DebtHunter: A Machine Learning-based Approach for Detecting Self-Admitted Technical Debt

IRENE SALA, University of Milano Bicocca, Italy
ANTONELA TOMMASEL, ISISTAN CONICET/UNCPBA, Argentina
FRANCESCA ARCELLI FONTANA, University of Milano Bicocca, Italy

Due to limited time, budget or resources, a team is prone to introduce code that does not follow the best software development practices. This code that introduces instability in the software projects is known as Technical Debt (TD). Often, TD intentionally manifests in source code, which is known as Self-Admitted Technical Debt (SATD). This paper presents `DebtHunter`, a natural language processing (NLP)- and machine learning (ML)- based approach for identifying and classifying SATD in source code comments. The proposed classification approach combines two classification phases for differentiating between the multiple debt types. Evaluations over 10 open source systems, containing more than 259k comments, showed that the approach was able to improve the performance of others in the literature. The presented approach is supported by a tool that can help developers to effectively manage SATD. The tool complements the analysis over Java source code by allowing developers to also examine the associated issue tracker. `DebtHunter` can be used in a continuous evolution environment to monitor the development process and make developers aware of how and where SATD is introduced, thus helping them to manage and resolve it.

Additional Key Words and Phrases: technical debt, self-admitted technical debt, machine learning, natural language processing

ACM Reference Format:
Irene Sala, Antonela Tommasel, and Francesca Arcelli Fontana. 2021. DebtHunter: A Machine Learning-based Approach for Detecting Self-Admitted Technical Debt. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3463274.3464455

1 INTRODUCTION

Intentionally or unintentionally a team is prone to introduce code that does not follow the best software development practices. These pieces of code may introduce technical debt (TD) [4], which occurs when developers trade-off short-term benefits with long-term stability, implementing non-optimal-solutions. These non-optimal solutions can refer to quick hacks, technological gaps, and wrong architectural or structural choice [12] leading to a rise in software maintainability issues. For example, if implementing a new functionality requires three days, in the presence of lack of internal quality it may take up to five days [7]. These additional days are the *interest* of the TD Detecting TD is not a simple task and in the current literature, there is no technique that can guarantee a 100% reliability in the identification of every part of the code that generates or will generate a debt. Technical Debt can manifest at different abstraction levels, such as code, design, architecture, and test [1]. For example, developers write comments to inform the rest of the team that the current implementation is not optimal and needs future maintenance. This particular type of debt intentionally introduced in source code by developers is known as Self-Admitted Technical Debt (SATD) [17]. Although there is no clear relationship between SATD and defects, files with SATD were found to present more bug-fixing changes than files without SATD [21]. Moreover, changes on SATD defects are more difficult to perform than on regular ones [19].

Assessing SATD has many advantages when compared to the traditional approach of only considering TD in source code: is more lightweight (it does not construct a representation of source code like Abstract Syntax Trees) and source code comments can be easily extracted, for example, using regular expressions. In addition, SATD is considered a more effective indicator of debt than that detected in source code because SATD represents an explicit indicator of debt introduced by the developers. Even though alternatives to identify SATD with manual or automated techniques have been proposed in the literature, most approaches are limited to binary classification (SATD or not SATD) rather than discriminating the debt type, and restrict the analysis to source code comments.

In this work, we propose `DebtHunter`, an approach to automatically detect the five most common SATD types by investigating the adequacy of Natural Language Processing (NLP)- and Machine Learning (ML)-based approaches for this task. The approach is embedded in a tool to facilitate its usage in a real software development process. The main contributions of this study are the following:

(1) `DebtHunter`, an automatic SATD detection approach that exploits text mining and machine learning techniques.
(2) The materialization of `DebtHunter` into a tool that could be used as part of a continuous evolution environment to monitor the development process and make developers aware of how and where SATD is introduced, thus helping them to manage and resolve it. The tool can analyse both source code and the corresponding issue tracker.

The remainder of this paper is organized as follows. Section 2 reviews the field of SATD and how to detect it. Section 3 motivates and describes the approach. Section 4 describes `DebtHunter` as a tool. Section 5 presents the study design and the experimental settings for evaluating the presented approach. Section 6 describes the observed results and their threats of validity. Section 8 reports the conclusions and outlines future work.

2 RELATED WORK

This Section reviews related works on the SATD detection topic: first, studies that perform SATD detection based on manually defined indicators and patterns, then, studies based on ML techniques.

