Doctoral Dissertation

Unlocking Software Documentation: Sentiment Classification and On-hold Self-Admitted Technical Debt Identification

Rungroj Maipradit
Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Kenichi Matsumoto
Software Engineering Lab. (Division of Information Science)

Submitted on August 22, 2022

A Doctoral Dissertation
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of Engineering

Rungroj Maipradit

Unlocking Software Documentation: Sentiment Classification and On-hold Self-Admitted Technical Debt Identification[*]

Rungroj Maipradit

Abstract

Software documents refer to all written documents in software development, which play a crucial role in knowledge sharing between software developers. Despite the benefits of software documentation, document creation and maintenance are frequently overlooked. Many software documents are often outdated, and many small to medium software projects have little to no software documentation.

This thesis presumes that the problem of little to no software documentation and outdated documents can be tackled by unlocking software documents to show the benefit of existing information. To address this, this thesis first unlocks software document information by proposing a new technique for accessing existing information on software engineering data sets using sentiment classification. The results show that using automated machine learning with n-gram inverse document frequency shows promising results in tackling this problem. Second, to reduce outdated documents by finding what kinds of tasks are amenable to automated management. And found one particular class of debt amenable to automated management: on-hold SATD, i.e., debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality to be implemented elsewhere. Third, this thesis investigates the potential of removing outdated documents by automatically detecting on-hold SATD and identifying its condition. The results show that the proposed design can reliably

[*]Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, August 22, 2022.

identify on-hold SATD and also mine the issue tracker to check if the On-hold SATD instances are "superfluous" and can be removed.

In all, this thesis emphasizes the benefit of unlocking software documents for software development. Furthermore, this thesis provides practical implications for extracting information from software documents through sentiment classification and for managing outdated software documents through suggesting the removal of outdated on-hold SATD comments.

Keywords:

Software document classification, N-gram IDF, Automated machine learning, Self-admitted technical debt, Sentiment Classification

Acknowledgements

I would first like to express my sincere gratitude to my thesis supervisor, Prof. Kenichi Matsumoto, for giving me opportunities to study a doctoral's degree and also provide guidance and encouragement during my time in NAIST.

I would also like to express my gratitude to my co-supervisor, Assoc. Prof. Hideaki Hata for providing a suggestion, teaching and help me to start researching.

Besides my supervisor and my co-supervisor, I would like to thank the rest of my thesis committee, Prof. Shoji Kasahara, Assoc. Prof. Takashi Ishio, and Assist. Prof. Raula Gaikovina Kula, and Senior lecturer Christoph Treude, for their invaluable comments and suggestions to improve the quality of my research.

I would also like to sincerely thank Shade Ruangwan, and Bodin Chinthanet, who have been like brothers to me, for sharing their experience and advice.

I would like to thank my friends in NAIST and my labmates in Software Engineering laboratory for a very enjoyable time during my time in NAIST.

Last but not least, I would like to thank my family: my parents, my brother for their love, support and encouragement to pursue my study in a doctoral's degree. Without them, this thesis might not be written. Finally, I would like to express my most sincere appreciation to MEXT and NAIST for all kind of supports.

List of Publications

- **Wait for it: identifying "On-Hold" self-admitted technical debt**
  Rungroj Maipradit, Christoph Treude, Hideaki Hata, Kenichi Matsumoto Empirical Software Engineering (EMSE), 25: 3770–3798 (2020). (Accepted as a journal paper)
  - Present at ICSE 2021 Journal-First.
  - Received SIGSE Outstanding Paper Award 2021.
- **Automated Identification of On-hold Self-admitted Technical Debt**
  Rungroj Maipradit, Bin Lin, Csaba Nagy, Gabriele Bavota, Michele Lanza, Hideaki Hata, Kenichi Matsumoto 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 54-64 (Accepted as a conference paper)
- **Sentiment Classification Using N-Gram Inverse Document Frequency and Automated Machine Learning**
  Rungroj Maipradit, Hideaki Hata, Kenichi Matsumoto in IEEE Software, vol. 36, no. 5, pp. 65-70, Sept.-Oct. 2019 (Accepted as a journal paper)

Contents

List of Figures

List of Tables

1 | Introduction

Software documents refer to all written documents in software development which provide information about software for developers and users, including how software works, how to use it, and how it provides different types of information to people in different roles [29].

Correct, consistent, and complete software documentation plays an important role for developers in terms of knowledge sharing and software maintenance [33]. Additionally, high-quality and useful software documents improve the performance of engineers working on software projects and are believed to be one of the key factors in producing high-quality software [21, 64].

Despite the benefits of software documentation, document creation and maintenance are frequently overlooked. Especially in small to medium software projects, which have little to no software documentation [20]. Moreover, software documentation is often poorly written and frequently outdated. In the worst cases, some parts of the documentation are untrustworthy [39].

Sentiment analysis is defined as the task of finding the opinions of authors behind the texts. This allows users and companies to monitor their reputation and receive feedback in a timely manner [17]. In software development, sentiment analysis helps developers to analyze emotions behind various tasks, such as app reviews, responses to bug reports, and emotions in commit messages. Despite the benefit of sentiment analysis, current tools provide unreliable results when applied to software documents due to the fact that these tools are not developed for software engineering tasks [44].

One of the software documents is code comments, which exist within source code and are used by developers. Code comments assist developers understand source code and also help in software maintenance [78]. In many cases, software developers know that their current implementation is not optimal and indicate this using a source code comment (*i.e.,* self-admitted technical debt) [58]. However, current research is largely focused on the detection and classification of self-admitted technical debt, but has spent less effort on approaches to address the debt automatically.

This thesis focuses on two challenges in software documentation: (1) the lack of a liability sentiment analysis tool that works in a software engineering environment; and (2) assisting in software document management by identifying and removing unnecessary self-admitted technical debt (i.e., situations where a software developer knows that their current implementation is not optimal and indicates this using a source code comment).

1 Contributions

The main contributions of this thesis can be classified into two categories: sentiment analysis, self-admitted technical debt.

Access existing information using sentiment analysis

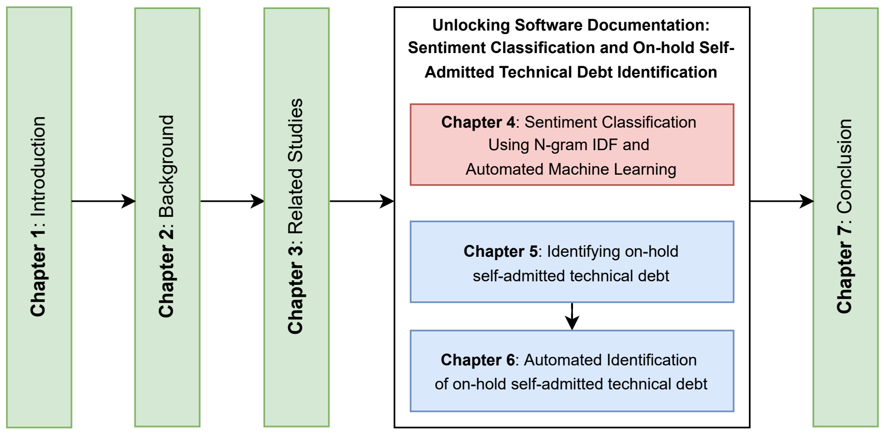
- The design and evaluation of a classifier for sentiment analysis in software documents. (Chapter 4)

Managing software documents by removing outdated self-Admitted Technical Debt

- A qualitative study on the removal of self-admitted technical debt. (Chapter 5)
- The definition of self-admitted technical debt which is amenable to automated management (on-hold SATD). (Chapter 5)
- The design and evaluation of a classifier for on-hold self-admitted technical debt. (Chapter 5)

- Large-scale empirical study to automatically detect on on-hold SATD. (Chapter 6)
- Large-scale empirical study to identify ready to be removed on-hold SATD. (Chapter 6)

2 Outline



Chapter 1: Introduction
Chapter 2: Background
Chapter 3: Related Studies
Unlocking Software Documentation: Sentiment Classification and On-hold Self-Admitted Technical Debt Identification
Chapter 4: Sentiment Classification Using N-gram IDF and Automated Machine Learning
Chapter 5: Identifying on-hold self-admitted technical debt
Chapter 6: Automated Identification of on-hold self-admitted technical debt
Chapter 7: Conclusion

Figure 1.1. An overview of the scope of the thesis.

In this section, I provide an outline of this thesis. Figure 1.1 illustrates the structure of the thesis and the outcomes of each section. The details of the rest of this thesis is structured as follows:

- **Chapter 4** presents analysis of sentiment in software documents. I design and implement sentiment classification using N-gram IDF and automated machine learning.
- **Chapter 5** introduces a new type of self-admitted technical debt that is able to manage management called “on-hold SATD”. Then propose tool support that can help developers manage self-admitted technical debt more effectively.
- **Chapter 6** presents large-scale empirical study to automatically detect on-hold SATD and identify ready to be removed on-hold SATD.

2 | Background

In this section, I introduce the component of classifier, scope of software documents, and relationship between sentiment analysis and self-admitted technical debt.

1 Component

I introduce the component of the classifier. The classifier consists of two part: N-gram IDF and automated machine learning.

1.1 N-gram IDF

N-gram is all sequences of word length n that appear in a text. Generally, n-gram phrases are considered to be informative and useful compared to single words. Nevertheless, using all n-gram terms is not useful because they provide a large volume of data. To overcome such problem of using n-grams, we utilize N-gram IDF.

N-gram IDF is a theoretical extension of IDF for handing multiple terms and phrases by bridging the theoretical gap between term weighting and multi-word expression extraction [68, 71]. N-gram IDF is able to find dominant N-grams among overlapping ones and extract key terms of any length.

Difference between N-gram IDF and IDF, IDF does not handle N-gram phrases when N>1 properly. IDF gives higher weight to term that appears less in documents, however unnatural n-gram terms are less likely to appear in documents.

1.2 Automated Machine Learning

Machine learning is an application based on algorithms which have an ability to automatically learn from data without being explicitly programmed.

In machine learning, two problems are known: (1) no single machine learning method performs best on all data sets, and (2) some machine learning methods rely heavily on hyperparameter optimization.

Automated machine learning aims to optimize choosing a good algorithm and feature preprocessing steps [18]. To obtain the best performance, we apply auto-sklearn [18], a tool of automated machine learning.

Auto-sklearn addresses these problems as a joint optimization problem [18]. Auto-sklearn includes 15 base classification algorithms from Scikit-learn libraries and produces results from an ensemble of classifiers derived by Bayesian optimization [18].

2 Scope

Software documentation refer to all written software documents in software development. However, only some software documents are consider in this thesis.

2.1 Documentation through code comments

Code comments are key to understand source code implementation by enhance the readability of the code. In many cases, developers know when they are about to cause technical debt, and they leave documentation to indicate its presence. Furthermore, when numerous developers work on the same project, it implies that other people will examine the code and comments to understand it.

Code comments have the following characteristics: (1) they are primarily used by developers; (2) they are used as a remainder or explanation of the code; (3) they exist within the code file ; and (4) they contain technical keywords.

2.2 Documentation through discussion channels

The other documents which are considered in this thesis are those that provide feedback through discussion channels, which come from three sources. A question and answer from Stack Overflow, a question and answer platform for software developers; reviews of mobile applications from Google Play and the Android application store; and comments in the Jira issue tracker.

Documents inside discussion channels have the following characteristics: (1) they are used by both developers and users; (2) they are used for various tasks (e.g., providing feedback for app reviews; providing answers for stack overflow; and providing comments for issue trackers); (3) they exist in an outside repository ; and (4) they contain technical keywords.

3 Relationship between sentiment analysis and self-admitted technical debt

The sentiment analysis chapter analyses software documents through discussion channels, whereas the self-admitted technical debt chapter analyses documentation through code comments. The similarities between these two challenges are that they both contain technical terms and use n-gram IDF to extract keywords.

The term abstraction in the data preparation process is one of the primary distinctions between the two challenges. Due to keyword inside technical debt specific to the project, and we are not interested in their content but instead interested in their types.

3 Related Studies

Complementary related works are introduced throughout the paper, in this Chapter, I discuss some key related works.

Sentiment analysis in software engineering documents

Panichella et al. [54] classified app reviews into categories relevant to software maintenance and evolution. One of the features the authors used was sentiment analysis. Combining three features: Natural Language Processing, Text Analysis, and Sentiment Analysis, classifier achieved better results than using each technique individually.

Ortu et al. [52] analyzed the relationship between sentiment, emotions, and politeness of developers' comments in Jira issue tracker and the duration it takes to fix that issue. They found that positive emotions such as joy and love in comments were linked to shorter fix times, while negative emotions such as sadness in comments were linked to longer fix times.

Zhang and Hou [95] extracted problematic APIs from online discussions. The authors also discovered that negative sentence-based solutions are more likely than other approaches to contain problematic API features.

Tourani et al. [82] applied sentiment analysis tool namely Sentistrength to extract information from mailing lists. However, they found the tool has low percision 29.56% for positive, and 13.18% for negative. Due to ambiguities from technical term.

Islam and Zibran [30] performed indepth exploratory study on public benchmark dataset which created from Jira issue tracker to exposing the difficulties in automatic sentiment analysis. And also proposed a tool called SentiStrength-SE to tackle problem, the tools achieved 73.85% precision and 85% recall.

Jongeling et al. [31], Lin et al. [44] conducted sentiment analysis using existing tools on software documents, which include the tools proposed by Islam and Zibran [30]. In all cases they found that current tools are not ready to use due to low accuracy and lack of agreement between tools.

Impact of self-admitted technical debt

Sierra et al. [74] conducted a survey about self-admitted technical debt by investigating three categories: (i) detection, (ii) comprehension, and (iii) repayment. Detection focuses on identifying and detecting self-admitted technical debt. Comprehension studies the life cycle of self-admitted technical debt. Repayment focuses on removal of self-admitted technical debt. This research found a lack of research related to the repayment of self-admitted technical debt.

Maldonado et al. [48] studied the removal of self-admitted technical debt by applying natural language processing to self-admitted technical debt. They found that (i) the majority of self-admitted technical debt was removed, (ii) self-admitted technical debt was often removed by the person who introduced it, and (iii) self-admitted technical debt lasts between 18 to 172 days (median). Using a survey, the authors also found that developers mostly use self-admitted technical debt to track bugs and code that requires improvement. Developers mostly remove self-admitted technical debt when they are fixing bugs or adding new features.

Zampetti et al. [93] conducted an in-depth quantitative and qualitative study of self-admitted technical debt. They found that (i) 20% to 50% of the corresponding comments were accidentally removed when entire methods or classes were dropped, (ii) 8% of self-admitted technical debt removals were indicated in the commit messages, and (iii) most of the self-admitted technical debt requires complex changes, often changing method calls or conditionals.

Bavota and Russo [4] introduced a large-scale empirical study across 159 software projects. From this data they performed manual analysis of 366 comments, showing (i) an average of 51 self-admitted technical debt comments per system,

(ii) that self-admitted technical debt consists of 30% code debt, 20% defect debt, and 20% requirement debt, (iii) the number of self-admitted technical debt comments is increasing over time, and (iv) on average it takes over 1,000 commits before self-admitted technical debt is fixed.

Wehaibi et al. [86] studied the relation between self-admitted technical debt and software quality based on five open source projects (i.e., Hadoop, Chromium, Cassandra, Spark, and Tomcat). Their result showed that (i) there is no clear evidence that files with self-admitted technical debt had more defects than other files, (ii) compared with self-admitted technical debt changes, non-debt changes had a higher chance of introducing other debt, but (iii) changes related to self-admitted technical debt were more difficult to achieve.

Mensah et al. [51] introduced a prioritization scheme. After running this scheme on four open source projects, they found four causes of self-admitted technical debt which was code smells (23.2%), complicated and complex task (22.0%), inadequate code testing (21.2%), and unexpected code performance (17.4%). The result also showed that self-admitted technical design debt was prone to software bugs, and that for highly prioritized self-admitted technical debt tasks, more than ten lines of code were required to address the debt.

Kamei et al. [34] used analytics to quantify the interest of self-admitted technical debt to see how much of the technical debt incurs positive interest, i.e., debt that indeed costs more to pay off in the future. They found that approximately 42–44% of the technical debt in their case study incurred positive interest.

Palomba et al. [53] conducted an exploratory study on the relationship between changes and refactoring and found that developers tend to apply a higher number of refactoring operations aimed at improving maintainability and comprehensibility of the source code when fixing bugs. In contrast, when new features are implemented, more complex refactoring operations are performed to improve code cohesion. In most cases, the underlying reasons behind the application of such refactoring operations were the presence of duplicate code or previously introduced self-admitted technical debt.

Mensah et al. [50] propose a new technique to estimate Rework Effort, i.e., the effort involved to resolve self-admitted technical debt. They performed an exploratory study using text mining to extract self-admitted technical debt from source code comments. In order to extract source code comments, the authors apply text mining on four open source projects. The result from four projects shows a rework effort between 13 and 32 commented lines of code on average per self-admitted technical debt comment.

Self-admitted technical debt Identification and Classification

Potdar and Shihab [58] tried to identify self-admitted technical debt by looking into source-code comments in four open source project (i.e., Eclipse, Chromium OS, Apache HTTP Server, and ArgoUML). Their study showed that (i) the amount of debt in these project ranged between 2.4% and 31% of all files, (ii) debt was created mostly by developers with more experience, and time pressures and code complexity did not correlate with the amount of self-admitted technical debt, and (iii) only 26.3% to 63.5% of self-admitted technical debt comments were removed.

de Freitas Farias et al. [15] proposed a tool called CVM-TD (Contextualized Vocabulary Model for identifying Technical Debt) to identify technical debt by analyzing code comments. The authors performed an exploratory study on two open source projects. The result indicated that (1) developers use dimensions of CVM-TD when writing code comments, (2) CVM-TD provides vocabulary that may be used to detect technical debt, and (3) models need to be calibrated.

de F. Farias et al. [14] investigated the use of CVM-TD with the purpose of characterizing factors that affect the accuracy of the identification of technical debt, and the most chosen patterns by participants as decisive to indicate technical debt items. The authors conducted a controlled experiment to evaluate CVM-TD, considering factors such as English skills and experience of developers.

Silva et al. [75] investigated the identification of technical debt in pull requests. The authors found that the most common technical debt categories are design, test, and project convention.

da Silva Maldonado et al. [13] tried identifying design-related and requirement-related self-admitted technical debt using a maximum entropy classifier.

Huang et al. [26] tried classifying comments in terms of whether they contained self-admitted technical debt or not, and reported that their proposal outperformed the baseline method.

Maldonado and Shihab [47] studied types of self-admitted technical debt using source code comments. This study classified types of self-admitted technical debt into design debt, defect debt, documentation debt, requirement debt, and test debt. The most common type of self-admitted technical debt is design debt and the second most common type is requirement debt. Self-admitted technical debt consist of 42% to 84% design debt, and 5% to 45% requirement debt.

Zampetti et al. [92] developed a machine learning approach to recommend when design technical debt should be self-admitted. They found their approach to achieve an average precision of about 50% and a recall of 52%. When predicting cross-projects, the performance of the approach improved to an average precision of 67% and a recall of 55%.

Yan et al. [89] identify self-admitted technical debt using change-level self-admitted technical debt determination. This model identifies whether a change introduces self-admitted technical debt. In order to create the model, they identified technical debt using all versions of source code comments. Then, they manually label changes that introduce technical debt in comments and extract 25 features which belong to three groups, i.e., diffusion, history, and message. After that, they create a classifier using random forest. Across seven projects, this model achieves an AUC of 0.82 and cost-effectiveness of 0.80.

Flisar and Podgorelec [19] developed a new method to detect self-admitted technical debt using word embedding trained from unlabeled code comments. They then apply feature selection methods (Chi-square, Information Gain, and Mutual Information), and use three classification algorithms (Naive Bayes, Support Vector Machine, and Maximum Entropy) to test on ten open source projects. Their proposed method was able to achieve 82% correct predictions.

Liu et al. [46] proposed a self-admitted technical debt detector tool which is able to detect debt comments using text mining and is able to manage detected comments in an IDE via an Eclipse plug-in.

Ren et al. [61] proposed a Convolutional Neural Network for classifying code comments as self-admitted technical debt or not, based on ten open source projects. Their approach outperforms text-mining-based methods both in terms of within-project and cross-project prediction.

