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
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 model 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 contain 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 stemmer1.

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].

---

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

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

---

SATD Detector: A Text-Mining-Based SATD Tool

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

[14] Xiu 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 ICSE.