2.1 Pattern-based Approaches

Potdar and Shihab [17] proposed the first approach for TD detection by identifying recurrent patterns expressing debt in source code comments, which were considered manifestations of SATD. Based on the manual classification of $101k$ comments, the authors defined 63 textual patterns indicating the presence of debt. da S. Maldonado and Shihab [5] introduced five heuristics to eliminate comments not expressing debt: removing license comments, aggregating consecutive single-line comments, removing automatically generated comments by the IDE, removing commented source code, and removing Javadoc comments. The authors manually classified $33k$ comments from five open source projects into the five debt types proposed by Alves and et al. [1]:

Defect debt: refers to known defects, bugs, unexpected behaviour of the code, or failures. Example "*Bug in the above method*" from Apache JMeter.

Design debt: refers to debt that can be discovered by source code analysis and identifying violations of the object-oriented design principles. Example "*/*TODO: really should be a separate class */*" from ArgoUML.

Documentation debt: refers to issues found in software project documentation, i.e. comments that express missing, outdated, inadequate, or incomplete documentation. Example "***FIXME** This function needs documentation*" from Columba.

Requirement debt: refers to trade-offs for what requirements need to implement or how to implement them; a method, class or program that presents an incompleteness. Example "*//TODO no methods yet for getClassname*" from Apache Ant.

Test debt: refers to issues found in testing activities that can affect the quality of those activities. Example "*//TODO enable come proper tests!!*" from Apache JMeter.

Bavota and Russo [2] replicated the work of Potdar and Shihab [17] to analyse the propagation and evolution of TD over 159 software projects. The authors concluded that fixing TD does not necessarily imply that the comments manifesting it will be removed. Instead, such comments can survive for over $1,000$ commits. More recently Guo and et al. [8] proposed to identify SATD comments by matching them with one of four task tags that can express debt: `TODO`, `FIXME`, `XXX`, and `HACK`. According to the authors, no labeled data and manually checking are required for this approach.

Pattern-based approaches present some limitations. First, they require analysing thousands of comment lines. Second, detected patterns might not generalize well over different projects, and are in constant evolution. In this sense, discovering the complete set of patterns expressing debt would require manually analysing a great number of projects, which is impractical. Furthermore, the available data, as well as being limited, will soon become obsolete. Lastly, as patterns only refer to the general presence of debt, and do not provide support for the identification of the different TD types.

2.2 Machine Learning Approaches

da S. Maldonado et al. [6] proposed a ML-based approach for automatically identifying design and requirement SATD. The authors also presented an extended dataset including $259k$ comments from 10 Java projects. As in their previous approach, the goal was to identify the most frequent five debt types. Unlike da S. Maldonado et al. [6], who trained their model based on the comments extracted from all the considered systems, Huang and et al. [9] trained a model for each selected Java project and combined their results using a classifier voting method. Comments were classified into one of six classes: defect, documentation, test, design, requirement, and non-SATD. Liu and et al. [13] proposed a binary classification of SATD comments using NLP (i.e., Word Tokenizer, a custom stopWords list, Porter stemmer, and tf-idf), feature selection (the top 10% of features were selected according their Information Gain score), and a Naïve Bayes Multinomial (NBM) classifier. The authors also presented *SATD Detector*, a tool for binary classifying source code comments. Ren and et al. [18] proposed a deep learning approach based on a Convolutional Neural Network (CNN), where the convolutional layer is used to extract the most informative features for representing the data (in this case, the source code comments). Such features were later used to differentiate between SATD and not-SATD, based on the dataset labeled by [5]. According to the authors, their approach outperformed the results of Huang and et al. [9], both in terms of within-project and cross-project prediction.

Most of the considered ML-based approaches perform a binary analysis of SATD (i.e. they separate SATD from non-SATD comments), and thus the particular types of debt are not identified. Moreover, existing approaches (both pattern-based and ML) are only based on analysing source code comments, disregarding the importance of other information sources, such as issue trackers. In this context, `DebtHunter` approach allows to analyze both, source code comments and issue tracker.

3 APPROACH

The main goal of `DebtHunter` approach is the identification and classification of Java *source code comments* into SATD types. In this context, we were interested in comparing `DebtHunter` to other techniques for classifying SATD in the literature, and also assessing the impact of ML mechanisms such as feature selection and data sampling on classification performance.

Figure 1 presents the comment processing workflow. To accomplish this task, comments undergo a simple data cleaning process, text preprocessing, data sampling, and feature selection (Steps 2 to 6). Then, a classification model assigns labels to the comments accordingly to the debt type expressed by them (Step 7). Unlike other approaches in the literature, `DebtHunter` implements a two-step classification phase: SATD identification and SATD classification into specific debt types. Finally, the trained models were evaluated with their respective test sets (Step 8, in the coloured rectangle).