Empirical Studies on (self-admitted) Technical Debt

Storey et al. [79] studied how annotations in code comments (*e.g.,* TODO, FIXME) are used by developers to keep track of tasks. Several types of activities are supported by these annotations, *e.g.,* the usage of TODOs to ask questions to other developers during code comprehension. These annotations are a subset of the ones used nowadays to detect SATD.

Guo et al. [23] studied a specific technical debt instance to assessing its impact on the project costs. Their findings confirmed the harmfulness of technical debt, showing that the delayed task resulted in tripled implementation costs.

Klinger et al. [35] investigated how decisions to acquire technical debt are made within IBM by interviewing four technical architects. They found that technical debt is often due to imposed requirements to meet a specific deadline sacrificing quality. Also, the interviewed architects reported a lack of effective communication between technical and non-technical stakeholders involved in technical debt management.

Lim et al. [42] interviewed practitioners (35 in this case) to investigate their perspective on TD. They found that most of the participants were familiar with the notion of TD and they do consider it as a poor programming practice, but more as an *intentional decision to trade off competing concerns during development* [42]. Practitioners also highlighted the difficulty in measuring the cost of TD. Similarly, Kruchten et al. [38] reported their understanding of the technical debt in industry as the result of a four-year interaction with practitioners.

Spinola et al. [77] asked 37 practitioners to validate 14 statements about TD (*e.g.,* "*The root cause of most technical debt is pressure from the customer*" [62]). The statement achieving the highest agreement was "*If technical debt is not managed effectively, maintenance costs will increase at a rate that will eventually outrun the value it delivers to customers*".

Kruchten et al. [37] provided theoretical foundations to the concept of TD by presenting the "technical debt landscape", classifying TD as visible or invisible and highlighting the debt types causing evolvability and maintainability issues. Alves et al. [3] proposed an ontology of terms on technical debt.

Potdar and Shihab [57] introduced the notion of SATD by mining five software systems to investigate (i) the amount of SATD they contain, (ii) the factors promoting the introduction of the SATD, and (iii) how likely is the SATD to be removed. Bavota and Russo [5] performed a differentiated replication of that study involving a larger set of subject systems (159), confirming the findings of the original study.

Zazworka et al. [94] studied the overlap between the technical debt instances detected by automated tools and by manual inspection, finding very little overlap.

Maldonado and Shihab [12] used the TD classification by Alves et al. [3] to investigate the types of SATD more diffused in open source projects. They identified 33k comments in five software systems reporting SATD. These comments have been manually read by one of the authors who found as the vast majority of them (∼60%) reported design debt.

Wehaibi et al. [86] studied the relationship between SATD and software quality, finding that files with SATD do not have more defects compared to files without SATD, but that changes in the context of SATD are more complex. Sierra et al. [73] conducted a survey about SATD research, categorizing it into: detection, comprehension, and repayment. They found a lack of research related to repayment and management of SATD.

Automatic detection/management of SATD

Potdar and Shihab [57], the authors identified SATD using 62 textual patterns. The patterns can be matched in code comments of a previously unseen project to identify SATD. Farias et al. [10] built on top of these 62 patterns and developed a model called CVM-TD (Contextualized Vocabulary Model for identifying TD) that exploits combinations of the patterns to identify different types of technical debt.

Maldonado et al. [11] presented an approach to automatically identify design and requirement SATD by applying Natural Language Processing (NLP) on code comments. A study performed on ten open source projects showed the superiority of their approach as compared to the state-of-the-art, represented at that time by the above-described pattern-based techniques. Wattanakriengkrai et al. [85] developed a classifier to identify design and requirements SATD using N-gram

IDF and automated machine learning on Maldonado's dataset. Comparing the result with the previous study [11], the classifier outperforms the NLP approach in both design and requirement. A similar idea has also been exploited by Huang et al. [26] that leveraged text-mining for SATD identification. Also in this case, the approach performed better than the pattern-based approach by Potdar and Shihab [57]. This approach is also available as an Eclipse plug-in [45].

Ren et al. [60] proposed an approach based on Convolution Neural Networks to classify code comments into SATD or non-SATD. An experiment performed on ten projects and 63k comments showed that their approach outperforms text mining techniques both for within-project and cross-project prediction.

Zampetti et al. [90] presented TEDIOUS (TEchnical Debt IdentificatiOn System), an approach to train a recommender to suggest developers writing new code when to self-admit design TD, or improve the code being written. TEDIOUS achieves an average precision of $\sim$50%. Yan et al. [89] proposed a model to determine whether a change introduces SATD. They manually labeled changes that introduced SATD in the past and built a model exploiting 25 features to characterize SATD-introducing changes. An empirical study across $\sim$100k changes reported an AUC for the model of 0.82.

4 | Sentiment Classification Using N-gram IDF and Automated Machine Learning

Sentiment analysis is the process to classify the writer's opinion in text into positive, neutral or negative. Sentiment analysis has been used to several practical purposes in software development. However, it is reported that no tool is ready to accurately classify sentences to negative, neutral, or positive, even if tools are specifically customized for certain software engineering tasks. In this chapter we propose a machine-learning based approach using n-gram features and an automated machine learning tool for sentiment classification.

1 Introduction

As software development is a human activity, identifying affective states in messages has become an important challenge to extract meaningful information. Sentiment analysis has been used to several practical purposes, such as identifying problematic API features Zhang and Hou [95], assessing the polarity of app reviews Panichella et al. [54], clarifying the impact of sentiment expressions to the issue resolution time Ortu et al. [52], and so on.

Because of the poor accuracy of existing sentiment analysis tools trained with general sentiment expressions Jongeling et al. [31], recent studies have tried to customize such tools with software engineering datasets Lin et al. [44]. However, it is reported that no tool is ready to accurately classify sentences to negative, neutral, or positive, even if tools are specifically customized for certain software engineering tasks Lin et al. [44].

One of the difficulties in sentiment classification is the limitation in a bag-of-words model or polarity shifting because of function words and constructions in sentences Li et al. [40]. Even if there are positive single words, the whole sentence can be negative because of negation, for example. For this challenge, Lin et al. adopted `Stanford CoreNLP`, a recursive neural network based approach that can take into account the composition of words Socher et al. [76], and prepared a software-engineering-specific sentiment dataset from a Stack Overflow dump Lin et al. [44]. Despite their large amount of effort on fine-grained labeling to the tree structure of sentences, they reported negative results (low accuracy) Lin et al. [44].

In this chapter we propose a machine-learning based approach using n-gram features and an automated machine learning tool for sentiment classification. Although n-gram phrases are considered to be informative and useful compared to single words, using all n-gram phrases is not a good idea because of the large volume of data and many useless features Bespalov et al. [7]. To address this problem, we utilize n-gram IDF, a theoretical extension of Inverse Document Frequency (IDF) proposed by Shirakawa et al. Shirakawa et al. [69]. IDF measures how much information the word provides; but it cannot handle multiple words. N-gram IDF is capable of handling n-gram phrases; therefore, we can extract useful n-gram phrases.

Automated machine learning is an emerging research area targeting the progressive automation of machine learning. Two important problems are known in machine learning: no single machine learning technique give the best result on all datasets, and hyperparameter optimization is needed. Automated machine learning addresses these problems by running multiple classifiers and tries different parameters to optimize the performance. In this study, we use auto-sklearn, which contains 15 classification algorithms (random forest, kernel SVM, etc.), 14 feature pre-processing solutions (PCA, nystroem sampler, etc.), and 4 data pre-process solutions (one-hot encoding, rescaling, etc.) Feurer et al. [18]. Using n-gram IDF and auto-sklearn tools, Wattanakriengkrai et al. outperformed

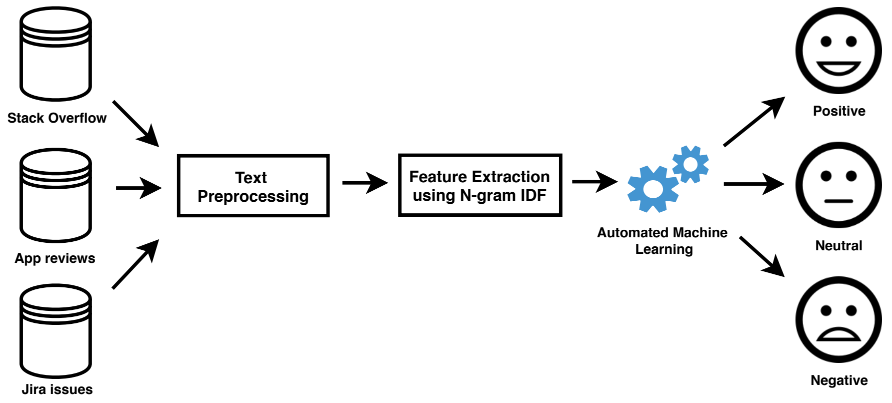


Figure 4.1. An overview of our sentiment classification approach

the state-of-the-art self-admitted technical debt identification Wattanakriengkrai et al. [84].

2 Method

Figure 4.1 shows an overview of our method with the following three components.

Text Preprocessing. Messages in software document sometimes contain special characters. We remove characters that are neither English characters nor numbers. Stop words are also removed by using spaCy library. spaCy tokenizes text and finds part of speech and tag of each token and also checks whether the token appears in the stop word list.

Feature extraction using N-gram IDF. N-gram IDF is a theoretical extension of IDF for handling words and phrases of any length by bridging the gap between term weighting and multiword expression extraction. N-gram IDF can identify dominant n-grams among overlapping ones Shirakawa et al. [69]. In this study, we use N-gram Weighting Scheme tool Shirakawa et al. [69]. The result after applying this tool is a dictionary of n-gram phrases ($n \leq 10$ as a default setting) with their frequencies. N-gram phrases appear only one time in the whole document (frequency equal one) are removed, since they are not useful for training.

Automated machine learning. To classify sentences into positive, neutral, and negative, we use auto-sklearn, an automated machine learning tool. Automated machine learning tries running multiple classifiers and applying different parameters to derive better performances. Auto-sklearn composes two steps: meta-learning and automated ensemble construction Feurer et al. [18]. We ran auto-sklearn with 64 gigabytes of memories, set 90 minutes limitation for each round, and configure it to optimize for a weighted F1 value, an average F1 value for three classes weighted by the number of true instance of each class.

3 Evaluation

3.1 Datasets and Settings

We employ a dataset provided by Lin et al.'s work Lin et al. [44]. There are three types of document in the dataset; sentences in questions and answers on Stack Overflow, reviews of mobile applications, and comments on Jira issue trackers. Each dataset has texts and labels of positive, neutral and negative.

Since our method requires just labeled (positive, neutral, or negative) sentences, we can use dataset-specific data for training. For training and testing, we apply 10-fold cross-validation for each dataset, that is, we split sentences in a dataset into ten subsets with maintaining the ratio from the oracles by using the function StratifiedShuffleSplit in scikit-learn.

3.2 Sentiment Classification Tools

To assess the performance of our method, we compare our method with tools presented in the previous work Lin et al. [44].

SentiStrength estimates the strength of positive and negative scores based on the sentiment word strength list prepared from MySpace comments Thelwall et al. [81].

NLTK is a natural language toolkit and is able to do sentiment analysis based on lexicon and rule-based VADER (Valence Aware Dictionary and sEntiment Reasoner), which is specifically tuned for sentiments expressed in social media Hutto and Gilbert [27].

Stanford CoreNLP adopts a deep learning model to compute the sentiment based on how words compose the meaning of the sentence Socher et al. [76]. The

model has been trained on movie reviews.
SentiStrength-SE is a tool build on top of SentiStrength and has been trained on JIRA issue comments Islam and Zibran [30].
Stanford CoreNLP SO prepared Stack Overflow discussions to train a model of Standford CoreNLP Lin et al. [44].

3.3 Result

Table 4.1 shows the number of correct predictions, precision, recall, and F1 values with all tools including our method (`n-gram auto-sklearn`). These values were presented in Lin et al. [44] (F1 values are calculated by us). Precision, recall, and F1 values are derived as the average from the 10 rounds of our 10-fold cross-validation, since same data can appear in different rounds with Stratified-ShuffleSplit.

We can see that the number of correct predictions are higher with our method in all three datasets, and our method achieved the highest F1 values for all three positive, all three negative, and one neutral. Although the values are low for the neutral class in App reviews, this is because the amount of neutral sentences is small in this dataset. In summary, our method using n-gram IDF and automated machine learning (auto-sklearn) largely outperformed existing sentiment analysis tools. Since our method relies on n-gram phrases, it cannot properly classify text without known n-gram phrases. Although a negative sentence "They all fail with the following error" was correctly classified with SentiStrength, NLTK, and Stanford CoreNLP, our method classified as neutral. Preparing more data is preferable to improve the performance.

Note that only our method trains within-dataset for all three cases. Although within-dataset training can improve the performances of other tools, preparing training data for those sentiment analysis tools require considerable manual effort Lin et al. [44]. Since our method can automatically learn dataset-specific text features, learning within-dataset is practically feasible.

The following are classifiers achieved the top three performances in `auto-sklearn` for each dataset.

- **Stack Overflow**: Linear Discriminant Analysis, LibSVM Support Vector Classification, and Liblinear Support Vector Classification

Table 4.1. The comparison result of the number of corrected prediction, precision, recall, and f1-score

dataset	tool	# correct	positive			neutral			negative		
		prediction	precision	recall	F1	precision	recall	F1	precision	recall	F1
Stack Overflow	SentiStrength	1043	0.200	**0.359**	0.257	0.858	0.772	0.813	0.397	0.433	0.414
positive: 178	NLTK	1168	0.317	0.244	0.276	0.815	**0.941**	0.873	**0.625**	0.084	0.148
neutral: 1,191	Standford CoreNLP	604	0.231	0.344	0.276	**0.884**	0.344	0.495	0.177	**0.837**	0.292
negative: 131	SentiStrength-SE	1170	0.312	0.221	0.259	0.826	0.930	0.875	0.500	0.185	0.270
sum: 1,500	Stanford CoreNLP SO	1139	0.317	0.145	0.199	0.836	0.886	0.860	0.365	0.365	0.365
	N-gram auto-sklearn	**1317**	**0.667**	0.316	**0.418**	0.871	0.939	**0.904**	0.600	0.472	**0.514**
	N-gram auto-sklearn with SMOTE†	-	0.680	0.005	0.009	0.344	0.930	0.499	0.657	0.160	0.251
App reviews	SentiStrength	213	0.745	0.866	0.801	0.113	0.320	0.167	0.815	0.338	0.478
positive: 186	NLTK	184	0.751	0.812	0.780	0.093	**0.440**	0.154	**1.000**	0.169	0.289
neutral: 25	Standford CoreNLP	237	0.831	0.715	0.769	**0.176**	0.240	**0.203**	0.667	0.754	0.708
negative: 130	SentiStrength-SE	201	0.741	0.817	0.777	0.106	0.400	0.168	0.929	0.300	0.454
sum: 341	Stanford CoreNLP SO	142	0.770	0.253	0.381	0.084	0.320	0.133	0.470	0.669	0.552
	N-gram auto-sklearn	**293**	**0.822**	**0.894**	**0.853**	0.083	0.066	0.073	0.823	**0.808**	**0.807**
	N-gram auto-sklearn with SMOTE†	-	0.520	0.885	0.641	0.100	0.058	0.073	0.648	0.622	0.607
Jira issues	SentiStrength	714	0.850	**0.921**	0.884	-	-	-	0.993	0.703	0.823
positive: 290	NLTK	276	0.840	0.362	0.506	-	-	-	**1.000**	0.269	0.424
neutral: 0	Standford CoreNLP	626	0.726	0.621	0.669	-	-	-	0.945	0.701	0.805
negative: 636	SentiStrength-SE	704	0.948	0.883	0.914	-	-	-	0.996	0.704	0.825
sum: 926	Stanford CoreNLP SO	333	0.635	0.252	0.361	-	-	-	0.724	0.409	0.523
	N-gram auto-sklearn	**884**	**0.960**	0.839	**0.893**	-	-	-	0.932	**0.982**	**0.956**
	N-gram auto-sklearn with SMOTE†	-	0.986	0.704	0.809	-	-	-	0.781	0.988	0.872

† Applying SMOTE, a oversampling technique, for my method.

- **App reviews**: Random forest, LibSVM Support Vector Classification, and Naive Bayes classifier for multinomial models
- **Jira issues**: Naive Bayes classifier for multinomial models, Adaptive Boosting, and Linear Discriminant Analysis

If we have a new unlabeled dataset, we can first try one of the common classifiers. By manually annotating labels, we can try auto-sklearn to find the best classifier for the dataset.

4 Discussions

4.1 Threats to Validity

Competitive sentiment classification tools are trained only with specific dataset. Since our method is based on a general text classification approach, we could conduct within-dataset training. However, because of a considerable amount of manual effort for training sentiment classification tools, they had been trained only with specific datasets. Although we think this is an advantage of our method, the comparison is not with the same condition.

Imbalanced data. In this multi-class classification, some datasets are not balanced; neutral class is the majority for Stack Overflow and no neural data for Jira issues. Applying some balancing techniques may improve the overall performances.

Our study might not be generalize to other datasets. Our approach is applied to comments, reviews, and questions and answers. Other types of document related to software engineering may derive different results.

4.2 Obtained N-gram Phrases

Why our method achieved high accuracy performance in sentiment classification? Table 4.2 shows selected n-gram phrases, which were useful for classifying positive, neutral, and negative statements, obtained in each dataset. For negative, we see 'bug', a software-engineering-specific negative word, and many negation expressions. We can also see reasonable n-gram phrases for positive cases, such

Table 4.2. Obtained n-gram phrases (selected)

dataset	positive	neutral	negative
Stack Overflow	‘useful’	‘suggest’, ‘using’	‘wrong’
	‘the’, ‘simplest’, ‘solution’	‘everyone’	‘bug’
	‘more’, ‘efficient’, ‘to’	‘limitation’	‘i’, ‘do’, ‘n’, ‘t’, ‘understand’
	‘helpful’	‘appointment’	‘i’, ‘do’, ‘n’, ‘t’, ‘know’
	‘a’, ‘good’, ‘example’	‘technical’	‘does’, ‘n’, ‘t’, ‘work’
App reviews	‘thanks’, ‘for’	‘let’, ‘me’, ‘know’	‘impossible’, ‘to’, ‘use’
	‘really’, ‘like’, ‘this’, ‘app’	‘gets’, ‘the’, ‘job’, ‘done’	‘game’, ‘constantly’, ‘freezes’
	‘well’, ‘done’	‘android’, ‘device’	‘uninstalled’
	‘awesome’	‘allow’	‘disappointing’
	‘easy’, ‘to’, ‘use’, ‘and’	‘suggestions’	‘lack’, ‘of’, ‘features’
Jira issues	‘thank’, ‘you’, ‘very’, ‘much’	-	‘problems’
	‘looks’, ‘good’, ‘we’	-	‘is’, ‘bad’
	‘awesome’, ‘work’	-	‘i’, ‘disagree’
	‘thanks’, ‘for’, ‘your’, ‘help’	-	‘really’, ‘sucks’
	‘awesome’, ‘stuff’	-	‘this’, ‘bug’

as ‘useful’, ‘really like this app’, ‘awesome work’, and so on. We can think that because of these dataset-specific positive, neutral, and negative patterns, n-gram IDF worked well for resolving the limitation in a bag-of-words model and our method have resulted in good performance.

5 Conclusion

In this chapter, we proposed a sentiment classification method using n-gram IDF and automated machine learning. We apply this method on three datasets including question and answer from Stack Overflow, reviews of mobile applications, and comments on Jira issue trackers.

Our good classification performance is not based only on an advanced automated machine learning. N-gram IDF also worked well to capture dataset-specific, software-engineering-related positive, neutral, and negative expressions. Because of the capability of extracting useful sentiment expressions with n-gram IDF, our method can be applicable to various software engineering datasets.

5 | Identifying on-hold self-admitted technical debt

Self-admitted technical debt refers to situations where a software developer knows that their current implementation is not optimal and indicates this using a source code comment. However, current research is largely focused on the detection and classification of self-admitted technical debt, but has spent less effort on approaches to address the debt automatically. In this chapter, we introduce new particular class of self-admitted technical debt of self-admitted technical debt amenable to automated management: on-hold SATD. We define on-hold SATD as self-admitted technical debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere and to propose tool support that can help developers manage self-admitted technical debt more effectively.

