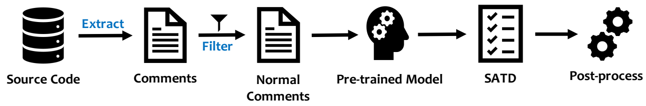

Figure 2: Workflow of Our Eclipse Plug-in

3 SATD DETECTOR

3.1 The Java Library

The Java Library in our tool provides command-line interfaces as well as Java APIs. Both of them provide the ability to re-train the model using user-specified data and detect SATD comments through our pre-trained model or models re-trained by users. The detailed manual of this library can be found in our GitHub repository[2].

3.2 The Eclipse Plug-in

3.2.1 Workflow. Figure 2 shows the workflow of our Eclipse plug-in. First of all, it 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") [9].

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

3.2.2 Life Cycle. After installation, our plug-in 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 plug-in ignores the files in the third party libraries. In our current implementation, the Eclipse plug-in only supports Java projects, and it will not parse non-Java source code files. But our back-end library only cares about source code comments, so it is not limited to Java projects. Once source code files are modified, they will be re-parsed and our plug-in will re-detect SATD in these files immediately. The markers created by our plug-in for SATD comments will not be persisted; hence while users are exiting Eclipse, these markers will be deleted and SATD Detector will then stop.

3.2.3 User Interface. Figure 3 presents the user interface (UI) of our Eclipse plug-in. It 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

[2] https://github.com/Tbabm/SATDDetector-Core

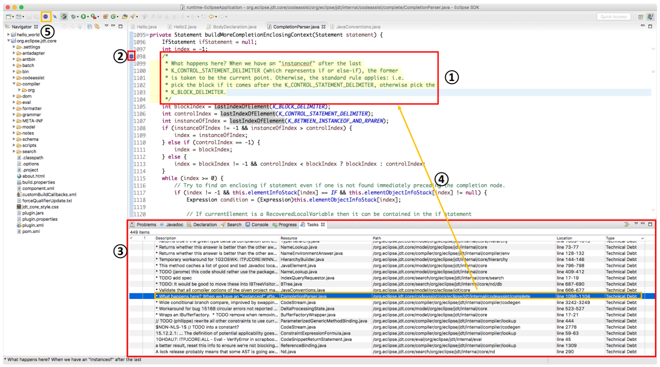

Figure 3: User Interface of Our Eclipse Plug-in

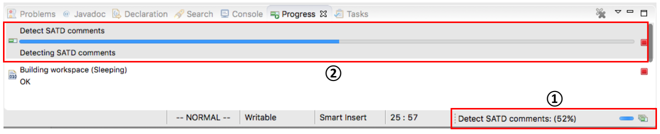

Figure 4: Re-detecting SATD in Background

(③ in Figure 3). This Eclipse view displays details of each detected 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 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, our plug-in 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

The dataset used to evaluate the performance of our tool is described in Section 2.3. 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 if it matches one of the 62 patterns published by Potdar and Shihab [8].
(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 [6] and build a maximum entropy classifier to predict whether a comment contains SATD or not.

The experimental results show that, on every target project our approach achieves the best performance in terms of F1-score, and outperforms the baseline approaches by a substantial margin. After observing the dataset, we find that different projects write SATD comments in different ways. Training sub-classifiers and combining them through the voting mechanism can reduce the bias to certain kind of SATD comments, and thus improve the performance. Readers can refer to our previous work [4] for more details of our evaluation and experimental results.

5 CONCLUSION & FUTURE WORK

In this paper, we present SATD Detector, a tool that is able to automatically detect SATD comments, and help developers manage them in an IDE. This tool consists of a back-end Java library and a front-end Eclipse plug-in. Through the back-end library, users can re-train the text mining model and integrate SATD Detector into other development tools easily. The Eclipse plug-in is able to remind developers and managers of existing SATD and help them pay for SATD in time. We are also interested in providing visualization tools to help developers further analyze SATD comments in different kinds of software projects.

ACKNOWLEDGMENT

This work was partially supported by NSFC Program (No. 61602403 and 61572426).

REFERENCES

[1] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *MSR*.
[2] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* (1993).
[3] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* (2009).
[4] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2017. Identifying self-admitted technical debt in open source projects using text mining. *EMSE* (2017).
[5] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A balancing act: what software practitioners have to say about technical debt. *IEEE software* (2012).
[6] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *TSE* (2017).
[7] Everton da S Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *MTD*.
[8] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *ICSME*.
[9] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* (1986).
[10] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* (1975).
[11] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *SANER*.
[12] Xin Xia, David Lo, Denzil Correa, Ashish Sureka, and Emad Shihab. 2016. It takes two to tango: Deleted stack overflow question prediction with text and meta features. In *COMPSAC*.
[13] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Bo Zhou. 2015. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering* (2015).
[14] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *ICSME*.
[15] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2013. Tag recommendation in software information sites. In *MSR*.
[16] Xin-Li Yang, David Lo, Xin Xia, Qiao Huang, and Jian-Ling Sun. 2017. High-impact bug report identification with imbalanced learning strategies. *Journal of Computer Science and Technology* (2017).