3.1 Text Preprocessing

The comments retrieved in Step 1 are cleaned by removing punctuation marks, and empty and duplicated comments (Step 2). Then, they are divided into two training sets (the fork of Step 3). The first one includes all comments labeled as SATD or non-SATD, while the second set only includes comments labeled as SATD and labels them according to the actual debt type they represent. As text cannot be used as input of the classification models, both training sets undergo a classical NLP pre-processing pipeline [10] (Step 4). First, text is tokenized (i.e. it is divided into its words or tokens). Second, stopwords are removed following a *Rainbow* [15] based list. Third, *Lovins stemmer* [14] is used to reduce the lexical variation of word. Finally, the remaining words were weighted using *tf-idf* [11]. Following those pre-processing steps, $13,056$ features were obtained for the first training set and $3,363$ features for the second one. As observed, the first training set comprises more instances, and thus more features than the second one.

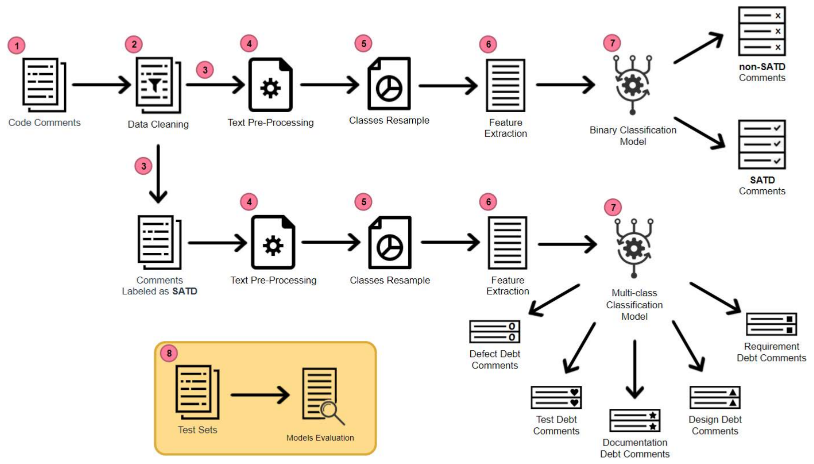

Fig. 1. DebtHunter approach Workflow

3.2 Data Sampling

Generally, ML algorithms achieve their best performance when trained over balanced data, while achieving lower performance over unbalanced data. Thus, data resampling methods were applied to both training sets (Step 5). In the case of the first training set, data ware resampled based on the *Spread Subsample*[1] technique. On the other hand, for the second dataset, it was applied an oversampling technique (called *SMOTE* [3]).

As SMOTE relies on creating synthetic instances, it could introduce false or noisy information. Nonetheless, in presence of slightly unbalanced data (as in the second classification step), this method yields good performance. When the dataset presents a heavy data balance problem, the SMOTE algorithm is not able to generate synthetic instances due to the huge heap space required. For this reason, the SMOTE technique can be applied only for the second step's training set, which has only 2.232 instances, versus 25.740 for the first step's training set. Spread SubSample can be applied regardless of the severity of the data balanced problem. It works by removing the excessive records of the majority classes, which could potentially lead to the loss of information.

3.3 Feature Selection

The pre-processing steps generate a large number of features that yield low frequency, and might not carry relevant information. This situation is known as the "curse of dimensionality" and refers to the increasing computational complexity of learning as the data volume grows regarding the space dimension, but becomes sparser. To overcome these problems, feature selection techniques (based on Information Gain) were applied on both training sets (Step 6).

3.4 Classification

DebtHunter follows a two-step classification model (Step 7): (1) a binary classification setting for identifying SATD comments (trained with the first training set), (2) a multi-class classification setting for classifying SATD comments into the debt type they express (trained with the second training set). Classification focuses on the five debt types proposed by [1] described in Section 2. In other words, the first step separates between comments either expressing SATD or not, while in the second step, discovered SATD comments are classified into their types. SMO (Sequential Minimal Optimization) [16] was used for both classification steps.

[1]https://weka.sourceforge.io/doc.dev/weka/filters/supervised/instance/SpreadSubsample.html

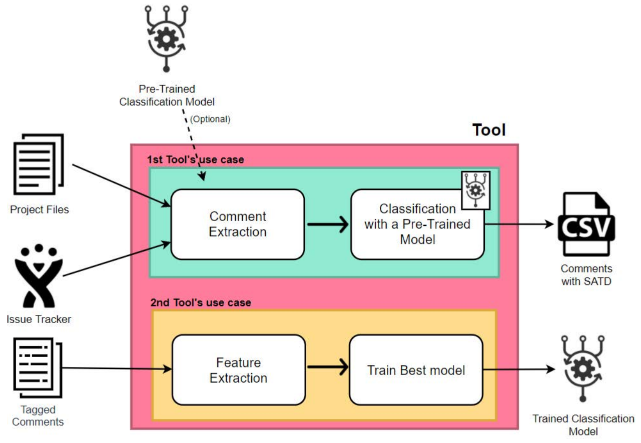

Fig. 2. DebtHunter tool schema.