1 Introduction

The metaphor of technical debt is used to describe the trade-off many software developers face when developing software: how to balance near-term value with long-term quality [16]. Practitioners use the term technical debt as a synonym for "shortcut for expediency" [49] as well as to refer to bad code and inadequate refactoring [36]. Technical debt is widespread in the software domain and can cause increased software maintenance costs as well as decreased software quality [43].

In many cases, developers know when they are about to cause technical debt, and they leave documentation to indicate its presence [13]. This documentation often comes in the form of source code comments, such as "`TODO: This method is too complex, [let's] break it up`" and "`TODO no methods yet for getClassname`".[1] Previous work [28] has explored the use of visualization to support the discovery and removal of self-admitted technical debt, incorporating gamification mechanisms to motivate developers to contribute to the debt removal. Current research is largely focused on the detection and classification of self-admitted technical debt, but has spent less effort on approaches to address the debt automatically, likely because work on the detection and classification is still very recent.

Previous work [13] has developed an approach based on natural language processing to automatically detect self-admitted technical debt comments and to classify them into either design or requirement debt. Self-admitted design debt encompasses comments that indicate problems with the design of the code while self-admitted requirement debt includes all comments that convey the opinion of a developer suggesting that the implementation of a requirement is incomplete. In general terms, design debt can be resolved by refactoring whereas requirement debt indicates the need for new code.

In this chapter, we hypothesize that it is possible to use automated techniques based on natural language processing to understand a subset of the technical debt categories identified in previous work in more detail, and to propose tool support that can help developers manage self-admitted technical debt more effectively. We make three contributions:

- A qualitative study on the removal of self-admitted technical debt. To understand what kinds of technical debt could be addressed or managed automatically, we annotated a statistically representative sample of instances of self-admitted technical debt removal from the data set made available by the authors of previous work [48]. While the focus of our annotators was on the identification of instances of self-admitted technical debt that could be automatically addressed, as part of this annotation, we also performed a partial conceptual replication [72] of recent work by Zampetti et al. [93],[2]

[1]Examples from ArgoUML and Apache Ant, respectively [13].

[2]Note that Zampetti et al. [93] was published after we commenced this project, i.e., we do not use their data.

```
// TODO the following code is copied from AbstractSimpleBeanDefinitionParser
// it can be removed if ever the doParse() method is not final!
// or the Spring bug http://jira.springframework.org/browse/SPR-4599 is resolved
```

Figure 5.1. Motivating Example[3]

who found that a large percentage of self-admitted technical debt removals occur accidentally. We were able to confirm this finding: in 58% of the cases in our sample, the self-admitted technical debt was not actually addressed, but the admission was simply removed. This finding is also in line with findings from Bazrafshan and Koschke [6] who reported a large number of accidental removals of cloned code. Zampetti et al. [93] further reported that in removing self-admitted technical debt comments, developers tend to apply complex changes. Our work indirectly confirms this by finding that a majority of changes which address self-admitted technical debt could not easily be applied to similar debt in a different project.

- The definition of *on-hold self-admitted technical debt* (on-hold SATD). Our annotation revealed one particular class of self-admitted technical debt amenable to automated management: on-hold SATD. We define on-hold SATD as self-admitted technical debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere. Figure 5.1 shows an example of on-hold SATD from the Apache Camel project. The developer is waiting for an external event (the visibility of `doParse()` changing or an external bug being resolved) and the comment admitting the debt is therefore on hold.

- The design and evaluation of a classifier for self-admitted technical debt. Since software developers must keep track of many events and updates in any software ecosystem, it is unrealistic to assume that developers will be able to keep track of all self-admitted technical debt and of events that signal that certain self-admitted technical debt is now ready to be addressed. To

[3]cf. https://github.com/apache/camel/blob/53177d55053a42f6fd33434895c60615713f4b78/components/camel-spring/src/main/java/org/apache/camel/spring/handler/BeanDefinitionParser.java

support developers in managing self-admitted technical debt, we designed a classifier which can automatically identify those instances of self-admitted technical debt which are on hold, and detect the specific events that developers are waiting for. Our classifier achieves an area under the receiver operating characteristic curve (AUC) of 0.98 for the identification, and 90% of the specific conditions are detected correctly. This is a first step towards automated tool support that can recommend to developers when certain instances of self-admitted technical debt are ready to be addressed.

The remainder of this chapter is structured as follows: In Section 2, we present our research questions and the methods that we used for collecting and analyzing data for the qualitative study. The findings from this qualitative study are presented in Section 3. Section 4 describes the design of our classifier to identify on-hold SATD, and we present the results of our evaluation of the classifier in Section 5. Section 6 discusses the discussions of this work, before Section 7 highlights the threats to validity. Section 8 outlines the conclusions and highlights opportunities for future work.

2 Research Methodology

In this section, we detail our research questions as well as the methods for data collection and analysis used in our qualitative study. We also describe the data provided in our online appendix.

2.1 Research Questions

Our research questions focus on identifying how self-admitted technical debt is typically removed and whether the fixes applied to this debt could be applied to address similar debt in other projects. To guide our work, we first ask about the different kinds of self-admitted technical debt that can be found in our data (RQ1.1), whether the commits which remove the corresponding comments actually fix the debt (RQ1.2), and if so, what kind of fix has been applied (RQ1.3). To understand the removal in more detail, we also investigate whether the removal was the primary reason for the commit (RQ1.4), before investigating the subset of self-admitted technical debt that could be managed automatically (RQ1.5). Based on the definition of on-hold SATD which emerged from our qualitative study to answer these questions, we then investigate its prevalence (RQ1.6) and the accuracy of automated classifiers to identify this particular class of self-admitted technical debt (RQ2.1) and its specific sub-conditions (RQ2.2):

RQ1 How do developers remove self-admitted technical debt?

- **RQ1.1** What kinds of self-admitted technical debt do developers indicate?
- **RQ1.2** Do commits which remove the comments indicating self-admitted technical debt actually fix the debt?
- **RQ1.3** What kinds of fixes are applied to address self-admitted technical debt?
- **RQ1.4** Is the removal of self-admitted technical debt the primary reason for the commits which remove the corresponding comments?
- **RQ1.5** Could the fixes applied to address self-admitted technical debt be applied to address similar debt in other projects?
- **RQ1.6** How many of the comments indicating self-admitted technical debt contain a condition to specify that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere?

RQ2 How accurately can our classifier automatically identify on-hold SATD?

- **RQ2.1** What is the best performance of our classifier to automatically identify on-hold SATD?
- **RQ2.2** How well can our classifier automatically identify the specific conditions in on-hold SATD?

2.2 Data Collection

To obtain data on the removal of self-admitted technical debt, we used the online appendix of Maldonado et al. [48] as a starting point. In their work, Maldonado et al. conducted an empirical study on five open source projects to examine how

Table 5.1. Data set

project	SATD removal commits	sample
Apache Camel	987	128
Apache Tomcat	910	125
Apache Hadoop	370	52
Gerrit Code Review	133	19
Apache Log4j	107	9
Total	2,507	333

self-admitted technical debt is removed, who removes it, for how long it lives in a project, and what activities lead to its removal. They make their data available in an online appendix[4], which contains 2,599 instances of a commit removing self-admitted technical debt. After removing duplicates, 2,507 instances remain. The first two columns of Table 5.1 show the number of commits for each of the five projects available in this data set. Note that as a consequence of reusing this data set, we are implicitly also reusing Maldonado et al. [48]'s definition of technical debt as well as their interpretation of what constitutes debt removal.

Based on this data set of commits which removed a comment indicating self-admitted technical debt (after removing duplicates), we created a statistically representative and random sample (confidence level 95%, confidence interval 5 of 333 commits. The last column of Table 5.1 shows the number of commits from each project in our sample.

2.3 Data Analysis

To answer our first research question "How do developers remove self-admitted technical debt?" and its sub-questions, we performed a qualitative study on the sample of 333 commits which had removed self-admitted technical debt according to the data provided by Maldonado et al. [48].

In the first step, the second and third author of this chapter independently analyzed twenty commits from the sample to determine appropriate questions

[4]http://das.encs.concordia.ca/uploads/2017/07/maldonado_icsme2017.zip

Table 5.2. Qualitative annotation schema

question	answers	motivation
Does the comment represent self-admitted technical debt?	yes/no	Observation that some comments that Maldonado et al. [48] had automatically identified as self-admitted technical debt did not actually constitute debt
RQ1.1 What kind of self-admitted technical debt was it?	open	Distinguishing different kinds of self-admitted technical debt, with the ultimate goal of identifying ones that can be addressed automatically
RQ1.2 Did the commit fix the self-admitted technical debt?	yes/no	Observation that commits which remove self-admitted technical debt do not necessarily fix the debt, as also found by Zampetti et al. [93]
RQ1.3 What kind of fix was it?	open	Distinguishing different kinds of fixes for self-admitted technical debt, to study whether fixes could be applied automatically
RQ1.4 Was removing the self-admitted technical debt the primary reason for the commit?	yes/no	Observation that even for those commits which addressed self-admitted technical debt, this was not necessarily their main purpose
RQ1.5 Could the same fix be applied to similar self-admitted technical debt in a different project?	possibly/no	Identifying fixes that could potentially be applied automatically
RQ1.6 Does the self-admitted technical debt comment include a condition?	yes/no	Exploring the phenomenon of on-hold SATD—which emerged from answering the previous question—in more detail

to be asked during the qualitative study, aiming to obtain insights into how developers remove self-admitted technical debt and to identify the kinds of debt that could be addressed or managed automatically. After several iterations and meetings, the second and third author agreed on seven questions that should be answered for each of the 333 commits during the qualitative study. These questions along with their motivation and answer ranges are shown in Table 5.2.

The first author annotated all 333 commits following this annotation schema, and the second and third author annotated 50% of the data each, ensuring that each commit was annotated according to all seven questions by two researchers. Note that not all questions applied to all commits. For example, all instances which we classified as not representing self-admitted technical debt were not considered for future questions, and all commits which we classified as not fixing self-admitted technical debt were not considered for questions such as "Could the same fix be applied to similar Self-Admitted Technical Debt in a different project?".

After the annotation, the first three authors conducted multiple meetings in which they determined consistent coding schemes for the two questions which allowed for open answers and collaboratively resolved all disagreements in the annotation until reaching consensus on all ratings. We report the initial agreement for each question before the resolution of disagreements as part of our findings in the next section.[5]

2.4 Online Appendix

Our online appendix contains descriptive information on the 333 commits which were labeled as removing self-admitted technical debt according to Maldonado et al. [48] along with our qualitative annotations in response to the seven questions. Our online appendix also includes the data set we use in testing and training our classifier. The appendix is available at `https://tinyurl.com/onholddebt`.

[5]We calculated kappa values using `https://www.graphpad.com/quickcalcs/kappa1/`.

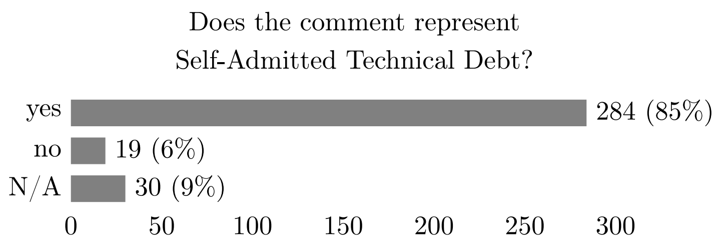


Figure 5.2. Distribution of answers to “Does the comment represent Self-Admitted Technical Debt?”. Initial agreement among the annotators before resolving disagreements: weighted kappa $\kappa = 0.820$ across 333 comments, i.e., “almost perfect” agreement [83].

3 Findings

In this section, we describe the findings derived from our qualitative study, separately for each sub-question of RQ1.

3.1 Initial Analysis

As shown in Figure 5.2, we found that not all commits which were automatically classified as removing self-admitted technical debt by the work of Maldonado et al. [48] actually removed a comment indicating debt. In some cases (9%)—indicated as N/A in Figure 5.2—the comment was not removed but only edited, and in other cases (6%), the comment had been incorrectly tagged as self-admitted technical debt, e.g., in the case of “`It is always a good idea to call this method when exiting an application`”.

3.2 RQ1.1 What kinds of self-admitted technical debt do developers indicate?

Our first research question explores the different kinds of self-admitted technical debt found in our sample. Figure 5.3 shows the final result of our coding after consolidating the coding schema. The two most common kinds of debt in our sample are “functionality needed” (44%) and “refactoring needed” (17%). An example for the former is the comment “`TODO handle known multi-value headers`” while

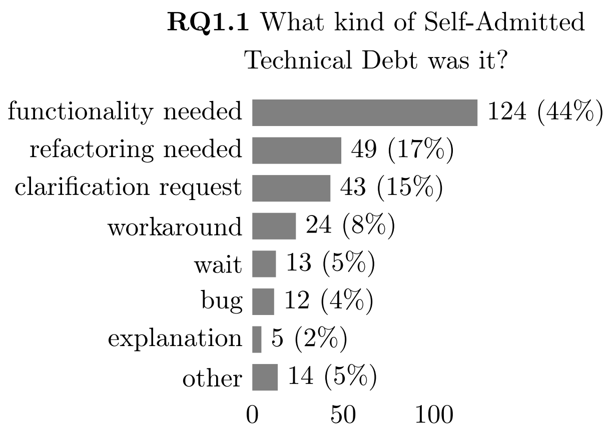


Figure 5.3. Distribution of answers to "What kind of Self-Admitted Technical Debt was it?". Initial agreement among the annotators before consolidating the coding schema: 45.07% across 284 comments.

"`XXX move message resources in this package`" is an example for the latter. We also identified a number of clarification requests (15%), such as "`TODO: why not use millis instead of nano?`". We coded self-admitted technical debt comments that explicitly stated that they were temporary as workaround (8%), e.g., "`TODO this should subtract resource just assigned TEMPROARY`". We identified some comments which indicated that the developer was waiting for something (5%), such as "`TODO remove these methods if/when they are available in the base class!!!`". We will focus our discussion on these comments in the later parts of this chapter. Finally, some comments which indicated technical debt describe bugs (4%, e.g., "`TODO this causes errors on shutdown...`") or focus on explaining the code (2%, e.g., "`some OS such as Windows can have problem doing rename IO operations so we may need to retry a couple of times to let it work`"). Note that for this annotation, we assigned exactly one code to each comment.

Previous classifications of self-admitted technical debt focused less on the actions required to remove the debt and more on what part of the software development lifecycle a debt item can be assigned to. For example, the categorisation of Maldonado and Shihab [47] revealed five categories (design, defect, documentation, requirement, and test), and the categorisation of Bavota and Russo [4]

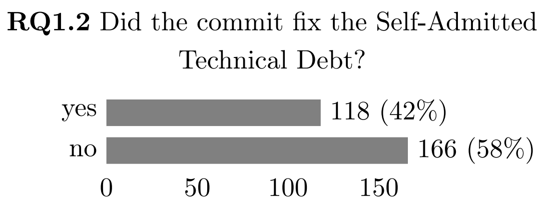


Figure 5.4. Distribution of answers to "Did the commit fix the Self-Admitted Technical Debt?". Initial agreement among the annotators before resolving disagreements: kappa $\kappa = 0.731$ across 284 comments, i.e., "substantial" agreement [83].

revealed the same five categories plus a sixth category called "code". In comparison, guided by our ultimate goal of identifying certain kinds of self-admitted technical debt which can be fixed automatically, our categorisation focuses more on what needs to be done in order to fix the debt, leading to categories such as "functionality needed" or "refactoring needed".

3.3 RQ1.2 Do commits which remove the comments indicating self-admitted technical debt actually fix the debt?

For the majority of commits (58%) which removed the comment indicating technical debt, the commit did not actually fix the problem described in the comment, see Figure 5.4. Instead, these commits often removed the comment along with the surrounding code. These findings are in line with recent work by Zampetti et al. [93] who found that between 20% and 50% of self-admitted technical debt is accidentally removed while entire classes or methods are dropped.

3.4 RQ1.3 What kinds of fixes are applied to address self-admitted technical debt?

In the cases where the commit fixed the self-admitted technical debt, we also coded the kind of fix that was applied. Figure 5.5 show the results of this coding: Debt was either fixed by implementing new code (58%), by refactoring existing

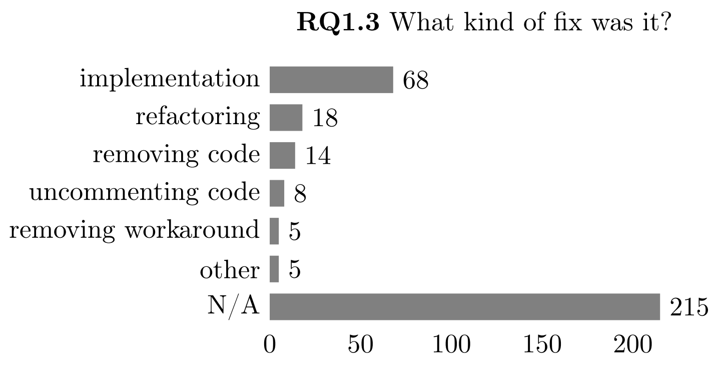


Figure 5.5. Distribution of answers to “What kind of fix was it?”. Initial agreement among the annotators before consolidating the coding schema: 83.90% across 118 comments.

code (15%), by removing code (12%), by uncommenting code that had been previously commented out (7%), or by removing a workaround (4%). Note that we used the commit message and/or related issue discussions to determine whether a change was meant to remove a workaround or was truly a refactoring. Other cases, such as uncommenting code, were easy to decide.

In the 215 cases where the commit does not fix the self-admitted technical debt, 30 commits do not remove the self-admitted technical debt comments or are tagged incorrectly, 19 comments do not represent self-admitted technical debt, and 166 commits do not fix self-admitted technical debt.

Our categorisation of the different kinds of fixes is at a slightly more coarse-granular level compared to that presented by Zampetti et al. [93] who identified five categories (add/remove method calls, add/remove conditionals, add/remove try-catch, modify method signature, and modify return) in addition to “other”. In their categorisation, “other” accounts for 44% (339/779) of all instances. In comparison, our categorisation is less fine-grained, but contains fewer “other” cases.

Table 5.3 shows the relationship between the two coding schemes that emerged from our qualitative data analysis: one for the kinds of technical debt indicated in developer comments, and one for the kinds of fixes applied to this debt. Unsurprisingly, many instances where new functionality was needed were addressed by the implementation of said functionality, and cases where refactoring was

Table 5.3. Types of Self-Admitted Technical Debt and the Corresponding Fixes. Each row represents a type of self-admitted technical debt, and each column represents a type of fix. The sum of each row and column indicates the overall numbers for the corresponding codes, respectively.

	implementation	refactoring	removing code	uncommenting code	removing workaround	other	not fixed
functionality needed	54	1	0	0	0	0	69
refactoring needed	2	16	1	0	0	1	29
clarification request	5	0	4	0	0	1	33
workaround	2	0	2	3	5	0	12
wait	2	0	2	3	0	0	6
bug	2	0	1	2	0	1	6
explanation	0	0	0	0	0	0	5
other	1	1	4	0	0	2	6

needed were addressed by refactoring. Interestingly, all comments of developers explaining technical debt were removed without addressing the debt. An example is the self-admitted technical debt comment "`some OS such as Windows can have problem doing delete IO operations so we may need to retry a couple of times to let it work`" in the Apache Camel project which was removed in commit f10f55e[6] together with the surrounding source code. We hypothesise that in some cases, developers decide to replace code which requires an explanation with simpler code. More work will have to be conducted to test this hypothesis. Waits could sometimes be addressed by uncommenting code that had been written in anticipation of the fix. A large number of comments indicating debt were not addressed—for example, out of 43 comments which we coded as clarification request, 33 (77%) were "resolved" by simply deleting the comment (e.g., the comment "`TODO why zero?`" was removed from the Apache Camel source code in commit 3d8f4e9[7] without further explanation. Note that in cases where more than one of our codes could apply, we noted the most prominent one. This could for example occur in cases of long comments which were used to communicate different concerns. In such rare cases, we applied the code for the longest section of the comment. This explains the small number of inconsistencies, e.g., a "functionality needed" debt fixed by a "refactoring".

3.5 RQ1.4 Is the removal of self-admitted technical debt the primary reason for the commits which remove the corresponding comments?

The removal of technical debt was often not the primary reason for commits which removed self-admitted debt, see Figure 5.6. We did not attempt to resolve disagreements between annotators for this question as the concept of "primary reason" can be ambiguous. Instead, instances where annotators disagreed are shown as "unclear" in Figure 5.6.

An example of a commit which removed self-admitted technical debt even

[6]https://github.com/apache/camel/commit/f10f55e38945686827dc249703b1606682657a62

[7]https://github.com/apache/camel/commit/3d8f4e9d68253269b4f5cf7e3cfea4553b46d74f

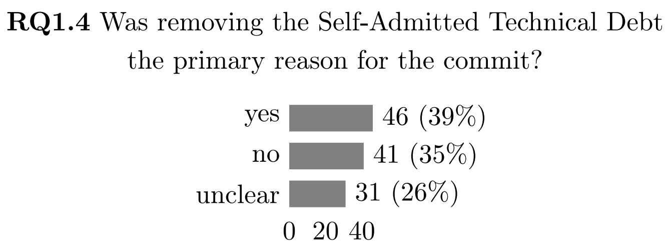


Figure 5.6. Distribution of answers to "Was removing the Self-Admitted Technical Debt the primary reason for the commit?". Agreement among the annotators: weighted kappa $\kappa = 0.630$ across 118 comments, i.e., "substantial" agreement [83].

though it was not the main purpose of the commit is Apache Camel commit f47adf.[8] The commit removed the following comment: "`TODO: Support ordering of interceptors`", but this was part of a much larger refactoring as described in the commit message: "`Overhaul of JMX`". On the other hand, the commit message of commit 88ca35[9] from the same project "`Added onException support to DefaultErrorHandler`" is very similar to the self-admitted technical debt comment that was removed in this commit "`TODO: in the future support onException`", which suggests that removing the debt was the primary reason for this commit.

3.6 RQ1.5 Could the fixes applied to address self-admitted technical debt be applied to address similar debt in other projects?

We annotated the 118 self-admitted technical debt comments which had been fixed by a commit in terms of whether the fix applied in this commit could be applied in a similar context in a different project. While this annotation was subjective to some extent—as also indicated by our kappa agreement of 0.540 which was the lowest across all questions we answered about the self-admitted

[8]https://github.com/apache/camel/commit/f47adf75510ef71a5b4071e8c77af7abb9c07dc9

[9]https://github.com/apache/camel/commit/88ca359343c3a96786d435985f46841eeffcfb6e

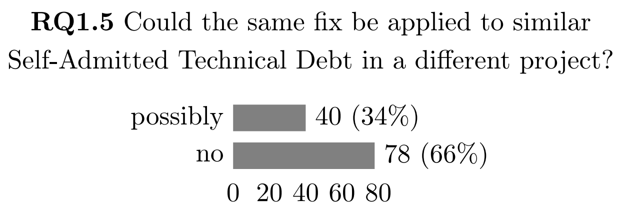


Figure 5.7. Distribution of answers to "Could the same fix be applied to similar Self-Admitted Technical Debt in a different project?". Agreement among the annotators: kappa $\kappa = 0.540$ across 118 comments, i.e., "moderate" agreement [83].

technical debt comments—we used our intuition about whether we could envision tool support to address a comment automatically. We used our experience of conducting research on automated tool support for source code manipulation for this step.

We identified two kinds of self-admitted technical debt that could possibly be handled automatically. The first kind are comments which are fairly specific, e.g., "`TODO gotta catch RejectedExecutionException and properly handle it`". Automated tool support could be built to at least catch the exception based on this description. The second kind are comments which indicate that a developer is waiting for something, which we will discuss further in the next subsection. Figure 5.7 shows the ratio of fixes that could possibly be automated and applied in other settings, which is one third of all fixes. Note that we counted all those comments as "possibly" that were rated as "possibly" by at least one annotator. This finding supports Zampetti et al. [93] who found that most changes addressing self-admitted technical debt require complex source code changes. The primary goal of investigating this research question was the identification of types of self-admitted technical debt likely amenable to being fixed automatically.

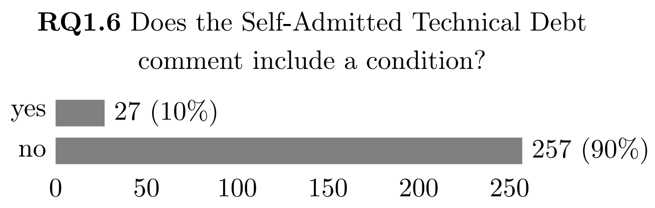


Figure 5.8. Distribution of answers to "Does the Self-Admitted Technical Debt comment include a condition?". Initial agreement among the annotators before resolving disagreements: weighted kappa $\kappa = 0.618$ across 284 comments, i.e., "substantial" agreement [83].

Table 5.4. Example of self-admitted technical debt on "on-hold" and "wait"

Example of SATD	**Category / On-hold or not**
// TODO change to file when this is ready	wait / non on-hold
// FIXME: Code to be used in case route replacement is needed	wait / non on-hold
// TODO: is needed when we add support for when predicate	add functionality / on-hold
// TODO: Camel 2.9/3.0 consider moving to org.apache.camel	refactor / on-hold

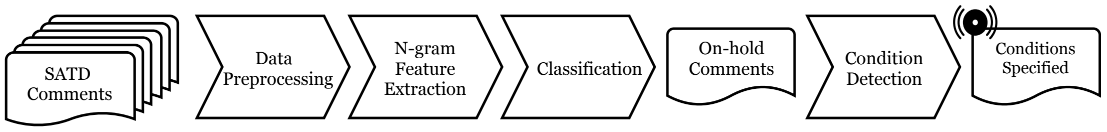


Figure 5.9. Classification overview.

3.7 RQ1.6 How many of the comments indicating self-admitted technical debt contain a condition to specify that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere?

A theme that emerged from answering the previous research question is the concept of self-admitted technical debt comments which include a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere. Since no other obvious class of self-admitted technical debt emerged which seemed amenable to automated tool support, we focus on this kind of self-admitted technical debt for building a classifier (see next section). We refer to this kind of debt as on-hold SATD—the comment is on hold until the condition is met (see Figure 5.1 for examples). In our sample, we identified 27 such comments, see Figure 5.8. These comments are also related to the "wait" category shown in Figure 5.3, but not necessarily identical since the question addressed by Figure 5.3 did not explicitly ask about conditions. Table 5.4 shows examples of "on-hold" comments and those classified in the "wait" category.

4 Design

Figure 5.9 shows the overview of our classifier for on-hold SATD identification and the detection of the specific conditions that developers are waiting for. Given self-admitted technical debt comments, data preprocessing and n-gram feature extraction are applied before classifying them into on-hold or not. Within identified on-hold SATD comments, specific conditions are detected.

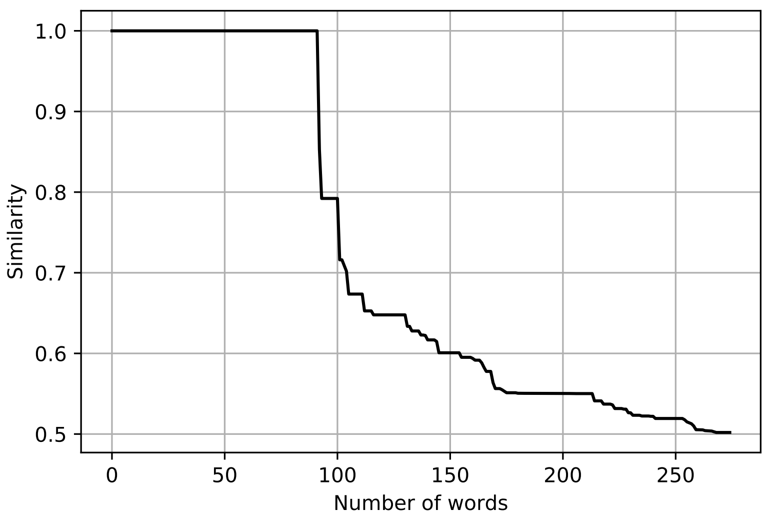


Figure 5.10. Similarity between project names and words.

4.1 Data Preprocessing

Three preprocessing steps are applied, namely, term abstraction, lemmatization, and special character removal.

Term Abstraction. Similar to a previous text classification study [59], we perform abstraction as a preprocessing step. The previous study [59] abstracted keywords from GitHub README files. Their abstraction included mail-to links, hyperlinks, code blocks, images, and numbers. We also apply abstraction for hyperlinks (URLs), however, we do not apply the others because images, mail-to links, and code blocks do not usually appear in comments. Instead, we introduce four kinds of abstraction which are related to on-hold conditions. We target the following terms: *date expression*, *version*, *bug id*, *URL*, and *product name*. Each term is abstracted into a string: `@abstractdate`, `@abstractversion`, `@abstractbugid`, `@abstracturl`, and `@abstractproduct`. Table 5.5 shows the regular expressions we use to detect `@abstractdate`, `@abstractversion`, `@abstractbugid`, and `@abstracturl`.

For abstracting product names for `@abstractproduct`, we try finding semantically similar words to the project names and their sub-project names in our data set, i.e., Apache, Camel, Tomcat, Hadoop, Gerrit, Log4j, Yarn, Mapreduce, Hdfs, Ant, Jmeter, ArgoUML, Columba, Emf, Hibernate, Distribution, JEdit, JFreechart, JRuby, and SQuirrel. Figure 5.10 shows the similarity between each word in comments and project name and their related project using Spacy [25].[10] According to the result, the similarity score drops drastically from 1.0—therefore, we consider words with similarity 1.0 as project names. We obtained 77 words.

We apply this process because we are more interested in the existence of these types rather than the actual terms, which do not appear frequently. For example, considering the comment "`TODO: CAMEL-1475 should fix this`", CAMEL-1475 will be changed to the string "`@abstractproduct @abstractbugid`". Table 5.5 summarizes the regular expressions we used for identifying targeted terms. Replacements using the regular expressions are conducted from top to bottom in the table. Subsequently, URLs linking to specific ids of bugs are abstracted to "`@abstracturl @abstractbugid`".

Lemmatization. Lemmatization is a process to reduce the inflection form of words into dictionary form by considering the context in the sentences. This process is applied to increase the frequency of words appearing by changing words into dictionary forms using tools from Spacy [25].

Special character removal. Since non-English characters and non-numeric ones do not represent words, we use the regular expression `[^A-Za-z0-9]+` to remove them. Stop word removal is not applied in this work because a stop word list contains important keywords for identifying on-hold SATD (e.g., when, until). We use Spacy to apply lemmatization which will change words into their dictionary form. However, some single characters will appear, e.g., when lemmatising "// TODO: Removed from UML 2.x" to "todo remove from uml 2 x".

[10]Spacy has recently been found to achieve a higher accuracy when applied to software-related text compared to other libraries [2].

4.2 N-gram Feature Extraction

We extract n-gram term features by applying N-gram IDF [68, 71]. Inverse Document Frequency (IDF) has been widely used in many applications because of its simplicity and robustness; however, IDF cannot handle phrases (i.e., groups of more than one term). Because IDF gives more weight to terms occurring in fewer documents, rare phrases are assigned more weight than good phrases that would be useful in text classification. N-gram IDF is a theoretical extension of IDF for handing multiple terms and phrases by bridging the theoretical gap between term weighting and multi-word expression extraction [68, 71].

Terdchanakul et al. [80] reported that for classifying bug reports into bugs or non-bugs, classification models using features from N-gram IDF outperform models using topic modeling features. In addition to this, we consider that n-gram word features are beneficial for comment classification rather than topic modeling because source code comments are generally short and contain only a small number of words.

Wattanakriengkrai et al. [85] created classification models to identify design and requirement self-admitted technical debt using source code comments. By using N-gram IDF and auto-sklearn automated machine learning, classification models outperform models with single word features.

In this study, we use an N-gram Weighting Scheme tool [65], which uses an enhanced suffix array [1] to enumerate valid n-grams. We obtain a list of all valid n-grams that contain at most 10 terms from the on-hold self-admitted technical debt comments and remove n-grams which have frequency equal to one. We obtain about one thousand two hundred n-gram terms from our 267 on-hold self-admitted technical debt comments. After that, we apply Auto-sklearn's feature selection. Auto-sklearn includes two feature selection functions from the sklearn library, sklearn.feature_selection.GenericUnivariateSelect (Univariate feature selector) and sklearn.feature_selection.SelectPercentile (Select features according to percentile). Calling these functions is part of Auto-sklearn's feature preprocessing—it selects suitable feature processing based on meta-learning automatically.

4.3 Classifier Learning

Given the set of n-gram term features from the previous step, we build a classifier that can identify on-hold SATD by classifying self-admitted technical debt comments into on-hold or not.

In machine learning, two problems are known: (1) no single machine learning method performs best on all data sets, and (2) some machine learning methods rely heavily on hyperparameter optimization. Automated machine learning aims to optimize choosing a good algorithm and feature preprocessing steps [18]. To obtain the best performance (RQ2.1), similar to Wattanakriengkrai et al. [85]'s work, we apply auto-sklearn [18], a tool of automated machine learning.

Auto-sklearn addresses these problems as a joint optimization problem [18]. Auto-sklearn includes 15 base classification algorithms, and produces results from an ensemble of classifiers derived by Bayesian optimization [18].

For classifier learning, we prepare feature vectors with N-gram TF-IDF scores of all n-gram terms. The score is calculated with the following formula:

$$\textit{n-gram TF-IDF} = log(\frac{|D|}{sdf}) * gtf$$

where $|D|$ is the total number of comments, sdf is the document frequency of a set of terms composing an n-gram, and gtf is the global term frequency.

4.4 On-hold Condition Detection

After on-hold SATD comments are identified, we try to identify their on-hold conditions. During our annotation, we found conditions of self-admitted technical debt that are related to waiting for a bug to be fixed, a release of a library, or a new version of a library.

- For a bug to be fixed, we abstract the bug report number. In a bug report tracking system, the bug report number is created by using the project name and report number which we abstract using the keywords @abstractproduct and @abstractbugid.

- For release date, we abstract it using the keyword @abstractdate.

- For a new version of a library, the library version usually appears in a project name and release version (e.g., 1.9.3, 4.0), which we abstract using the keywords @abstractproduct and @abstractversion.

As we have already replaced these terms with specific keywords shown in Table 5.5, we can derive conditions by recovering the original terms. The following is our detection process.

1. Extract keywords of `@abstractdate`, `@abstractversion`, `@abstractbugid`, and `@abstractproduct` by preserving the order of appearance in the identified on-hold SATD comments.

2. Group keywords to make valid conditions. Only the following sets of keywords are considered to be valid conditions, and other keywords that do not match the following orders are ignored.

    - `{@abstractdate}`: an individual date expression.
    - `{@abstractproduct, @abstractversion, ...}`: a product name followed by one or more version expressions, to indicate specific versions of the product.
    - `{@abstractproduct, @abstractbugid, ...}`: a product name followed by one or more bug ID expressions, to indicate specific bugs of the product.

Identifying these keywords as conditions is not trivial, because they also frequently appear in comments that do not indicate on-hold SATD. Since we limit this detection to the identified on-hold comments, we expect that this simple process can work.

5 Evaluation

In this section, we describe the steps we took to evaluate our classifier.

5.1 Data Preparation and Annotation

As shown in Figure 5.8, we found fewer than 30 on-hold SATD comments in the sample of 333 comments. Since it is difficult to train classifiers on such a small number of instances, we investigated all 2,507 comments again to prepare data for our classification. After that, the first and third author separately annotated the remaining comments in terms of (i) whether comments represent self-admitted technical debt (similar to Figure 5.2) and (ii) whether the self-admitted technical debt comments include a condition (similar to Figure 5.8). All conflicts in this annotation were resolved by the second author. Note that we decided to train the classifier on comments which had been removed through the resolution of self-admitted technical debt to ensure we were able to consider the entire lifecycle of the self-admitted technical comment before deciding whether to consider it on-hold.

We also include a data set from ten open source projects introduced by da Silva Maldonado et al. [13]. First, we randomly selected a sample of 30 comments out of all 3,299 comments. The first author, third author, and an external annotator annotated these comments, resulting in 97.78% overall agreement, i.e., "almost perfect" according to Viera and Garrett [83]. Then the first author annotated the remaining comments.

Tables 5.6 and 5.7 show the result of this data preparation. From 5,806 comments, 333 comments are samples of removed self-admitted technical debt and 225 comments that do not represent self-admitted technical debt are excluded. We obtained 267 on-hold comments and 4,981 other comments, which are used for our classification. After excluding duplicate comments, our dataset contains a total of 5,248 comments. Before exclusion, 410 comments were duplicates which can be grouped into 168 sets of comments. Among these 168 sets of duplicates, 11 sets (24 comments) are on-hold comments, and 157 sets (386 comments) are non on-hold comments.

5.2 Evaluation Settings

We measure the classification performance in terms of precision, recall, F_1, and AUC. AUC is the area under the receiver operating characteristic curve. The receiver operating characteristic curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.

$$Precision = \frac{tp}{tp + fp}$$
$$Recall = \frac{tp}{tp + fn}$$
$$F_1 = \frac{2 \cdot (precision \cdot recall)}{(precision + recall)}$$
$$TPR = \frac{tp}{tp + fn}$$
$$FPR = \frac{fp}{tn + fp}$$

where tp is the number of true positives, tn is the number of true negatives, fp is the number of false positives, and fn is the number of false negatives.

Comparison. A Naive Baseline is created based on the assumption that it is also possible to find on-hold technical debt comments while using basic searching similar to the `grep` command. The words we use for searching are selected from the top 30 words that appear frequently in comments. We manually classify words to select those that relate to on-hold technical debt. The words we selected are *"should"*, *"when"*, *"once"*, *"remove"*, *"workaround"*, *"fixed"*, *"after"*, and *"will"*.

To assess the effectiveness of n-gram features in classifying on-hold SATD comments, we compare the performances of classifiers using N-gram TF-IDF and traditional TF-IDF [63]. Except for feature extraction, the two classifiers are prepared using the same settings including term abstraction.

Ten-fold cross-validation. Ten-fold cross-validation divides the data into ten sets and every set is used as test set once while the others are used for training. Due to the imbalance between the number of positive and negative instances, we use the Stratified ShuffleSplit cross validator of scikit-learn made available by Pedregosa et al. [55], which intends to preserve the percentage of samples from each class. Because of this process, some instances can appear multiple times in different sets. Therefore we report the mean values of the evaluation metrics across all ten runs as the performance.

To measure the effect of rebalancing on our classification, we compare the performance of N-gram TF-IDF with and without Stratified ShuffleSplit.

5.3 RQ2.1 What is the best performance of a classifier to automatically identify on-hold SATD?

As shown in Table 5.8, our classifier with n-gram TF-IDF achieved a mean precision of 0.75, a mean recall of 0.78, a mean F_1-score of 0.77, and a mean AUC of 0.98. N-gram TF-IDF has the best performance in every evaluation except precision which has a similar score with N-gram TF-IDF without rebalancing. We consider that both precision and recall are essential for this kind of recommendation system. Precision is important since false positives (i.e., unwarranted recommendations) will annoy developers. However, recall is still important since false negatives (i.e., recommendations that the system could have made but did not) might cause problems since developer will be unaware of important information. Table 5.9 shows the top features from N-gram TF-IDF ranked by how frequently our classifier uses them to distinguish on-hold comments from other self-admitted technical debt comments.

We also run an experiment for both cross-project classification and within-project classification on projects for which the ratio of on-hold SATD comments among all self-admitted technical debt comments is more than 2%. For cross-project classification, we divide data into sets according to their project. Every set is used as test set once while the other sets are used for training. Table 5.10 shows the results for each project. On average, our classifier with cross-project classification achieved a mean precision of 0.46, a mean recall of 0.52, a mean F_1-score of 0.45, and a mean AUC of 0.93.

For within-project classification, in each project, we apply a ten-fold classification with the Stratified ShuffleSplit cross validator. Table 5.11 shows the result for each project. Among them, five projects (Camel, Hadoop, Tomcat, ArgoUML, and JRuby) have a similar score or higher compared to cross-project classification according to all metrics. Another four projects (Ant, Gerrit, JEdit, and SQuirrel) have a lower score compared to cross-project classification according to all metrics. The difference between these two groups is that the first group has the

```
/*
* TODO: After YARN-2 is committed, we should call containerResource.getCpus()
* (or equivalent) to multiply the weight by the number of requested cpus.
*/
```

Figure 5.11. On-hold SATD example which we correctly identify[11]

number of on-hold comments >= 20 while the second group has the number of on-hold comments < 10.