4 DEBTHUNTER TOOL

To allow developers to more effectively manage SATD, the proposed approach is embedded in the `DebtHunter` tool[2]. Figure 2 represents the tool schema. The overall tool procedures are included in the pink box. The tool aims at helping developers and practitioners to comprehend and remove the SATD in their projects, by offering two use cases:

(1) Comment labelling (represented by the green box). It receives input from the *Java project files* or *issue tracker*. The outputs correspond to the extracted comments and their associated labels. Classification is performed based on the best `DebtHunter` pre-trained model.

(2) New model training (represented by the yellow box). It receives as input labeled comments, which are used for training a new model. The output is the classification model learnt from the input data.

5 STUDY DESIGN

To evaluate the performance of the proposed approach to identify the SATD types, we propose three research questions.

RQ1: ***Can* `DebtHunter` *outperform other SATD detection techniques in the literature?*** In this RQ we aim at finding whether `DebtHunter` can improve the performance of other approaches in the literature and open the way towards new research directions.

RQ2: ***Can feature selection improve the performance of SATD detection?*** This RQ aims at analysing the impact of feature selection techniques on the performance of `DebtHunter`. In particular, how the quality of classification varies according to the selected number of features for training the models.

RQ3: ***Can data sampling improve the performance of SATD classification?*** Given the unbalanced nature of data, this RQ aims to analyse whether classification performance is affected by resampling techniques.

5.1 Experimental Settings

Weka[3] was chosen to support the Java implementation of `DebtHunter`. SMO hyper-parameters were optimized using *Grid Search*. The performance of the trained models was evaluated based on a *stratified ten-fold cross-validation*. For all performed evaluations, the training and the validation sets comprise the same set of records. Results were evaluated in terms of Precision, Recall, and F-Measure. To check the statistical significance of result differences, the Wilcoxon signed-rank test [20] at 95% significance level was used.

[2]https://github.com/PandaMinore/DebtHunter-Tool
[3]https://www.cs.waikato.ac.nz/ml/weka/

Table 1. Characterization of the analyzed projects.

Project	Project details					Comments details		Technical debt details		
	Relase	# of classes	Domain	SLoC	# of contributors	# of comments	# of TD comments	% of design debt	% of requirement debt	% of other debt
Apache Ant	1.7.0	1,475	Java library & Command-line Tool	115,881	74	21,587	4,137	72.51	09.92	17.55
ArgoUML	0.34	2,609	UML Modeling Tool	176,839	87	67,716	9,548	56.68	29.08	14.22
Columba	1.4	1,711	E-mail Client	100,200	9	33,895	6,478	61.76	21.07	17.15
Eclipse EMF	2.4.1	1,458	Eclipse Model Driven Architecture	228,191	30	25,229	4,401	75.00	15.38	09.61
Hibernate	3.3.2.GA	1,356	ORM Framework	173,467	226	11,630	2,968	75.21	13.55	11.22
JEdit	4.2	800	Text Editor	88,583	57	16,991	10,322	76.56	05.46	17.96
JFreeChart	1.0.19	1,065	Char Library	132,296	19	23,474	4,423	88.03	07.17	04.78
JMeter	2.10	1,181	Performance Tester	81,307	33	20,084	8,162	84.49	05.61	09.89
JRuby	1.4.0	1,486	Ruby Interpreter	150,060	328	11,149	4,897	55.14	17.68	27.17
SQuirreL	3.0.3	3,108	SQL Client	215,234	46	27, 474	7,230	73.07	17.48	09.44
Average		1,625		146,206	91	25,923	6,257	71.84	14.24	13.89
Total		16,249		1,462,058	909	259,229	62,566	-	-	-

5.2 Dataset

All evaluations were based on the code comment collection created and labeled by da S. Maldonado and Shihab [5]. The collection includes 259k labeled source code comments belonging to 10 open source systems: `Apache Ant`, `Apache JMeter`, `ArgoUML`, `Columba`, `Eclipse EMF`, `Hibernate`, `JEdit`, `JFreeChart`, `JRuby`, `SQuirreL`. Each comment is associated to a label that can be non-SATD or a SATD type (i.e., defect, design, documentation, requirement, test). Issue tracker comments were not considered for the training and evaluation of `DebtHunter`. Table 1 summarizes the characteristics of the dataset.

5.3 Baselines

To verify the effectiveness of `DebtHunter`, its performance was compared with other baselines in the literature:

Baseline 1 (pattern-based): Potdar and Shihab [17] proposed a pattern-based approach for detecting SATD comments in four projects. A comment was considered to express debt if it matched any of the 63 defined patterns.