5.4 RQ2.2 How well can our classifier automatically identify the specific conditions in on-hold SATD?

Because of our treatment of imbalanced data (see Section 5.3), some comments can appear multiple times in the test set. We consider that an on-hold comment is correctly identified only if it has been classified correctly in all cases where it was part of the test set. Our classifier was able to identify 219 out of 267 on-hold comments correctly. Among them, 94 comments contain abstraction keywords which indicate a specific condition, and all those instances were confirmed to be specific conditions by manual investigation. Table 5.12 shows examples of on-hold comments and their specific conditions. However, some comments do not mention specific conditions, such as "`This crap is required to work around a bug in hibernate`". Among the 48 false positives (incorrectly identified comments), 10 comments contain abstraction keywords, but these keywords are used for references and not for conditions that a developer is waiting for. In summary, 90% (94/(10+94)) of the detected specific conditions are correct, and for 43% (94/219) of the on-hold comments, we were able to identify the specific condition that a developer was waiting for.

[11]cf. `https://github.com/apache/hadoop/commit/80eb92aff02cc9f899a6897e9cbc2bc69bd56136/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-nodemanager/src/main/java/org/apache/hadoop/yarn/server/nodemanager/util/CgroupsLCEResourcesHandler.java`

[12]cf. `https://github.com/gerrit-review/gerrit/commit/0485172aaa70e3b1f0e98c00215672657e6f462e/gerrit-gwtui/src/main/java/com/google/gerrit/client/diff/CodeMirrorDemo.java`

```
/**
 * Ugly workaround because CodeMirror never hides lines completely.
 * TODO: Change to use CodeMirror's official workaround after
 * updating the library to latest HEAD.
 */
```

Figure 5.12. On-hold SATD example which our classifier cannot identify[12]

Figure 5.11 shows an example of an on-hold SATD comment. Our model can identify conditions using the keywords @abstractproduct and @abstractbugid referring to YARN-2. Figure 5.12 shows an example that our classifier could not identify correctly. The on-hold condition refers to a workaround waiting for an update to the CodeMirror library.

5.5 Developer Feedback

To evaluate whether our approach for detecting on-hold SATD could be useful in practice, we ran the cross-project classifier on the source code of the open-source project JabRef, a graphical Java application for managing BibTeX and biblatex (.bib) databases.[13] We used source code comments containing SATD keywords from Huang et al. [26] (Table 1 and Table 16) and an abstractkeyword as input and classified the resulting data into on-hold and not on-hold. From the classification result, we obtained a total of 22 potential on-hold comments. A manual analysis revealed that 19 cases were not actually SATD and that 3 cases are on-hold SATD. Note that our classifier was developed to classify SATD comments into on-hold or not, and not to determine whether any comment is SATD—this has been done in previous work (Maldonado and Shihab [47]). We then sent three instances of on-hold SATD to one of JabRef's core developers. Table 5.13 shows these comments along with the explanation we sent to the developer. In addition, for each on-hold SATD comment, we included a link to the exact line of code from which we had extracted the comment.

Regarding the first comment, the developer pointed out that the comment had been deleted in the meantime, but noted

[13]https://github.com/JabRef/jabref/

As a final check, it would be helpful though.

Regarding the second comment, the developer mentioned the potential overlap between notifications that our approach could produce and other notifications that the developer would already receive anyway:

> *Maybe, here an active 'Comment Checking' would be more helpful. Then remembering if the comment should be kept - so that an additional scan with the same setting does not trigger a notification again. - For me, getting notified because of new comments additionally, would not be helpful as I would have been notified of GitHub.*

The third comment turned out to be the most useful one, in the developer's perception:

> *In this case, a bot posting a message to the issue with following text would have been helpful: 'I found following references to this closed issue in the code. Maybe, the code has to be adapted, too?'*

In response to our final question "Do you think such a tool could be useful?", the developer responded

> *Since the last example was really useful, you hear me saying: "Yes". :-).*

6 Discussions

The ultimate goal of our work is to enable the automated management of certain kinds of self-admitted technical debt. Previous work [93] has found that most changes which address self-admitted technical debt require complex code changes—as such, it is unrealistic to assume that automated tool support could handle all kinds of requirement debt and design debt that developers admit in source code comments. Thus, in this work we set out to first identify a sub-class of self-admitted technical debt amenable to automated management and second develop a classifier which can reliably identify this sub-class of debt.

Our qualitative study revealed one particular class of self-admitted technical debt potentially amendable to automated tooling: on-hold SATD, i.e., comments in which developers express that they are waiting for a certain external event or updated functionality from an external library before they can address the debt that is expressed in the comment. In other words, the comment is on hold until the condition has been met.

Based on the data set made available by Maldonado et al. [48] and da Silva Maldonado et al. [13], we identified a total of 293 comments which indicate on-hold SATD, confirming that this phenomenon is prevalent and exists in different projects. Our classifier to identify on-hold SATD was able to reach an AUC of 0.98 in identifying comments that belong to this sub-class. In addition, we were able to identify specific conditions contained within these comments (90% of conditions are detected correctly). Based on 15 projects, there are 293 on-hold comments out of 5,529 self-admitted technical debt comments, resulting in a relative frequency of 5.30%. Out of 15 projects, the ratio of on-hold comments compared to all self-admitted technical debt comments is larger than 2% for nine projects.

Given all the events and new releases that happen in a software project at any given point in time, it is unrealistic to assume that developers will be able to stay on top of all instances of technical debt that are ready to be addressed once a condition has been met. Instead, there is a risk that developers forget to go back to these comments and debt instances even when the event they were originally waiting for has occurred.

This work builds a first step towards the design of automated tools that can support developers in addressing certain kinds of self-admitted technical debt. In particular, based on the classifier introduced in this work, it is now possible to build tool support which can monitor the specific external events we have identified in this work (e.g., certain bug fixes or the release of new versions of external libraries) and notify developers as soon as a particular debt is ready to be addressed. While the ratio of on-hold comments is fairly low, such comments appeared in almost all of the studied projects, and we argue that alerting developers when such comments are ready to be addressed can prevent bugs or vulnerabilities that might otherwise occur, e.g., because of outdated libraries.

In terms of tool support, we envision a tool which supports the developer by indicating comments that are ready to be addressed rather than a tool which addresses comments automatically. Addressing comments automatically—even though it is an interesting research challenge—is problematic for two reasons: (1) the precision of such a tool would have to be really high, and current work including our own suggests that this is not yet the case; and (2) developers are unlikely to relinquish control over their code base to a tool which automatically changes code.

7 Threats to Validity

Regarding threats to *internal validity*, it is possible that we introduced bias through our manual annotation. While we generally achieved high agreement regarding the annotation questions listed in Table 5.2, the initial agreement regarding RQ1.1 was low which is explained by the nature of the open-ended question. We resolved all disagreements through multiple co-located coding sessions with the first three authors of this chapter. Note that we do not use the results of RQ1.1 as an input for our classifier. We may possibly have wrongly classified the removal of self-admitted technical debt, since in particular for comments indicating the need for new features, it can be hard to judge whether the new feature was indeed fully implemented. Another concern is that we did not manually validate the entire data set that we are reusing from previous work. There might be further quality issues with this which would affect the performance of our approach, such as self-admitted technical debt being identified in the header of classes.

For *external validity*, while we analyzed a statistically representative sample of commits for RQ1 and the entire data set made available by Maldonado et al. [48] (after removing duplicates) for RQ2, we cannot claim generalizablity beyond the projects contained in this data set and our classifier might be biased as a result of the small number of projects. The limited data set allowed us to perform an in-depth qualitative analysis, and future work will need to investigate the applicability of our results to other projects and within-project prediction.

For *construct validity*, this is related to the manual labeling of on-hold SATD.

A label might be affected by annotator misunderstand or mislabeling. Despite annotators resolving disagreements through discussion, the labels might still be incorrect.

8 Conclusions and Future Work

Self-admitted technical debt refers to situations in which software developers explicitly admit to introducing technical debt in source code comments, arguably to make sure that this debt is not forgotten and that somebody will be able to go back later to address this debt. In this work, we hypothesize that it is possible to develop automated techniques to manage a subset of self-admitted technical debt.

As a first step towards automating a part of the management of certain kinds of self-admitted technical debt, in this chapter, we contribute (i) a qualitative study on the removal of self-admitted technical debt in which we annotated a statistically representative sample of 333 technical debt comments using seven questions that emerged as part of the qualitative analysis; (ii) on-hold SATD (debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere) which emerged from this qualitative analysis as a particular class of self-admitted technical debt that can potentially be managed automatically; and (iii) the design and evaluation of a classifier for self-admitted technical debt which can detect on-hold debt with an AUC of 0.98 as well as identify the specific conditions that developers are waiting for.

Building on these contributions, in our future work we intend to build the tool support that our classifier enables: a recommender system which can indicate for a subset of self-admitted technical debt in a project when it is ready to be addressed. We found that self-admitted technical debt is sometimes addressed by uncommenting source code that has already been written in anticipation of the debt removal. As another step towards the automation of technical debt removal, in future work, we will explore whether it is possible to address such debt automatically.

Table 5.5. Regular expressions for term abstraction

abstraction	**pattern**

@abstractdate	`(0[1-9]|[12]\d|3[01]).(0[1-9]|1[0-2]).([12]\d3)`
	# year.month.date, e.g., 21.02.2011
	`(0[1-9]|[12]\d|3[01])\/(0[1-9]|1[0-2])(\/([12]\d3))*`
	# day/month(/year), e.g., 25/05, 22/05/2012
	`((([0-9])|([0-2][0-9])|([3][0-1]))(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\w+ \d4`
	# day month year, e.g., 23 June 2013
	`\d+-\d+-\d+ \d+:\d+:\d+ [-|+]\d+`
	# year-month-day timestamp, e.g., 2006-03-06 23:16:24 +0100

@abstractversion	`[0-9]{1,2}\.[0-9]{1,2}([+-]|\.[0-9]{1,3}|\.[A-Za-z]{1,2})*(_[0-9]{1,3})*` # release version, e.g., 1.9.3, 4.0, 8.0.x, 1.0.12_25
@abstractbugid	`abstractproduct[ |-]*\d+` # bug id, e.g., jetty-9.3
@abstracturl	`https?:\/\/(www\.)?[-a-zA-Z0-9@:%._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#?&//=]*)` # url

Table 5.6. Annotated self-admitted technical debt comments

	characteristic	**number**
excluded	not self-admitted technical debt	225
	sample of removed self-admitted technical debt	333
classification data	with condition (on-hold)	267
	without condition	4,981
sum		5,806

Table 5.7. Number of on-hold SATD comments in each project

project	**number**	**example**
Apache Camel	88	// @deprecated will be removed on Camel 2.0 ...
Apache Tomcat	20	// TODO This can be fixed in Java 6 ...
Apache Hadoop	20	// TODO need to get the real port number MAPREDUCE-2666
Gerrit Code Review	6	// TODO: remove this code when Guice fixes its issue 745
Apache Log4j	1	// TODO: this method should be removed if OptionConverter becomes a static
Apache Ant	7	// since Java 1.4 ... // workaround for Java 1.2-1.3
Apache Jmeter	2	// TODO this bit of code needs to be tidied up ... Bug 47165
ArgoUML	77	// TODO: gone in UML 2.1
Columba	0	–
EMF	1	// Note: Registry based authority is being removed ... which would obsolete RFC 2396. If the spec is added ... needs to be removed.
Hibernate	5	// FIXME Hacky workaround to JBCACHE-1202
JEdit	6	// undocumented hack to allow browser actions to work. // XXX - clean up in 4.3
JFreeChart	2	// TODO: In JFreeChart 1.2.0 ...
JRuby	23	// Workaround for JRUBY-4149
SQuirrel	9	// We know this fails - Bug# 1700093
total	267	–

Table 5.8. Performance comparison

	naive baseline	**TF-IDF**	**N-gram TF-IDF**	**N-gram TF-IDF without rebalancing**
Precision	0.12	0.73	0.75	**0.76**
Recall	0.66	0.60	**0.78**	0.77
F_1	0.20	0.66	**0.77**	0.76
AUC	0.70	0.97	**0.98**	**0.98**

Table 5.9. Top 10 N-gram TF-IDF frequent features only appear in on-hold comments.

N-gram Features	**frequency**
‘remove’, ‘in’, ‘abstractproduct’, ‘abstractversion’	7
‘in’, ‘uml’, ’2’, ‘x’	7
‘fix’, ‘in’	6
‘workaround’, ‘to’	6
‘todo’, ‘cmueller’, ‘remove’, ‘the’	6
‘ref’, ‘attribute’	6
‘be’, ‘remove’, ‘in’, ‘abstractproduct’, ‘abstractversion’	6
‘for’, ‘abstractversion’	5
‘after’, ‘abstractproduct’, ‘abstractbugid’	5
‘workaround’, ‘for’, ‘abstractproduct’, ‘abstractbugid’	4

Table 5.10. Cross-project classification on projects which contain on-hold more than 2%

Project	**#, (% of on-hold)**	**Precision**	**Recall**	**F1**	**AUC**
Apache Ant	7, (5.6%)	0.50	0.57	0.53	0.97
Apache Camel	88, (10.9%)	0.81	0.39	0.52	0.96
Apache Hadoop	20, (8.2%)	0.61	0.85	0.71	0.96
Apache Tomcat	20, (2.8%)	0.19	0.50	0.27	0.92
ArgoUML	77, (6.7%)	0.36	0.17	0.23	0.82
Gerrit Code Review	6, (6.3%)	0.60	0.50	0.55	0.91
JEdit	6, (2.6%)	0.33	0.67	0.44	0.91
JRuby	23, (5.0%)	0.46	0.57	0.51	0.97
SQuirrel	9, (3.9%)	0.24	0.44	0.31	0.91
Average	-	0.46	0.52	0.45	0.93

Table 5.11. Within-project classification on projects which contain on-hold more than 2%

Project	**#, (% of on-hold)**	**Precision**	**Recall**	**F1**	**AUC**
Apache Ant	7, (5.6%)	0.19	0.80	0.30	0.86
Apache Camel	88, (10.9%)	0.91	0.61	0.73	0.99
Apache Hadoop	20, (8.2%)	0.85	0.80	0.79	0.99
Apache Tomcat	20, (2.8%)	0.73	0.65	0.66	0.99
ArgoUML	77, (6.7%)	0.66	0.79	0.71	0.98
Gerrit Code Review	6, (6.3%)	0.15	0.30	0.20	0.86
JEdit	6, (2.6%)	0.11	0.80	0.18	0.90
JRuby	23, (5.0%)	0.70	0.80	0.70	0.99
SQuirrel	9, (3.9%)	0.15	1.00	0.25	0.89
Average	-	0.49	0.73	0.50	0.94

Table 5.12. Examples of specific conditions in on-hold SATD comments

specific condition	**example of on-hold SATD comments**
@abstractdate	// Workaround for, Adobe Read 9 plug-in on IE bug // Can be removed after **26 June 2013**
@abstractproduct, @abstractversion	// TODO cmueller:, remove the “httpBindingRef” look up in **Camel 3.0**
@abstractproduct, @abstractbugid	// FIXME (**CAMEL-3091**): @Test

Table 5.13. On-hold SATD sent for developer feedback

#	on-hold SATD	explanation
1	“... todo: reenable customize entry types feature (<link to issue 4719>) ...”	we could notify developers once issue 4719 has been closed
2	“... we must not clean the url. this is the deal with @manastungare - see <link to comment on issue 684> ...”	we could have notified developers once issue 684 was closed and/or if there have been responses to the comment
3	“... - handling of identically fields with different names (<link to issue 521>) ...”	we could have notified developers once issue 521 was closed

6 Automated identification of on-hold self-admitted technical debt

In the previous chapter, we introduce new particular class of self-admitted technical debt of self-admitted technical debt amenable to automated management: on-hold SATD. Hence, in this chapter, an empirical study is performed to identify cases in which the On-hold SATD should be removed when the waiting condition has been fulfilled.

1 Introduction

Technical debt (TD) was first mentioned as a concept by Cunningham close to 30 years ago Cunningham [9], when he wrote the following lines: *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly [...] The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt."*

In simple words, TD is a short-term "hack" (often induced by industrial reality, which dictates that either time and/or money are short) with long-lasting consequences if not properly handled. Since developers naturally keep working on new parts and do not revisit something unless it is strictly necessary, very often TD results, in the long run, in low maintainability and poor performance

Lim et al. [41].

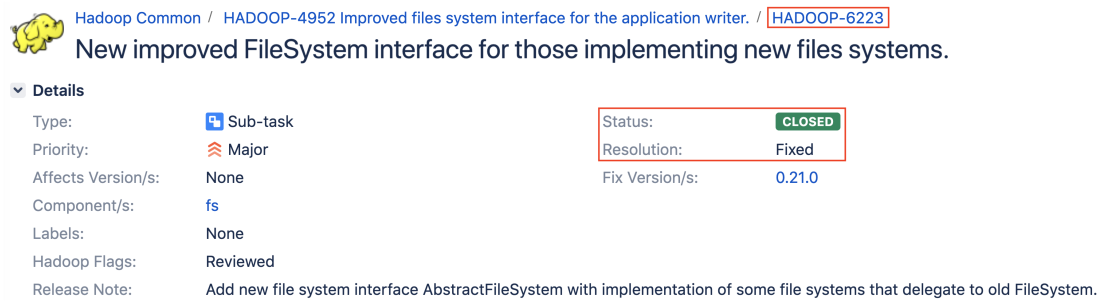
Potdar and Shihab extended the concept of TD to the notion of self-admitted technical debt (SATD) Potdar and Shihab [57], performed intentionally by developers, but mentioned/admitted as comments in the source code. They found that SATD is present, depending on the system, in 2.4% to over 30% of the files and that only 26%-63% gets removed, *i.e.,* a non-SATD often remains in the code. Zampetti et al. furthermore found that 20% - 50% of the removals were accidental and are even unintended Zampetti et al. [91].

Maldonado and Shihab categorized SATD into 5 types: design debt, defect debt, documentation debt, requirement debt, and test debt Maldonado and Shihab [47], with design debt and requirement debt being the most common ones. Xavier et al. also found that SATD not only manifests itself as comments in the source code, but is also present in issue reports Xavier et al. [88].

We focus on a particular type of SATD, first introduced in Chapter 5: "On-hold SATD", defined as self-admitted technical debt due to a waiting condition for an external event to happen before the technical debt can be removed. In particular, this chapter focuses on On-hold SATD with references to issues.

```
@InterfaceStability.Unstable /* return type will change to AFS once
                              HADOOP-6223 is completed */
public AbstractFileSystem getDefaultFileSystem() {
  return defaultFS;
}
```

(a) A SATD code comment referencing an issue



Hadoop Common / HADOOP-4952 Improved files system interface for the application writer. / HADOOP-6223

New improved FileSystem interface for those implementing new files systems.

Details

Type:	Sub-task	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	0.21.0
Component/s:	fs		
Labels:	None		
Hadoop Flags:	Reviewed		
Release Note:	Add new file system interface AbstractFileSystem with implementation of some file systems that delegate to old FileSystem.		

(b) A referenced issue report (https://tinyurl.com/ybunu2dj)

Figure 6.1. Motivating Example

Motivating Example. Fig. 6.1-(a) shows code from Apache Hadoop. The comment in the code indicates that an action will be taken once a condition is fulfilled, *i.e.,* the closing of issue 6223. As we see from Fig. 6.1-(b), the issue has in fact already been closed, but the On-hold SATD was not removed, thus creating confusion to anyone inspecting the code.

In essence, On-hold SATD are intentionally reminders left in the source code whose sole purpose is to be removed.

We present a large-scale empirical study to ascertain whether (i) On-hold SATD can be automatically detected, and (ii) it is possible to identify cases in which the On-hold SATD should be removed, since the "waiting condition" has been fulfilled, thus making the SATD a form of "wrong documentation" in the code. Besides quantitatively evaluating the approaches we built to identify and remove On-hold SATD instances, we also show its usefulness in practice by collecting feedback from developers of open source projects.

2 Method

We aim to build a classifier which automatically detects On-hold SATD and indicates whether it is ready to be removed. To achieve this goal, we took the following four steps (Fig. 6.2): 1) issue reference detection, 2) dataset creation, 3) data preprocessing, and 4) On-hold SATD classification.

2.1 Issue Reference Detection

To detect On-hold SATD, our first step is to locate the code comments referring to issues.

Project Selection

We selected ten open source projects that consistently used for their entire change history a specific issue tracking system (ITS). This allowed us to run our study without the risk of missing important information due to the migration between different issue tracker systems (*e.g.,* starting on JIRA and then moving to the GitHub issue tracker). Table 6.1 lists the projects used in this study.

Table 6.1. Details of the projects in my dataset. SLOC is calculated on Java files using SLOCCounts [87].

Project	**Version**	**ITS**	**SLOC**	**# Contributors**	**# Remaining comments that refer to issues**	**# Removed comments that refer to issues**
Apache Ant	1.10.7	Bugzilla	144,966	47	27	22
Apache Camel	3.0.0	Jira	1,267,905	544	42	62
Apache Dubbo	2.7.4	Github	148,377	268	8	4
Apache Hadoop	2.10.0	Jira	1,885,604	239	272	269
Apache Jmeter	5.2.1	Bugzilla	142,030	19	116	136
Apacha Kafka	2.4.0	Jira	319,990	606	24	21
Apache Log4j	1.2.17	Bugzilla	30,608	7	6	3
Apache Logging-log4j2	2.13.0	Jira	159,353	76	179	153
Apache Tomcat	10.0.0	Bugzilla	341,192	31	82	73
Mockito	3.3.10	Github	48,292	173	15	16
Total	-	-	4,488,317	2,010	771	759

Table 6.2. Regular expressions to identify issue in comments

ITS	**Regular expression**
Bugzilla	`(?<![A-Za-z])(?:bug|projectname|bugzilla|bz)[ -](?:#)?\d+(?:\.[0-9xX*]+)*`
	# issue IDs, e.g., Bug 34383
	`https?:\/\/[\w._/]*show_bug.cgi\?id=\d+`
	# URLs, e.g., `https://bz.apache.org/bugzilla/show_bug.cgi?id=51687`
Github	`(?<![A-Za-z])(?:bug|issues?)[ -](?:#)?\d+(?:\.[0-9xX*]+)*`
	# issue IDs, e.g., issue 55
	`https?:\/\/github.com/[\w._/]*\/issues\/\d+`
	# URLs, e.g., `https://github.com/apache/dubbo/issues/3251`
Jira	`(?<![A-Za-z])(?:bug|projectname)[ -](?:#)?\d+(?:\.[0-9xX*]+)*`
	# issue IDs, e.g., HADOOP-7234

Comment Extraction

We iterated over the commits in the repositories of the selected projects, and extracted all single line comments (*e.g.,* // ...) and multi-line comments (*e.g.,* /* ... */) from Java files. If multiple comments are next to each other (*e.g.,* /* ... */ // ...) they are considered as a single block of comments. Comments from test files are ignored, as issue references there are most likely to serve as explanations of what developers are testing, instead of SATD.

Issue Identification

Issue references are identified using regular expressions to match issue IDs and issue URLs. Table 6.2 shows the regular expressions used for each issue tracking system. For each identified issue reference, we also recorded its life cycle: We iterated over the commit history and extracted the date when the issue reference was first introduced in the comment, and in case, when it was removed.

From the 10 selected projects, we identified 1,530 comments containing issue references, among which 759 had already been removed, while the remaining 771 still remain in the latest commit by the date of data collection.

Table 6.3. Regular expressions for term abstraction

String	**ITS**	**Regular expression**
abstractissueid	Bugzilla	`(?<![A-Za-z])(?:bug|projectname|bugzilla|bz)[ -](?:#)?\d+(?:\.[0-9xX*]+)*`
		# issue IDs
		`https?:\/\/[\w._/]*show_bug.cgi\?id=\d+`
		# URLs
	Github	`(?<![A-Za-z])(?:bug|issues?)[ -](?:#)?\d+(?:\.[0-9xX*]+)*`
		# issue IDs
		`https?:\/\/github.com/[\w._/]*\/issues\/\d+`
		# URLs
	Jira	`(?<![A-Za-z])(?:bug|projectname)[ -](?:#)?\d+(?:\.[0-9xX*]+)*`
		# issue IDs
		`https?:\/\/issues.apache.org/\/jira\/browse\/(?:projectname)-\d+`
		# URLs
abstracturl	—	`https?:\/\/(www\.)?[-a-zA-Z0-9@:%._\+~#=]{2,256}\.[a-z]{2,6}\b`
		`([-a-zA-Z0-9@:%_\+.~#?&//=]*)`

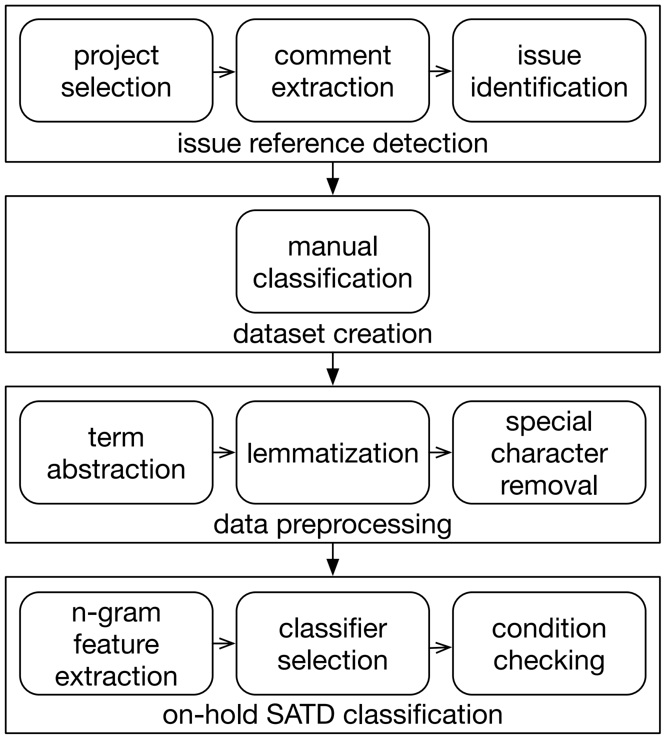


Figure 6.2. Approach for On-hold SATD detection and removal

2.2 Dataset Creation

To build the On-hold SATD classifier, we created a dataset for training and testing, based on the issue-referring comments collected in our previous step. For each of the 1,530 comments, I and the second researcher independently labeled whether it is an actual instance of On-hold SATD or, instead, it is used as cross-reference. We evaluated the inter-rater reliability with the Cohen's kappa coefficient, and the score of 0.748 demonstrates a substantial agreement between the two labelers. The third researcher resolved labeling conflicts. As a result, we got 133 On-hold SATD and 1,397 cross-reference comments.

Table 6.4 summarizes the annotation results. 133 (8.7%) of the issue-referring comments are instances of On-hold SATD.

Table 6.4. Statistics of annotated comments containing issue references

	On-hold SATD	**Cross-reference**	**Total**
Remaining comments	40	731	771
Removed comments	93	666	759
Total	133	1,397	1,530

2.3 Data Preprocessing

Before extracting features from the comments and feeding them into the classifier, we performed three preprocessing steps: 1) term abstraction, 2) lemmatization, and 3) special character removal.

Term Abstraction

For all the comments, we abstracted issue IDs and hyperlinks referring to issues to the string "`abstractissueid`", while the hyperlinks unrelated to issues were abstracted to "`abstracturl`". This is done to eliminate the impact of issue IDs and hyperlinks during classification, as we are not interested in their real content. Table 6.3 summarizes the regular expressions we used to extract relevant issue IDs and hyperlinks for different issue tracking systems.

Lemmatization

We applied lemmatization with the Spacy natural language processing tool Honnibal and Montani [25], which normalizes words with the same root but different surfaces into the same format Jurafsky and Martin [32]. For example, the words "sang", "singing", and "sings" will be converted into "sing".

Special character removal

We removed all non-English and non-numeric characters using the regular expression `[^A-Za-z0-9]+`.

For our study we did not apply stop word removal, a commonly used text preprocessing step, as it might remove some keywords important for identifying

On-hold SATD, such as “when” and “until”.

2.4 On-hold SATD Classification

After preprocessing, we extracted n-gram features from the comments and used them to train a classifier to identify On-hold SATD. We also checked issue status and issue resolution to determine whether an On-hold SATD comment is ready to be removed.

N-gram Feature Extraction

Similar to another SATD classification approach by Wattanakriengkrai et al. Wattanakriengkrai et al. [85], we extracted n-gram features by applying n-gram IDF Shirakawa et al. [67, 70]. N-gram IDF is a theoretical extension of IDF (Inverse Document Frequency). The traditional IDF approach assigns more weight to terms occurring in fewer documents, which does not work well for n-grams. For example, “Leonardo da is” might have higher weight than “Leonardo da Vinci”. N-gram IDF is designed to address this issue and can determine the dominant n-grams and extract key terms of any length Shirakawa et al. [67, 70]. In this study, we extracted n-grams from SATD comments using the library n-gram weighting scheme Shirakawa [66] with default settings. We obtained the list of all valid n-gram terms containing up to 10-gram terms. In total, we receive 644 terms of n-grams.

Classifier Selection

After extracting the n-gram terms, we build a classifier to identify bug referencing comments into On-hold SATD or not. While there many different algorithms available for supervised classification, it is hard to decide which one to pick, as different datasets and hyper-parameter settings might both impact the performance of these algorithms. Automated machine learning addresses this problem by running multiple classifiers with different parameters to optimize performance. In this study, we used auto-sklearn Feurer et al. [18], which includes 15 classification algorithms, 14 feature preprocessing and 4 data preprocessing techniques Feurer et al. [18].

Condition Checking

After identifying the On-hold SATD using our classifier, our program automatically checks the referred issue status and resolution to decide whether the SATD is ready to be removed. In the issue tracking system, if the status of the referred issue is set to "resolved", "closed", or "verified", and the field of resolution (if applicable) is set to "fixed", we consider it ready for removal.

3 Study Design

The *goal* of this study is to evaluate the accuracy of our approach for On-hold SATD identification and removal. Moreover, we are interested in the evolution of On-hold SATD in open source projects. The *context* of the study consists of 1,530 code comments containing issue references, extracted from the previously presented 10 open source projects.

3.1 Research Questions

In this study, we answer the following three research questions (RQs):

- **RQ$_1$:** *What is the accuracy of our approach in identifying On-hold SATD?* This RQ investigates the performance of our classifier in identifying On-hold SATD. We also examined the impact of oversampling, different features and machine learning algorithms on the performance of our classifier:
    - **RQ$_{1.1}$:** *How do n-grams impact the performance of our classifier as compared to Bag-Of-Words features?*
    - **RQ$_{1.2}$:** *How does oversampling impact the performance of the classifier?*
    - **RQ$_{1.3}$:** *How do different machine learning algorithms impact the performance of the classifier?*
- **RQ$_2$:** *How does On-hold SATD evolve in open source projects?* To gain deeper insights on how On-hold SATD evolves in the projects, with this RQ

we inspect the duration of existence of On-hold SATD in software projects, and the time it takes to address SATD after the relevant issue is resolved.

- **RQ$_3$:** *To what extent can our approach identify "ready-to-be-removed" On-hold SATD?* This RQ empirically evaluates the reliability of our approach in identifying On-hold SATD which should be removed, since it was already "paid back".

3.2 Context Selection & Data Collection

In this study, we used the dataset presented in Section 2.2, which contains 1,530 annotated comments containing issue references.

To answer RQ$_1$, we built a classifier using auto-sklearn with n-grams extracted by n-gram IDF Wattanakriengkrai et al. [85] as features. N-grams were extracted from On-hold SATD comments only. N-grams from Cross-reference comments are not included because we want to extract important patterns to detect on-hold SATD, and we use these patterns to discriminate between On-hold SATD and Cross-reference. We performed a ten-fold cross validation: We divided the 1,530 issue-referring comments into ten different sets, each one composed of 153 comments. Then, we iteratively used one set as the *test set*, while the remaining 1,377 comments were used for *training*.

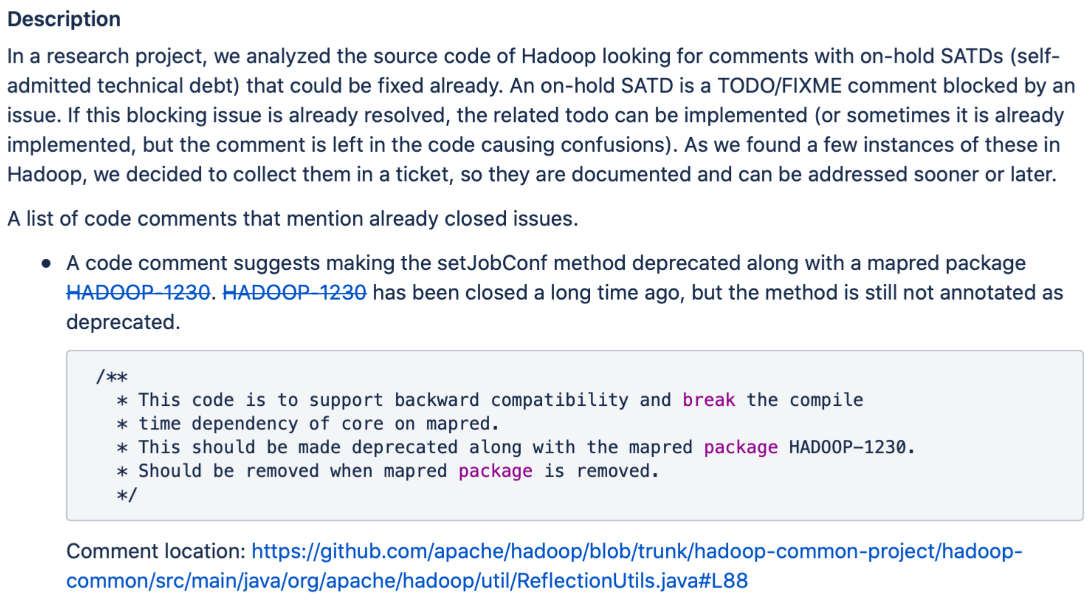
To answer RQ$_{1.1}$, we ran a different classifier implementation on the dataset, using Bag-Of-Words (BOW) as features.

To answer RQ$_{1.2}$, we applied an oversampling technique (*i.e.,* SMOTE) to our training set, and then compared the results achieved by our classifier with/without oversampling.

To answer RQ$_{1.3}$, we built three variants of the classifier with different machine learning algorithms: Naive Bayes, Support Vector Machine (SVM), and K-Nearest Neighbors (KNN).

To answer RQ$_2$, we inspected the removed issue-referring comments for both On-hold and cross-reference comments. We first checked the time interval between the introduction and the removal of these comments. Then, for the instances referring issues that have been solved, we compute the difference between the issue resolution time and the corresponding On-hold SATD removal event.

To answer RQ$_3$, we identified the On-hold SATD comments which are ready to be removed from the 40 still remaining On-hold issue-referring comments, based on the corresponding issue status and resolution, as described in Section 2.4. In total, we identified 10 "ready-to-be-removed" On-hold SATD comments. By the time we started working on RQ$_3$, 4 of 10 comments had already been removed by developers (three were removed thanks to code changes addressing the On-hold SATD, while one was removed due to the deletion of the file containing it). We reported the remaining six "ready-to-be-removed" On-hold SATD comments to the developers by creating issue reports in the respective issue tracker. In the issue report, we inform developers why the On-hold SATD comments should be removed and where they are located. An example of the issue reports can be seen in Fig. 6.3.


Description

In a research project, we analyzed the source code of Hadoop looking for comments with on-hold SATDs (self-admitted technical debt) that could be fixed already. An on-hold SATD is a TODO/FIXME comment blocked by an issue. If this blocking issue is already resolved, the related todo can be implemented (or sometimes it is already implemented, but the comment is left in the code causing confusions). As we found a few instances of these in Hadoop, we decided to collect them in a ticket, so they are documented and can be addressed sooner or later.

A list of code comments that mention already closed issues.

- A code comment suggests making the setJobConf method deprecated along with a mapred package ~~HADOOP-1230~~. ~~HADOOP-1230~~ has been closed a long time ago, but the method is still not annotated as deprecated.

```
/**
  * This code is to support backward compatibility and break the compile
  * time dependency of core on mapred.
  * This should be made deprecated along with the mapred package HADOOP-1230.
  * Should be removed when mapred package is removed.
  */
```

  Comment location: https://github.com/apache/hadoop/blob/trunk/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/util/ReflectionUtils.java#L88

Figure 6.3. An example issue report.

3.3 Data Analysis

To answer **RQ$_1$** we compare the precision, recall, F1-score, and area under the ROC curve (AUC) of each experimented approach in classifying issue-referring

Table 6.5. Performance of classifiers in identifying On-hold SATD

	Original approach	**BOW as feature**	**With Oversampling**	**Different ML algorithms**		
	n-gram + auto-sklearn	BOW + auto-sklearn	n-gram + oversampling + auto-sklearn	n-gram + Naive Bayes	n-gram + SVM	n-gram + KNN
Precision	0.79	0.69	0.38	0.64	0.87	0.88
Recall	0.70	0.68	0.48	0.56	0.38	0.15
F1-score	0.73	0.67	0.41	0.59	0.51	0.25
AUC	0.97	0.94	0.87	0.81	0.95	0.76

comments (as belonging or not to On-hold SATD) for the dataset of 1,530 comments.

The comparisons are also performed via the Mann-Whitney test Conover [8], with results intended as statistically significant at $\alpha = 0.05$. For $\text{RQ}_{1.3}$, to control the impact of multiple pairwise comparisons (*e.g.,* the precision of auto-sklearn is compared with Naive Bayes, SVM, and KNN), we adjust p-values with Holm's correction Holm [24]. We estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure Grissom and Kim [22]. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ Grissom and Kim [22].

To answer $\textbf{RQ}_2$, we present via violin plots the life spans of both On-hold SATD and cross-reference comments, as well as the duration between the resolution of issues and the removal of corresponding SATD comments.

To answer $\textbf{RQ}_3$, we qualitatively analyze the developers' feedback.

4 Evaluation

4.1 RQ_1: What is the accuracy of our approach in identifying On-hold SATD?

Table 6.5 reports the average precision, recall, F1-score, and AUC of each experimented classifier implementations during 10-fold evaluation.

Table 6.6 reports the statistical results of comparisons between different classifier implementations.

Table 6.6. Statistical results of performance comparisons of classifiers

	P-value (Precision)	**Effect size (Precision)**	**P-value (Recall)**	**Effect size (Recall)**
n-gram+auto-sklearn vs *BOW+auto-sklearn*	< 0.01	0.48 (large)	0.32	-
n-gram+auto-sklearn vs *n-gram+oversampling+auto-sklearn*	< 0.01	0.92 (large)	0.01	0.67 (large)
n-gram+auto-sklearn vs *n-gram+Naive Bayes*	0.06	-	0.03	0.58(large)
n-gram+auto-sklearn vs *n-gram+SVM*	0.30	-	0.03	0.8 (large)
n-gram+auto-sklearn vs *n-gram+KNN*	0.30	-	0.03	1.0 (large)
n-gram+Naive Bayes vs *n-gram+SVM*	0.06	-	0.03	0.58 (large)
n-gram+Naive Bayes vs *n-gram+KNN*	0.30	-	0.03	1.0 (large)
n-gram+SVM vs *n-gram+KNN*	0.31	-	0.03	0.74 (large)

Fig. 6.4 also shows the results of the 10-fold evaluation for each experimented classifier in terms of precision, recall, F1-Score, and AUC.

As can be seen from Table 6.5, the precision, recall, and F1-score achieved by our approach ("n-gram + auto-sklearn") are all between 0.7 to 0.8, while AUC is as high as 0.97. This result demonstrates the reliability of our approach in On-hold SATD detection.

To gain a better understanding of how our classifier works, we list the important n-gram features which frequently appear in On-hold SATD comments in Table 6.7.