Baseline 2 (Binary): Most approaches in the literature (e.g. [13, 17]) propose only identifying SATD instances, regardless of their type. This approach implements a binary classification model based on SMO.

Baseline 3 (All-labels): Other approaches in the literature have proposed a multi-class model to identify SATD debt types. This baseline follows the same idea based on the LibSVM algorithm.

Baseline 4 (Liu2018): Liu et al. [13] proposed a binary classification approach based on training an NBM classifier for each analyzed project.

Baseline 5 (MAT): Guo et al. [8] proposed a pattern-based approach for detecting SATD comments. A comment was considered SATD if it matched any of `TODO`, `FIXME`, `XXX`, and `HACK` words.

6 EVALUATION

This section reports the performance of `DebtHunter` to answer the proposed research questions. To answer the RQs, different configurations of *Raw* `DebtHunter` will be explored aiming at improving its performance.

6.1 RQ1: Can `DebtHunter` outperform other SATD detection techniques in the literature?

To assess the usefulness of `DebtHunter`, its performance needs to be compared to that of other approaches in the literature. In this regard, performance is compared with the selected baselines: Baseline 1 (*pattern-based*), Baseline 2 (*Binary*), Baseline 3 (*All-labels*), Baseline 4 (*Liu2018*), and Baseline 5 (*MAT*). For this initial evaluation, we consider a "raw" `DebtHunter` configuration in which data are cleaned and pre-processed, but no resampling nor feature selection is performed.

Table 2. *Raw* `DebtHunter` compared to baselines performance.

Metrics	Raw DebtHunter	Internal Baselines		External Baselines		
		Binary	All-labels	Liu2018	pattern-based	MAT
Precision	**0.972**	0.897	*0.945*	0.583	0.746	0.882
Improv.	-	+8.36%	+2.86%	+66.72%	+30.29%	+10.20%
Recall	**0.967**	0.777	*0.952*	0.767	0.052	0.766
Improv.	-	+24.45%	+26.03%	+26,08%	+1759.6%	+26.24%
F1-Score	**0.965**	0.833	*0.947*	0.662	0.098	0.820
Improv.	-	+15.85%	+1.90%	+45.77%	+884.69%	+17.68%

Table 3. `DebtHunter` first step varying the number of selected features.

Metrics	all	200	300	400	500	750	1k	1.5k	2k	2.5k	3k	5k	7.5k	10k
Precision	**0.897**	*0.895*	*0.895*	0.893	0.893	0.892	0.893	0.890	0.891	0.891	0.891	0.892	0.890	*0.895*
Improv.	-	+0, 223%	+0, 223%	+0.448%	+0.448%	+0.561%	+0.448%	+0.787%	+0.673%	+0.673%	+0.673%	+0.561%	+0.787%	+0, 223%
Recall	0.777	0.789	0.789	0.791	0.793	*0.794*	**0.796**	0.793	0.793	0.793	0.793	*0.794*	0.793	0.785
Improv.	-	−1.52%	−1.52%	−2.57%	−2.02%	−2.14%	−2.39%	−2.02%	−2.02%	−2.02%	−2.02%	−2.14%	−2.02%	−1.02%
F1-Score	0.833	0.838	0.838	0.839	*0.840*	**0.841**	0.796	0.839	*0.840*	*0.840*	*0.840*	**0.841**	0.839	0.836
Improv.	-	−0.597%	−0.597%	−0.715%	−0.833%	−0.951%	+4.65%	−0.715%	−0.833%	−0.833%	−0.833%	−0.951%	−0.715%	−0.359%

Table 4. `DebtHunter` second step varying the number of feature selected performance.

Metrics	all	200	300	400	500	750	1k	1.5k	2k	2.5k	3k
Precision	0.698	0.673	0.685	0.674	0.684	0.668	0.692	0.688	**0.705**	0.690	*0.703*
Improv.	-	+3.72%	+1.90%	+3.56%	+2.05%	+4.49%	+0.867%	+1.45%	−0.993%	+1.16%	−0.711%
Recall	*0.729*	0.722	0.723	0.715	0.716	0.712	0.721	0.723	0.727	0.725	**0.731**
Improv.	-	+0.970%	+0.830%	+1.96%	+1.82%	+2.39%	+1.11%	+0.830%	+0.275%	+0.552%	−0.274%
F1-Score	0.661	**0.663**	*0.662*	0.651	0.647	0.647	0.631	0.652	0.653	0.647	**0.663**
Improv.	-	−0.302%	−0, 151%	+1, 54%	+2, 16%	+2, 16%	+4.75%	+1, 38%	+1, 23%	+2.16%	−0, 302%