Table 6.7. N-gram features which frequently appear in On-hold SATD comments

N-gram features	**Frequency**
'after', 'abstractissueid'	20
'once', 'abstractissueid'	18
'for', 'now'	12
'temporary', 'fix'	10
'workaround'	10
'this', 'be', 'a', 'temporary'	8
'via', 'abstractissueid'	7
'be', 'commit'	7
'can', 'be', 'remove'	7
'remove', 'after', 'abstractissueid'	5

These features help discriminate On-hold SATD from cross-reference. We can see that n-grams such as "*once abstractissueid*", "*this be a temporary*", and "*remove after abstractissueid*" are especially important for identifying On-hold SATD.

Additionally, we also illustrate some classification results in Table 6.8. From the two true positive examples (correctly identified as On-hold SATD by our approach), we can clearly see the patterns including "*workaround*", "*temporary fix*" and "*remove after abstractissueid*", which can be related to Table 6.7. Therefore, it is not surprising that our classifier can correctly identify these On-hold SATD comments.

Table 6.8. Example of classification results of my approach

Type	**Comment**
True Positive	TODO: **workaround** (filling fixed bytes), to **remove after HADOOP-11938**
	... This is a **temporary fix** ... See the discussion on HDFS-1965.
False Negative	TODO: Temporarily keeping ... This has *to be revisited* as part of HDFS-11029.
	placeholder for javadoc to prevent broken links, *until HADOOP-6920*
False Positive	**TODO: after MAPREDUCE-2793** YarnException is probably not expected here anymore but keeping it for now ...
	... (CAMEL-9657) **[TODO] Remove** in 3.

If we take a look at the two false negative examples (On-hold SATD classified as cross-reference), we find that phrases like "*to be revisit*" and "*until abstractissueid*" are probably useful n-grams for identifying On-hold SATD. Due to absence or infrequent occurrence, these n-grams are not used as features for the classifier. Expanding the training set can be a potential way for addressing the n-gram feature limitations.

In the two false positive examples (cross-reference classified as On-hold SATD), we can see that the n-gram terms like "*todo after abstractissueid*" and "*todo remove*" can be actually matched, and our classifier misclassified them into On-hold SATD. However, if we check the comments carefully, we can find that in the first sentence, it is clear that the issue has already been resolved, however, for some reason the developers decided to say "*keeping it for now*", where "*it*" refers to `YarnException`. In the second sentence, what follows "*todo remove*" is actually not a reference to an issue, but a reference to a version. Some heuristic rules might help our classifier to better deal with these cases.

To understand how n-grams impact the performance of our classifier as compared to Bag-Of-Words (BOW) features, we inspect the first two columns of Table 6.5, and the first row of Table 6.6. Using n-grams as features leads to a higher precision with a statistically significant difference and a large effect size. As for the recall, while the average value is higher when using n-grams, the performed analysis does not indicate a statistically significant difference. We conclude that compared to BOW features, n-grams lead to a significantly higher precision.

To understand how oversampling impacts the performance of the classifier, we inspect the first and the third column of Table 6.5, as well as the second row of Table 6.6.

From the tables we can observe that the classifier obtains a statistically significant higher precision and recall when oversampling is not applied. Meanwhile, the effect sizes for both precision and recall comparisons are large. Indeed, after applying oversampling, the average precision, recall, F1-score, and AUC drop by around 40%, 20%, 30%, and 10%, respectively. We conclude that oversampling reduces the performance of our classifier in identifying On-hold SATD.

To understand how different machine learning algorithms impact the performance of the classifier, we inspect the first and the last three columns of Table 6.5, as well as the last six rows of Table 6.6. From the tables we can see that all the implementations achieved comparable precisions (from 0.64 to 0.88). Indeed, there is no statistically significant difference in terms of precision among these implementations. However, the differences emerge when comparing recall. Using auto-sklearn achieves a significantly higher recall than classifiers using other machine learning algorithms (*i.e.,* Naive Bayes, SVM and KNN).

We also inspected which machine learning algorithm was adopted by auto-sklearn after automatic classifier selection. The records show that in 9 of the ten rounds of 10-fold evaluation Extra Trees was adopted, while the remaining one adopted Random Forest. That is, these two machine learning algorithms would potentially be a good choice for identifying On-hold SATD when automatic selection of the classifier is not possible.

4.2 RQ$_2$: How does On-hold SATD evolve in open source projects?

To answer RQ$_2$, we first looked into the life span of removed issue-referring comments for On-hold SATD and cross-reference comments separately. The life span distributions can be found in Fig. 6.5.

The median life span of On-hold SATD comments is 42 days, while it is 119.5 days for cross-reference comments. That is, overall, the median life span of cross-reference comments is almost three times of that of On-hold SATD.

Indeed, while both types of comments contain issue references, only On-hold SATD requires maintenance actions from developers. Cross-reference comments stay much longer as they are usually used for documentation purposes.

We then investigated how long it takes to address On-hold SATD comments after the corresponding issues are resolved, and plotted the duration distribution in Fig. 6.6.

Around 53% of On-hold SATD were removed within the same day when the issue was resolved. However, it takes longer than one year to remove 13% of On-hold SATD.

Additionally, we observed that some developers did not wait until the issue was resolved to address On-hold SATD comments. In fact, from a total of 93 removed On-hold SATD comments, we found that only 30 of them were removed after the issues were resolved. The corresponding issues of 9 On-hold SATD comments are still open or have the resolution set to "wontfix". 54 On-hold SATD comments were removed before the issues were resolved, although these issues have been resolved in the meantime.

4.3 RQ$_3$: To what extent can our approach identify "ready-to-be-removed" On-hold SATD?

To understand how well our approach performs in identifying "ready-to-be-removed" On-hold SATD comments, we reported six identified cases to developers in three issue reports, as these six cases correspond to three subsystems of the Apache Hadoop project (two for Hadoop Common, one for Hadoop HDFS, and three for Hadoop YARN). By the time of writing, we have received the feedback from the developers about the two "ready-to-be-removed" On-hold SATD comments in the Hadoop Common subsystem.

Table 6.9 lists these two instances of On-hold SATD reported to the developers in JIRA issue tracking system.[1]

For the first case, the return type had already changed to AFS, and the resolution of the referred issue "HADOOP-6223" had been set to "resolved". In the issue report, we suggested that this On-hold SATD comment should be removed.

[1]https://issues.apache.org/jira/browse/HADOOP-17047

Table 6.9. Two “ready-to-be-removed” On-hold SATD comments which received developers’ feedback

#	**Ready On-hold SATD**
1	“/* return type will change to AFS once HADOOP-6223 is completed */”
2	“... This should be made deprecated along with the mapred package HADOOP-1230. ... ”

Developers agree that it can be removed:

> *“I think this is correct finding. Would you like to put a patch for this”*

Later on, the patch we submitted got integrated into the repository.

Regarding the second case, we found that the referred issue “HADOOP-1230” had also been resolved. Thus, we suggested that developers could apply corresponding changes (*i.e.,* making the `setJobConf` method deprecated). The developers agreed that the action should be taken but it is a rather complicated fix, thus recommending a new JIRA issue thread:

> *“...we need to update the document in a separate jira.”*

> *“... Given that is a bigger subject than this fix, we should discuss on that separately ...”*

Overall, the two cases for which we have already received feedback on indicate the practical value of our approach for On-hold SATD identification and removal.

4.4 Replication

To facilitate replication, we released our dataset in our online appendix, which can be accessed at https://tinyurl.com/onholdissue. The spreadsheet file of our dataset contains three sheets: removed comments, remaining comments, and identified “ready-to-be-removed” On-hold SATD. For all the comments in our dataset, we include the comment context, code file path, line number, referred issue, and our annotation (On-hold SATD or cross-reference). For removed comments, we also include when the code comment was introduced and removed. For the "ready-to-be-removed" On-hold SATD, there are also the status and the resolution of the corresponding issues.

5 Towards a On-Hold SATD Recommender

Our findings can serve as guideline for developers writing reference issues in code comments:

- Developers should check SATD comments referring to issues which had already been resolved, as we reported that 13% of comments were removed with a delay of more than one year.
- When the code comments refer to issues, developers should clearly mention the intention in the comments, *i.e.,* whether the issue is used for documentation or to denote the condition on which one is waiting on.

While we plan on expanding our work to analyze more projects and to include also other issue tracking systems, we believe that our work can be synthesized into a recommender system for On-Hold SATD. In Fig. 6.7 we depicted a mock-up of such a recommender.

The tool would report the list of On-hold SATD comments, ready to be addressed On-hold SATD comments, and removed On-hold SATD. Each item would include comments, links to the original comments and to the pertaining issue (including its status and duration).

6 Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work they are mainly due to the measurements we performed:

- *Imprecisions in the identification of issue references in comments.* We used the regular expressions in Table 6.2 to mine issue references in code comments. The regular expressions have been defined and tested by myself, and are customized for each of the issue trackers used by the subject systems.

- *Subjectivity/errors in the manual classification.* To mitigate this threat, I and second researcher independently classified the 1,530 issue-referencing comments as On-hold SATD or as cross-reference. Then, the third researcher resolved the conflicts.

Threats to *external validity* concern the generalizability of results. Rather than going large-scale, we preferred to work on a set of ten well-known Java open source projects and to manually validate all issue-referencing comments we found in them in such a way to increase the reliability of the presented data. Other systems should be included in the analysis to allow for a broader generalizability of our conclusions. Also, the results of RQ_3 are based on only two feedback we received from developers, thus do not allowing any sort of generalizability but only serving as pointers for qualitative analysis.

7 Conclusion

Since the definition of the term "technical debt" by Cunningham three decades ago Cunningham [9], researchers have investigated the phenomenon, leading to the understanding that it is its creeping, barely visible nature that leads to maintenance and evolvability problems down the road. Developers cannot be faulted for the introduction of technical debt, as software industry functions under great time and budget pressure, and compromises have to be made to meet said time and budget constraints. Indeed, developers often admit that they are creating technical debt, which led to the term "self-admitted technical debt" (SATD) coined by Potdar and Shihab Potdar and Shihab [57].

A particular type of SATD is the one we named "On-hold" SATD, where a developer has to make a compromise or halt development because of an external condition. Human nature dictates that often On-hold SATD is simply forgotten about.

We performed an empirical study to understand whether On-hold SATD can be automatically detected: We analyzed ten open source projects, and found that 8% of the comments referring to issues are On-hold SATD. To identify On-hold SATD, we developed a classifier using n-gram and auto-sklearn, resulting in an average precision of 0.79, an average recall of 0.70, an average F_1-score of 0.73, and an average AUC of 0.97. In short, On-hold SATD can indeed be detected automatically in a fairly reliable way.

To understand how On-hold SATD evolves, we looked into life-span of removed issue-referring comments. We found that the median life-span of On-hold comments is 42 days. This is certainly beyond the horizon of human short-term memory, and indeed we found that after the issues were resolved, 13% of On-hold SATD takes longer than one year to remove. To evaluate the reliability in identifying On-hold SATD which should be removed, we collected feedback from developers from open source projects. Developers agreed with our findings that the reported On-hold SATD should be fixed or removed.

The next logical step is thus the design and implementation of the recommender system we described in Section 5 and aimed at facilitating the identification, understanding, and resolution of On-hold SATD instances.

Acknowledgement

We gratefully acknowledge the financial support of Japan Society for the Promotion of Science for the JSPS KAKENHI Grant No. 16H05857 and 20H05706, and the Swiss National Science Foundation for the project SENSOR (SNF-JSPS Project No. 183587).

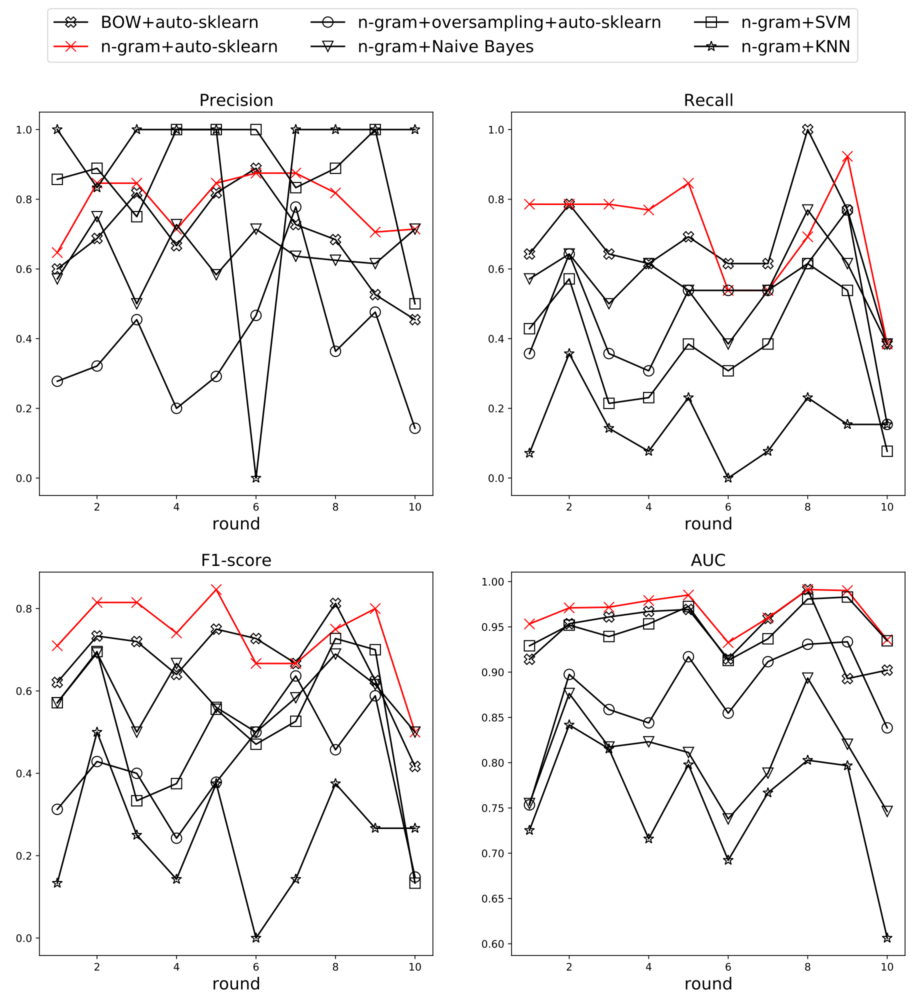


Figure 6.4. Results of each round in 10-fold evaluation for different classifier implementations

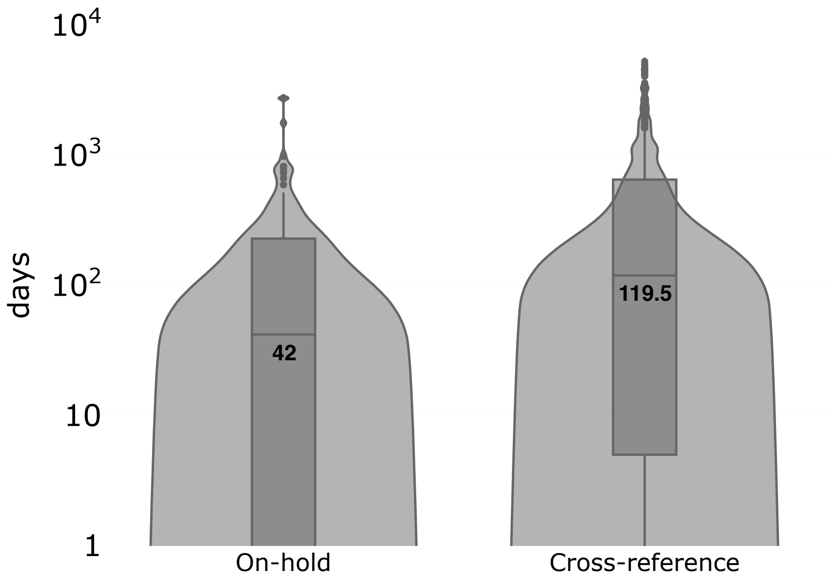


Figure 6.5. Distribution of life spans of removed issue-referring comments

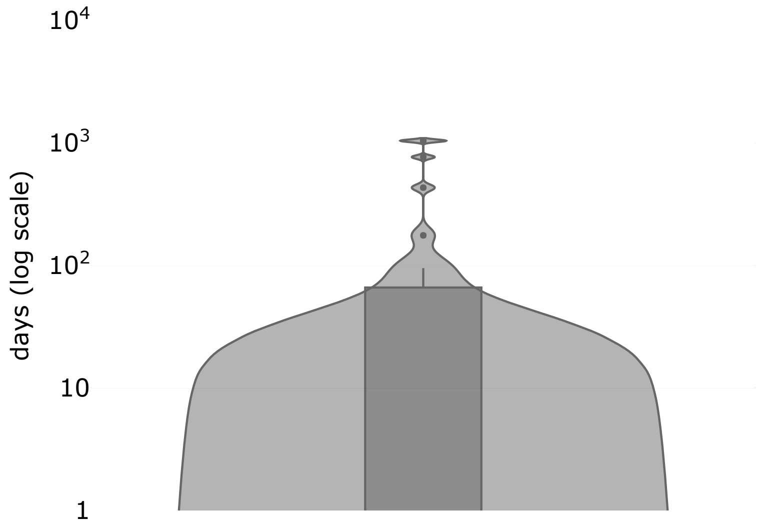
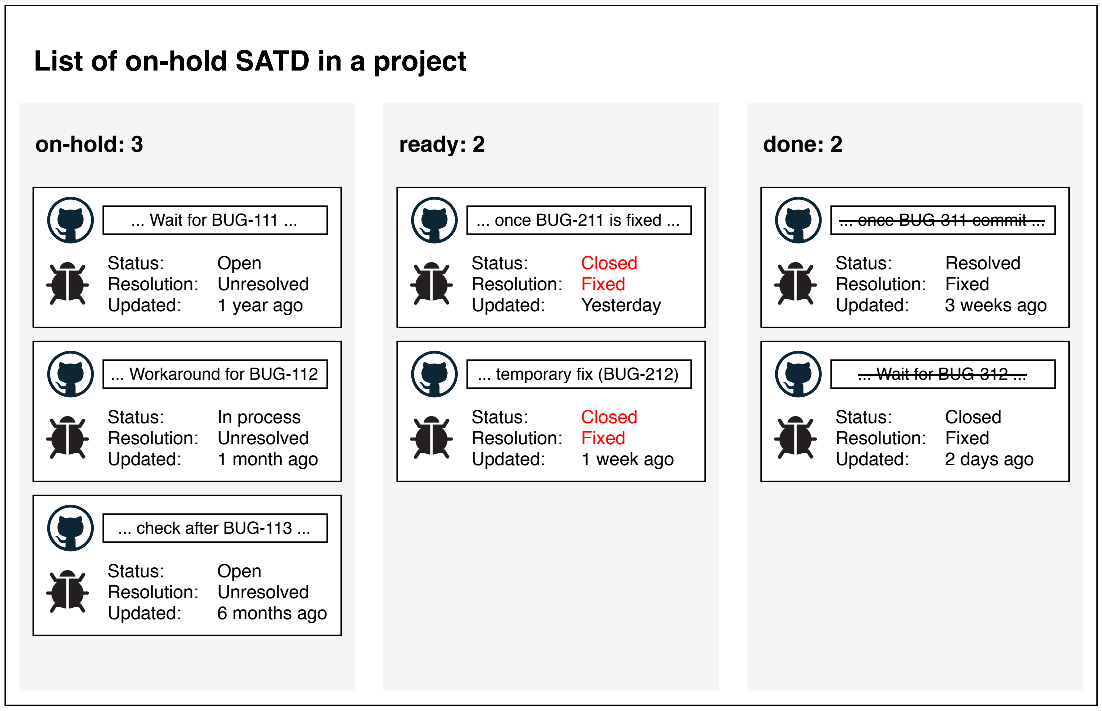


Figure 6.6. Distribution of days needed to address SATD comments after issues were resolved



List of on-hold SATD in a project

on-hold: 3

... Wait for BUG-111 ...

Status:	Open
Resolution:	Unresolved
Updated:	1 year ago

... Workaround for BUG-112

Status:	In process
Resolution:	Unresolved
Updated:	1 month ago

... check after BUG-113 ...

Status:	Open
Resolution:	Unresolved
Updated:	6 months ago

ready: 2

... once BUG-211 is fixed ...

Status:	Closed
Resolution:	Fixed
Updated:	Yesterday

... temporary fix (BUG-212)

Status:	Closed
Resolution:	Fixed
Updated:	1 week ago

done: 2

~~... once BUG-311 commit ...~~

Status:	Resolved
Resolution:	Fixed
Updated:	3 weeks ago

~~... Wait for BUG-312 ...~~

Status:	Closed
Resolution:	Fixed
Updated:	2 days ago

Figure 6.7. A mockup of On-hold SATD identification tool

7 | Conclusion

Software documentation plays an important role in software development. It provides information about software for developers and users, including how software works and how to use it. A well-written software document boosts efficiency and quality in software development. Despite its benefits, software documentation is often poorly written and frequently outdated.

In this thesis, I focus on demonstrate the value of existing information in addressing problems caused by a lack of software documentation and frequently outdated documents: (1) In order to address the issue of lack of software documentation, I propose new technique to access existing information using sentiment analysis, and (2) In order to address the issue of frequently outdated document problem, I assist in managing software documents by identifying and removing unnecessary self-admitted technical debt (i.e., situations where a software developer knows that their current implementation is not optimal and indicates this using a source code comment).

For the first problem, I designed a new identification tool to help developers understand the sentiment of the text in a software engineering environment. As a result, developers could use this tool to automatically understand the intention underlining messages. The results showed the proposed tool outperformed the current off-the-shelf tool, especially in predicting positive and negative sentiment.

For the second problem, I identify one type of self-admitted technical debt that is amenable to automated management "on-hold SATD". I define on-hold SATD as self-admitted technical debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere. I designed a classifier which can automatically identify those instances of self-admitted technical debt that are on hold and detect the specific events that developers are waiting for. For that, three types of conditions are identified, waiting for a bug to be fixed, releasing a library, or releasing a new version of a library. Among these, one of the conditions, which refers to outside developer control, is waiting for a bug to be fixed, which refers to the issue tracker system. From that, it is possible to identify cases in which the On-hold SATD should be removed, since the "waiting condition" has been fulfilled, thus making the SATD a form of "wrong documentation" in the code.

The outcomes of this thesis contribute to: (1) designing and evaluating tools which can identify underlining sentiment from text in software engineering documents, (2) understanding what kind of self-admitted technical debt is amenable to automated management (*i.e.,* on-hold SATD), (3) designing and evaluating tools that can classify on-hold SATD and its waiting condition, (4) suggesting which on-hold SATD is ready to be removed.

Based on the results of this thesis, by unlocking software documentation, I could support developers in maintaining and understanding software documents. Specifically, the first part of this thesis contributes to the unlocking of underlining messages from various software documents using sentiment analysis. The second part of this thesis contributes to the managing of software documents, especially self-admitted technical debt in code comments.

1 Implications

The main goal of this thesis is to help developers unlocking software documents to show the benefit of existing information (1) proposing new technique to develop sentiment analysis framework in software engineering environment, and (2) address the issue of frequently outdated document problem by assisting in managing software documents by identifying and removing unnecessary self-admitted technical debt (i.e., situations where a software developer knows that their current implementation is not optimal and indicates this using a source code comment). To achieve these goals, I perform two developer surveys and three empirical studies with three design of classification tools. As a result, the empirical findings of this thesis may be useful to both developers and researchers. I summarize the findings and suggestion for each part as follows:

- **For developers:** Regarding developers writing code comments, my findings can serve as guideline developers should check SATD comments referring to issues which had already been resolved, as I reported that 13% of comments were removed with a delay of more than one year. When the code comments refer to issues, developers should clearly mention the intention in the comments, *i.e.,* whether the issue is used for documentation or to denote the condition on which one is waiting on.

- **For project maintainers:** In terms of the impact of sentiment analysis classification utilizing the suggested tools, this enables project maintainers to monitor feedback in real time. Regarding the condition of on-hold SATD comments, three main conditions are related to waiting for a bug to be fixed, releasing a library, or releasing a new version of a library. This finding shows that by maintaining SATD comments that are ready to be addressed can prevent bugs or vulnerabilities that might otherwise occur, e.g., because of outdated libraries.

2 Opportunities for Future Research

In this thesis, I investigate (1) sentiment analysis and (2) on-hold self-admitted technical debt. However, there are still a lot of research aspect that can be done in order to help developers towards creating the high-quality application. In the following, I outline the research opportunities for the immediate future.

Sentiment analysis on software engineering documents In Chapter 4, I design a classification tool for software engineering documents which help developers in understanding the sentiment in a software engineering context. Additional research in software engineering documents, such as the relationship between sentiment in pull request comments and pull request status, can be further explore using my suggested method.

On-hold self-admitted technical debt condition detection. In Chapter 5, I found that three main types of conditions that developers were waiting for were waiting for a bug to be fixed, releasing a library, or releasing a new version of a library. However, those three conditions account for half of the waiting conditions, implying that more research is needed to expand for other conditions in on-hold self-admitted technical debt.

A recommender system for On-Hold SATD In Chapter 6, I found that it is possible to identify which on-hold SATD should be removed when the waiting condition has been fulfilled. From that, I believe that my work can be synthesized into a recommended system for On-Hold SATD. One example is a tool developed by Phaithoon et al. [56].

References

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms, 2(1): 53–86, 2004.

[2] Fouad Nasser A Al Omran and Christoph Treude. Choosing an nlp library for analyzing software documentation: A systematic literature review and a series of experiments. In Proceedings of the International Conference on Mining Software Repositories, pages 187–197, 2017.

[3] N.S.R. Alves, L.F. Ribeiro, V. Caires, T.S. Mendes, and R.O. Spinola. Towards an ontology of terms on technical debt. In Managing Technical Debt (MTD), 2014 Sixth International Workshop on, pages 1–7, 2014.

[4] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pages 315–326, 2016.

[5] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016, pages 315–326, 2016.

[6] Saman Bazrafshan and Rainer Koschke. An empirical study of clone removals. In Proceedings of the International Conference on Software Maintenance, pages 50–59, 2013.

[7] Dmitriy Bespalov, Bing Bai, Yanjun Qi, and Ali Shokoufandeh. Sentiment classification based on supervised latent n-gram analysis. In Proceedings of the 20th ACM

International Conference on Information and Knowledge Management, CIKM '11, pages 375–382, 2011.

[8] William J. Conover. Practical nonparametric statistics. Wiley New York, 3rd edition, 1998.

[9] Ward Cunningham. The wycash portfolio management system. In Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum), OOPSLA '92, page 29–30, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897916107. doi: 10.1145/157709.157715. URL https://doi.org/10.1145/157709.157715.

[10] M. A. d. F. Farias, M. G. d. M. Neto, A. B. d. Silva, and R. O. Spínola. A contextualized vocabulary model for identifying technical debt on code comments. In 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pages 25–32, 2015.

[11] E. d. S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering, 43(11):1044–1062, 2017.

[12] Everton da S. Maldonado and Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. In 7th IEEE International Workshop on Managing Technical Debt, MTD 2015, Bremen, Germany, October 2, 2015, pages 9–15, 2015.

[13] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering, 43(11):1044–1062, 2017.

[14] Mário André de F. Farias, Railan Xisto, Marcos S. Santos, Raphael S. Fontes, Methanias Colaço, Rodrigo Spínola, and Manoel Mendonça. Identifying technical debt through a code comment mining tool. In Proceedings of the XV Brazilian Symposium on Information Systems, SBSI'19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372374. doi: 10.1145/3330204.3330227.

[15] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, André Batista da Silva, and Rodrigo Oliveira Spínola. A contextualized vocabulary model for identifying technical debt on code comments. In 2015 IEEE 7th International Workshop

on Managing Technical Debt (MTD), pages 25–32, 2015. doi: 10.1109/MTD.2015.7332621.

[16] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In Proceedings of the Joint Meeting on Foundations of Software Engineering, pages 50–60, 2015.

[17] Ronen Feldman. Techniques and applications for sentiment analysis. Commun. ACM, 56(4):82–89, apr 2013. ISSN 0001-0782. doi: 10.1145/2436256.2436274. URL https://doi.org/10.1145/2436256.2436274.

[18] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, Advances in Neural Information Processing Systems 28, pages 2962–2970. Curran Associates, Inc., 2015.

[19] Jernej Flisar and Vili Podgorelec. Identification of self-admitted technical debt using enhanced feature selection based on word embedding. IEEE Access, 7:106475–106494, 2019. doi: 10.1109/ACCESS.2019.2933318.

[20] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In Proceedings of the 2002 ACM Symposium on Document Engineering, DocEng '02, page 26–33, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135947. doi: 10.1145/585058.585065. URL https://doi.org/10.1145/585058.585065.

[21] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. Evaluating usage and quality of technical software documentation: An empirical study. In Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, EASE '13, page 24–35, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318488. doi: 10.1145/2460999.2461003. URL https://doi.org/10.1145/2460999.2461003.

[22] Robert J. Grissom and John J. Kim. Effect sizes for research: A broad practical approach. Mahwah, NJ: Earlbaum, 2005.

[23] Yuepu Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F.Q.B. da Silva, A.L.M. Santos, and C. Siebra. Tracking technical debt - an exploratory case study. In Software Maintenance (ICSM), 2011 27th IEEE International Conference on, pages 528–531, 2011.

[24] Sture Holm. A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics, pages 65–70, 1979.

[25] Matthew Honnibal and Ines Montani. spacy - industrial-strength natural language processing in python. https://spacy.io/, 2017. (Accessed on 13/04/2019).

[26] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering, 23(1):418–451, 2018.

[27] C. Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text, 2014.

[28] Tomohiro Ichinose, Kyohei Uemura, Daiki Tanaka, Hideaki Hata, Hajimu Iida, and Kenichi Matsumoto. ROCAT on KATARIBE: Code visualization for communities. In Proceedings of the International Conference on Applied Computing and Information Technology, pages 158–163, 2016.

[29] AltexSoft Inc. Software documentation types and best practices. https://blog.prototypr.io/software-documentation-types-and-best-practices-1726ca595c7f/, 2018. (Accessed on 29/06/2022).

[30] Md Rakibul Islam and Minhaz F. Zibran. Leveraging automated sentiment analysis in software engineering. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, pages 203–214, 2017.

[31] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. On negative results when using sentiment analysis tools for software engineering research. Empirical Softw. Engg., 22(5):2543–2584, October 2017.

[32] Daniel Jurafsky and James H. Martin. Speech and language processing, 2009.

[33] Mira Kajko-Mattsson. A survey of documentation practice within corrective maintenance. Empirical Software Engineering, 10(1):31–55, jan 2005. doi: 10.1023/b:

lida.0000048322.42751.ca. URL https://doi.org/10.1023%2Fb%3Alida.0000048322.42751.ca.

[34] Yasutaka Kamei, Everton Maldonado, Emad Shihab, and Naoyasu Ubayashi. Using analytics to quantify the interest of self-admitted technical debt. CEUR Workshop Proceedings, 1771:68–71, 2016.

[35] Tim Klinger, Peri Tarr, Patrick Wagstrom, and Clay Williams. An enterprise perspective on technical debt. In Proceedings of the 2Nd Workshop on Managing Technical Debt, MTD ’11, pages 35–38, 2011.

[36] Henrik Kniberg. Good and bad technical debt (and how TDD helps), 2013. http://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt.

[37] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. IEEE Software, 29(6):18–21, 2012. ISSN 0740-7459.

[38] Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, and Davide Falessi. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. SIGSOFT Softw. Eng. Notes, 38(5):51–54, 2013.

[39] T.C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. IEEE Software, 20(6):35–39, 2003. doi: 10.1109/MS.2003.1241364.

[40] Shoushan Li, Sophia Yat Mei Lee, Ying Chen, Chu-Ren Huang, and Guodong Zhou. Sentiment classification and polarity shifting. In Proceedings of the 23rd International Conference on Computational Linguistics, COLING ’10, pages 635–643, 2010.

[41] E. Lim, N. Taksande, and C. Seaman. A balancing act: What software practitioners have to say about technical debt. IEEE Software, 29(6):22–27, 2012.

[42] E. Lim, N. Taksande, and C. Seaman. A balancing act: What software practitioners have to say about technical debt. Software, IEEE, 29(6):22–27, 2012.

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

[44] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. Sentiment analysis for software engineering: How far can we go? In Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pages 94–104, 2018.

[45] 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 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 9–12, 2018.

[46] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. Satd detector: A text-mining-based self-admitted technical debt detection tool. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 9–12, 2018.

[47] Everton da S. Maldonado and Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. In 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pages 9–15, 2015. doi: 10.1109/MTD.2015.7332619.

[48] Everton Da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 238–248, 2017. doi: 10.1109/ICSME.2017.8.

[49] Steve McConnell. Technical debt, 2007. http://www.construx.com/10x_Software_Development/Technical_Debt/.

[50] Solomon Mensah, Jacky Keung, Michael Franklin Bosu, and Kwabena Ebo Bennin. Rework effort estimation of self-admitted technical debt. CEUR Workshop Proceedings, 1771:72–75, 2016.

[51] Solomon Mensah, Jacky Keung, Jeffery Svajlenko, Kwabena Ebo Bennin, and Qing Mi. On the value of a prioritization scheme for resolving self-admitted technical debt. Journal of Systems and Software, 135(C):37–54, 2018.

[52] Marco Ortu, Bram Adams, Giuseppe Destefanis, Parastou Tourani, Michele Marchesi, and Roberto Tonelli. Are bullies more productive?: Empirical study of affectiveness vs. issue fixing time. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pages 303–313, 2015.

[53] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In Proceedings of the International Conference on Program Comprehension, pages 176–185, 2017.

[54] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), ICSME '15, pages 281–290, 2015.

[55] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. the Journal of machine Learning research, 12:2825–2830, 2011.

[56] Saranphon Phaithoon, Supakarn Wongnil, Patiphol Pussawong, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Rungroj Maipradit, Hideaki Hata, and Kenichi Matsumoto. Fixme: A github bot for detecting and monitoring on-hold self-admitted technical debt. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1257–1261, 2021. doi: 10.1109/ASE51524.2021.9678680.

[57] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME '14, page 91–100, USA, 2014. IEEE Computer Society. ISBN 9781479961467. doi: 10.1109/ICSME.2014.31. URL https://doi.org/10.1109/ICSME.2014.31.

[58] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 91–100, 2014. doi: 10.1109/ICSME.2014.31.

[59] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github readme files. Empirical Software Engineering, 2019.

[60] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. Neural network-based detection of self-admitted technical debt: From performance

to explainability. ACM Trans. Softw. Eng. Methodol., 28(3), July 2019. ISSN 1049-331X. doi: 10.1145/3324916. URL https://doi.org/10.1145/3324916.

[61] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. Neural network-based detection of self-admitted technical debt: From performance to explainability. ACM Trans. Softw. Eng. Methodol., 28(3), jul 2019. ISSN 1049-331X. doi: 10.1145/3324916.

[62] Kenneth S. Rubin. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley Professional, 1st edition, 2012.

[63] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. Information Processing and Management, 24(5):513–523, 1988.

[64] CJ Satish and M Anand. Software documentation management issues and practices: A survey. Indian Journal of Science and Technology, 9(20):1–7, 2016.

[65] Masumi Shirakawa. Github - iwnsew/ngweight: N-gram weighting scheme. https://github.com/iwnsew/ngweight, June 2017. (Accessed on 04/13/2019).

[66] Masumi Shirakawa. N-gram weighting scheme, Jul 2017. URL https://github.com/iwnsew/ngweight.

[67] Masumi Shirakawa, Takahiro Hara, and Shojiro Nishio. N-gram idf: A global term weighting scheme based on information distance. In Proceedings of the 24th International Conference on World Wide Web, WWW '15, page 960–970, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee. ISBN 9781450334693. doi: 10.1145/2736277.2741628. URL https://doi.org/10.1145/2736277.2741628.

[68] Masumi Shirakawa, Takahiro Hara, and Shojiro Nishio. N-gram idf: A global term weighting scheme based on information distance. In Proceedings of the International Conference on World Wide Web, pages 960–970, 2015.

[69] Masumi Shirakawa, Takahiro Hara, and Shojiro Nishio. N-gram idf: A global term weighting scheme based on information distance. In Proceedings of the 24th International Conference on World Wide Web, WWW '15, pages 960–970, 2015.

[70] Masumi Shirakawa, Takahiro Hara, and Shojiro Nishio. Idf for word n-grams. ACM Trans. Inf. Syst., 36(1), June 2017. ISSN 1046-8188. doi: 10.1145/3052775. URL https://doi.org/10.1145/3052775.

[71] Masumi Shirakawa, Takahiro Hara, and Shojiro Nishio. Idf for word n-grams. ACM Transactions on Information Systems, 36(1):5:1–5:38, 2017.

[72] Forrest J Shull, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. Empirical software engineering, 13 (2):211–218, 2008.

[73] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. A survey of self-admitted technical debt. Journal of Systems and Software, 152:70 – 82, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.02.056. URL http://www.sciencedirect.com/science/article/pii/S0164121219300457.

[74] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. A survey of self-admitted technical debt. Journal of Systems and Software, 152:70–82, 2019.

[75] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. Does technical debt lead to the rejection of pull requests? In: Proceedings of the 12th Brazilian Symposium on Information Systems, ser. SBSI '16, pages 248–254, 2016.

[76] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, pages 1631–1642, 2013.

[77] R.O. Spinola, N. Zazworka, A. Vetró, C. Seaman, and F. Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013.

[78] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In 2013 21st International Conference on Program Comprehension (ICPC), pages 83–92, 2013. doi: 10.1109/ICPC.2013.6613836.

[79] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 251–260, 2008.

[80] Pannavat Terdchanakul, Hideaki Hata, Passakorn Phannachitta, and Kenichi Matsumoto. Bug or not? bug report classification using n-gram idf. In 2017 IEEE international conference on software maintenance and evolution (ICSME), pages 534–538. IEEE, 2017.

[81] Mike Thelwall, Kevan Buckley, Georgios Paltoglou, Di Cai, and Arvid Kappas. Sentiment strength detection in short informal text. J. Am. Soc. Inf. Sci. Technol., 61(12):2544–2558, December 2010.

[82] Parastou Tourani, Yujuan Jiang, and Bram Adams. Monitoring sentiment in open source mailing lists: Exploratory study on the apache ecosystem. In Proceedings of 24th Annual International Conference on Computer Science and Software Engineering, CASCON '14, page 34–44, USA, 2014. IBM Corp.

[83] Anthony J Viera and Joanne M Garrett. Understanding interobserver agreement: the kappa statistic. Family Medicine, 37(5):360–363, 2005.

[84] Supatsara Wattanakriengkrai, Rungroj Maipradit, Hideaki Hata, Morakot Choetkiertikul, Thanwadee Sunetnanta, and Kenichi Matsumoto. Identifying design and requirement self-admitted technical debt using n-gram idf. In Proc. of 9th IEEE International Workshop on Empirical Software Engineering in Practice (IWESEP 2018), pages 7–12, 2018.

[85] Supatsara Wattanakriengkrai, Rungroj Maipradit, Hideki Hata, Morakot Choetkiertikul, Thanwadee Sunetnanta, and Kenichi Matsumoto. Identifying design and requirement self-admitted technical debt using n-gram idf. In 2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP), pages 7–12, 2018. doi: 10.1109/IWESEP.2018.00010.

[86] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. Examining the impact of self-admitted technical debt on software quality. In 2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER), volume 1, pages 179–188. IEEE, 2016.

[87] David A Wheeler. Sloccount user's guide, 2004.

[88] Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. Beyond the code: Mining self-admitted technical debt in issue tracker systems. arXiv preprint arXiv:2003.09418, 2020.

[89] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. Automating change-level self-admitted technical debt determination. IEEE Transactions on Software Engineering, 45(12):1211–1229, 2018.

[90] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta. Recommending when design technical debt should be self-admitted. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 216–226, 2017.

[91] F. Zampetti, A. Serebrenik, and M. Di Penta. Was self-admitted technical debt removal a real removal? an in-depth perspective. In 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pages 526–536, 2018.

[92] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. Recommending when design technical debt should be self-admitted. In Proceedings of the International Conference on Software Maintenance and Evolution, pages 216–226, 2017.

[93] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Was self-admitted technical debt removal a real removal? an in-depth perspective. In Proceedings of the International Conference on Mining Software Repositories, pages 526–536, 2018.

[94] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. A case study on effectively identifying technical debt. In Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, EASE '13, pages 42–47, 2013.

[95] Y. Zhang and D. Hou. Extracting problematic api features from forum discussions. In 2013 21st International Conference on Program Comprehension (ICPC), pages 142–151, 2013.