Table 2 shows the approaches performance. In terms of Precision, Recall and F1-Score, *Raw* `DebtHunter` outperformed all selected baselines. *Raw* `DebtHunter` and *All-labels* achieved similar results. *Pattern-based* achieved medium to high Precision, but very low Recall, which also affects F1-Score. Medium-high precision means that the majority of comments labeled as debt were actually expressing debt, while at the same time, most of the comments expressing debt were missed. *Pattern-based* identified as SATD comments that either did not express debt or expressed a design debt. Only one defect comment was labeled as SATD. This could be due to patterns having a limited scope, as they heavily rely on the words and expressions used in the analyzed systems. The performed statistical analysis showed that differences were statistically significant, confirming the superiority of `DebtHunter` over the selected baselines.

6.2 RQ2: Can feature selection improve the performance of SATD classification?

Feature selection can be useful to reduce the number of features when there is a large number of features in the training set, and not every feature provides relevant information. In this context, it is necessary to analyse the usefulness of feature selection. For answering this RQ we enriched *Raw* `DebtHunter` by performing Feature Selection. As the two classification steps rely on different training data, they might require different sets of features. Hence, they were separately analyzed. For the first training set, after pre-processing approximately $13k$ features were obtained. Hence, scenarios including 200, 300, 400, 500, 750, $1k$, $1.5k$, $2k$, $3k$, $5k$, $7.5k$, $10k$ and all features in the training set were considered. Instead, for the second training set, approximately $3k$ features were obtained after pre-processing. Hence, for this step, scenarios including between 200 to $3k$ and all features in the training set features were considered. Features were selected according to their Information Gain score.

Table 3 shows the performance for the first classification step when varying the number of selected features. The *all* column presents the results of *Raw* `DebtHunter` without applying Feature selection. The highest Precision was obtained for *all*. Instead, the best Recall was obtained when selecting $1k$ features. Finally, the highest F1-Score was achieved when selecting either 750 or $5k$ features. No statistically significant differences were observed between *all* and 200, 300, 400, 500, 750, $1k$, $1.5k$, $2k$, $3k$, $5k$ configurations. Based on the F1-Score results, in the selection of the best approach, it is important to consider the effect of the number of selected features over computational complexity and training time. As smaller feature sets lead to lower computational complexity and thus, lower training times, it is preferable to apply feature selection techniques to reduce the original feature set, and only keep 750 features.

Table 5. `DebtHunter` first step varying the resample method.

Metrics	no R	SS 1:1	SS 1:5	SS 1:10
Precision	**0.892**	*0.686*	0.672	**0.892**
Improv.	-	+30.03%	+32.74%	-
Recall	0.794	**0.885**	*0.869*	0.795
Improv.	-	−10.28%	−8.63%	−0.126%
F1-Score	**0.841**	*0.773*	0.758	**0.841**
Improv.	-	+8.80%	+10.95%	-

Table 6. `DebtHunter` second step varying the resample method.

Metrics	no R	SS 1:1	SS 1:5	SS 1:10	SM50	SM100	SM150
Precision	0.668	0.660	0.681	0.693	0.696	**0.700**	*0.699*
Improv.	-	+1.21%	−1.91%	−3.61%	−4.02%	−4.57%	−4.43%
Recall	0.712	0.342	0.508	0.579	**0.729**	*0.722*	0.709
Improv.	-	+108.19%	+40.16%	+22.97%	−2.33%	−1.39%	−0.42%
F1-Score	0.647	0.350	0.541	0.608	0.692	**0.704**	*0.701*
Improv.	-	+84.86%	+19.59%	+6.41%	−6.50%	−8.10%	−7.70%

Table 4 shows the performance obtained by the second classification step when varying the number of selected features. The highest Precision was observed when considering $2k$ features. On the other hand, selecting $3k$ features achieved the highest Recall and F1-Score. The highest F1-Score was obtained when considering 200 features. Based on the observed results, features seemed to carry more relevance than for the first step, leading to the selection of almost the full set of features. In particular, selecting the best $3k$ features achieved the highest Precision and F1-score

6.3 RQ3: Can data sampling improve the performance of SATD classification?

To answer this RQ, oversampling (i.e., SMOTE) and undersampling (i.e., Spread Subsample) techniques were applied to `DebtHunter`. Starting from the best configuration found for RQ2, the effectiveness of applying resampling methods to training data is studied. The two classification steps are evaluated separately. Several sampling configurations were analyzed. For the first classification step, the configurations were: no Resample method (*no R*), Spread Subsample with maximum ratio 1:1 (*SS 1:1*), 1:5 (*SS 1:5*), and 1:10 (*SS 1:10*). On the other hand, for the second classification step, SMOTE configurations were also included, adding 50% (*SM50*), 100% (*SM100*) and 150% synthetic instances (*SM150*).

Table 5 presents the results for the first classification step. The *no R* configuration outperformed *SS 1:1* and *SS 1:5* in terms of Precision and F1-Score. However, *no R* achieved the worst Recall. *SS 1:1* achieved the highest Recall but a medium Precision. High Recall implies a low number of false negatives (few SATD comments labeled as non-SATD), while a medium value of Precision means a large number of false positives (a lot of non-SATD comments are labeled as SATD). In this case, the classifier is not able to distinguish SATD with high Precision. The same problem was observed for the *SS 1:5*. Even though *no R* and *SS 1:10* achieved very similar results, it is preferable to select *SS 1:10* as it reduces the model complexity, which translates in lower training times.

Table 6 presents the results for the second classification step. The best Precision and F1-Score, were obtained with the *SM100* configuration. Similarly, *SM50* achieved the highest Recall. In general, the Spread Subsample method performed worse than SMOTE. Given that the training set was slightly unbalanced, removing instances led to the loss of information as it removed a large number of comments, thus leaving few training examples for the classifier to learn. In this sense, in this case, it could be useful to create a few new synthetic instances to improve performance.

Although it slightly increased the complexity of the model when compared to the *no R* configuration, *SM50* allowed to significantly improve classification performance without introducing overfitting, thus it selected as the best performing alternative.

6.4 Summary

The **first RQ** aimed at assessing whether `DebtHunter` was able to improve other approaches in the literature. Based on the experimental evaluation, we can conclude that `DebtHunter` was able to significantly outperform the selected baselines.

The **second RQ** assessed the usefulness of feature selection for the SATD detection task. The performance of `DebtHunter` improved when applying feature selection. Particularly, the best configuration of the first classification step included 750 features, while the best

Table 7. `DebtHunter` and All-labels classes performance.

Approach	Metrics	Test	Req.	Design	Defect	Doc.	non-SATD
DebtHunter	Precision	0.875	0.515	**0.757**	0.483	**0.909**	**0.980**
	Recall	**0.656**	**0.338**	**0.920**	**0.125**	**0.588**	0.990
	F1-Score	**0.750**	**0.408**	**0.830**	**0.199**	**0.714**	**0.985**
Raw DebtHunter	Precision	**0.905**	**0.595**	0.732	**0.538**	**0.909**	0.978
	Recall	0.594	0.162	0.255	0.063	0.588	**0.991**
	F1-Score	0.717	0.255	0.732	0.112	0.714	0.984
All-labels	Precision	0.857	0.463	0.663	0.375	0.818	0.979
	Recall	0.563	0.162	0.735	0.134	0.529	0.990
	F1-Score	0.697	0.240	0.697	0.197	0.643	**0.985**

configuration of the second classification step included $3k$ features. In both cases, the computational complexity of the models was reduced while increasing their performance, thus showing the benefits of performing feature selection.

Finally, based on the unbalanced nature of data, the **third RQ** assessed that `DebtHunter` could benefit using resampling methods. In particular, the best results were obtained when using Spread Subsample (with a maximum class ratio of 1:10) for the first classification step and SMOTE (adding 50% synthetic instances) for the second one.

As Table 7 shows, for each debt type the final configuration of `DebtHunter` outperformed *All-labels* and *Raw* `DebtHunter`. The comparison shows that `DebtHunter` improved the balance between Precision and Recall by discovering a higher proportion of debt comments than *All-labels* and *Raw* `DebtHunter`. In particular, performing feature selection and applying data sampling techniques allowed to significantly improve the classification performance of the Requirements and Defect debt types, which achieved the lowest results for *Raw* `DebtHunter`. In summary, these results highlight the usefulness of `DebtHunter` for debt type classification.

7 THREATS TO VALIDITY

According to the presented results, some factors might have biased our study.

Internal Validity is the degree to which the variables measure correctly the actual phenomenon. In this study, we used the dataset provided by da S. Maldonado and Shihab [5], where code comments were manually labeled into five debt types. Even though this inspection was made by experts in the software engineering field, comments could be miss-classified due to syntactic ambiguity and personal bias. Increasing the number of labeled comments in the dataset could reduce the impact of this threat. Besides, a manual re-labelling of comments could be performed.

External Validity considers the degree of generalization of our findings. The dataset used comprises comments belonging to 10 open-source projects selected from different domains. Nonetheless, `DebtHunter` might not adequately generalize to other projects with dissimilar characteristics or with different policies for expressing SATD. The impact of this threat could be mitigated by replicating the study across a wider variety of projects, including projects from multiple organizations, and even commercial projects.

Construct Validity concern the relation between the theory and the observation. A threat could arise from the NLP pipeline, including also the resampling strategy for dealing with the unbalanced nature of the dataset. Although we carefully configured the pipeline to avoid overfitting and mitigate this threat, a slightly different configuration of the pipeline could lead to different performance results.

8 CONCLUSIONS

In this paper, we presented `DebtHunter`, an NLP- and ML-based approach, for detecting and identifying SATD in source code comments. This approach is embedded into a tool for helping developers effectively manage the SATD in their projects. The tool provides developers with the capability of classifying either source code comments or issue tracker comments, and with the possibility of training new classification models to learn the particularities of the systems to be analyzed. The answers to the research questions highlighted that `DebtHunter` was able to improve the performance of the selected baselines. The performed evaluations showed the importance of a two-step classification model (RQ1), of applying feature selection based on Information Gain (RQ2) and using class sampling methods (RQ3). In summary, results showed that `DebtHunter` can be useful for easily and accurately detecting SATD.

We envision several aspects to be tackled as future works. First, extending the evaluation to other systems and validate the performed classifications and usefulness of the tool with developers. Second, applying semi-supervised models, saving the data labeled in the first tool's use case and merge them with the da S. Maldonado and Shihab [5] dataset. Third, extending the training sets, allowing the trained models to adapt to comment changes and evolution. Fourth, even though `DebtHunter` allows developers to analyze issue tracker comments, they were not included in the evaluation of the approach, as labeled datasets are not easily nor broadly available. In this regard, it would be desirable to create a newly labeled collection including issue trackers comments of, at least, the same systems analyzed by da S. Maldonado and Shihab [5] to assess the descriptive power of issue trackers regarding SATD and improve the `DebtHunter` model. Fifth, using deep learning techniques (such as word embeddings) to semantically represent comments and thus reduce the dependency on lexical models. This would allow a richer comment representation that could capture their semantic particularities.

REFERENCES

[1] Nicolli S.R. Alves and et al. 2014. Towards an Ontology of Terms on Technical Debt. In *2014 Sixth International Workshop on Managing Technical Debt*. IEEE. https://doi.org/10.1109/mtd.2014.9

[2] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16, ACM Press*. ACM Press. https://doi.org/10.1145/2901739.2901742

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research* (2002). https://doi.org/10.1613/jair.953

[4] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* (1993). https://doi.org/10.1145/157710.157715

[5] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE. https://doi.org/10.1109/mtd.2015.7332619

[6] Everton da S. Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering* (2017). https://doi.org/10.1109/tse.2017.2654244

[7] Martin Fowler. 2019. TechnicalDebt. https://martinfowler.com/bliki/TechnicalDebt.html Accessed March 2021.

[8] Zhaoqiang Guo and et al. 2019. MAT: A simple yet strong baseline for identifying self-admitted technical debt. *arXiv preprint arXiv:1910.13238* (2019).

[9] Qiao Huang and et al. 2017. Identifying Self-Admitted Technical Debt in Open Source Projects using Text Mining. *Empirical Software Engineering* (2017). https://doi.org/10.1007/s10664-017-9522-4

[10] Nitin Indurkhya and Fred J. Damerau. 2010. *Handbook of Natural Language Processing* (2nd ed.). Chapman & Hall/CRC.

[11] Karen Sparck Jones. 1972. A Statistical Interpretation of Term Specificity and its Application in Retrieval. *Journal of Documentation* (1972). https://doi.org/10.1108/eb026526

[12] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* (2012). https://doi.org/10.1109/ms.2012.167

[13] Zhongxin Liu and et al. 2018. SATD Detector: A text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. ACM. https://doi.org/10.1145/3183440.3183478

[14] Julie Beth Lovins. 1968. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics* 11 (1968), 22–31.

[15] Andrew Kachites McCallum. 1996. Rainbow stopwords. http://www.cs.cmu.edu/~mccallum/bow/rainbow/ Accessed March 2021.

[16] J. Platt. 1998. Fast Training of Support Vector Machines using Sequential Minimal Optimization. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press. http://research.microsoft.com/~jplatt/smo.html

[17] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. https://doi.org/10.1109/icsme.2014.31

[18] Xiaoxue Ren and et al. 2019. Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Transactions on Software Engineering and Methodology* (2019). https://doi.org/10.1145/3324916

[19] Sultan Wehaibi and et al. 2016. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. https://doi.org/10.1109/saner.2016.72

[20] Frank Wilcoxon. 1950. Some rapid approximate statistical procedures. *Annals of the New York Academy of Sciences* (1950).

[21] Meng Yan and et al. 2019. Automating Change-Level Self-Admitted Technical Debt Determination. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/tse.2018.2831232