



Predicting technical debt from commit contents: reproduction and extension with automated feature selection

Leevi Rantala¹  · Mika Mäntylä¹

Published online: 4 July 2020
© The Author(s) 2020

Abstract

Self-admitted technical debt refers to sub-optimal development solutions that are expressed in written code comments or commits. We reproduce and improve on a prior work by Yan et al. (2018) on detecting commits that introduce self-admitted technical debt. We use multiple natural language processing methods: Bag-of-Words, topic modeling, and word embedding vectors. We study 5 open-source projects. Our NLP approach uses logistic Lasso regression from Glmnet to automatically select best predictor words. A manually labeled dataset from prior work that identified self-admitted technical debt from code level commits serves as ground truth. Our approach achieves + 0.15 better area under the ROC curve performance than a prior work, when comparing only commit message features, and + 0.03 better result overall when replacing manually selected features with automatically selected words. In both cases, the improvement was statistically significant ($p < 0.0001$). Our work has four main contributions, which are comparing different NLP techniques for SATD detection, improved results over previous work, showing how to generate generalizable predictor words when using multiple repositories, and producing a list of words correlating with SATD. As a concrete result, we release a list of the predictor words that correlate positively with SATD, as well as our used datasets and scripts to enable replication studies and to aid in the creation of future classifiers.

Keywords Natural language processing · Latent Dirichlet allocation · Logistic regression · Word embeddings · Topic modeling · Data mining

✉ Leevi Rantala
leevi.rantala@oulu.fi

Mika Mäntylä
mika.mantyla@oulu.fi

¹ M3S / ITEE / University of Oulu, P.O.B. 4500, 90014 University of Oulu, Oulu, Finland

1 Introduction

Technical debt is a term that is used to depict non-optimal choices made in the software development process. Several types of technical debt have been identified including code debt, design and architectural debt, environmental debt, knowledge distribution and documentation debt, and testing debt (Yan et al. 2018). A study by Sas and Avgeriou (2019) shows that technical debt degrades maintainability. This occurs when developers prioritize other features, such as performance or functionality, over maintainability. Accumulation of technical debt has also other negative effects in long-term development besides worse maintainability, as it can lead to weakened morale, lower productivity, worse quality, and higher risks (Tom et al. 2013). Another study by Besker et al. (2018) reports that approximately 25% of developers total time is wasted due to technical debt, and in 24% of all cases when developers encountered technical debt, they are forced to introduce additional debt because of it.

Recently, researchers such as Potdar and Shihab (2014) have started to work on a subset of technical debt which they named *self-admitted technical debt* (SATD). This means that developers leave some kind of message either to source code comment or commit message, where they state the existence of technical debt, e.g., “Add a TODO to remind me to come back and look at this.”¹

SATD appearance on software systems has been studied on calculating the percentage of SATD code comments from all comments several different scales. Research done by da Silva Maldonado et al. (2017) looked at code comments from 10 projects, and discovered that 0.41–5.57% (median 1.27%) of all comments were SATD comments. Another paper by Maldonado E.D.S. and Shihab (2015) found out that 3.2–16.8% (median 4.6%) of all code comments were SATD comments. Even when the amount of SATD comments from overall comments is low, it can increase software complexity (Wehaibi et al. 2016). When introduced to the development, SATD can survive for a very long time (Bavota and Russo 2016), and gather interest (Kamei et al. 2016) by increasing the debt over time. It takes a lot of rework to remove SATD out of the system (Mensah et al. 2016).

Previous studies have researched SATD by analyzing commit metrics, like lines of code added and number of files changed (Yan et al. 2018), and source code comments where the debt is admitted (Huang et al. 2018). Previous work has so far left commit messages largely out of the picture, when predicting self-admitted technical debt appearance. Commit messages allow developers to describe the changes they have made into the code. These explanations are not necessarily just factual descriptions, and commit messages can contain also other information. They have been studied for variety of other reasons such as analyzing developers’ sentiments (Guzman et al. 2014; Sinha et al. 2016), for discovering security issues (Zhou Y and Sharma A 2017), and examining developer patterns for refactoring activities (AlOmar et al. 2019).

For self-admitted technical debt, the study by Yan et al. (2018) looked at code commit messages and analyzed the appearance of five different keywords (bug, feature, improve, document, refactor) and message length in commit messages to predict if that message related to self-admitted technical debt. They achieved an average area under curve ROC (AUC-ROC) performance of 0.57 while predicting through commit messages. Overall commit messages have not been inspected deeply with regard to self-admitted technical debt, even when the messages have been studied for other purposes to discover tacit

¹ Piece of a commit message from Tomcat.

information. In this work, we replicate and extend the work by Yan et al. (2018) by answering the following general research problem: “To what extent does the use of NLP with automated feature selection help to improve the prediction of self-admitted technical debt?”

We further divide the research problem into the following research questions:

- RQ1: What NLP technique predicts most accurately SATD from commit messages?
- RQ2: Can we improve with our models over previous work on message analysis?
- RQ3: What are the best generalizable terms that can predict appearance of SATD?
- RQ4: How well does the best model perform in cross-project testing?

In order to answer these research questions, we focus on five repositories, which are Camel, Log4J, Hadoop, Gerrit, and Tomcat. These repositories have pre-labeled SATD datasets, that were introduced in Maldonado et al. (2017). They were selected by Maldonado et al. (2017), as they cover different application domains, and they are all large and well-commented open-source projects. Instead of relying on manually detected patterns (Potdar and Shihab 2014) or looking for the appearance of manually selected predictor words (Yan et al. 2018), we automatize the predictor selection process by doing an automated feature (word) selection among all the words of the commit messages. Manual labor is both time consuming and error prone, and performing feature selection automatically can save both time and money. For feature selection and regularization, we use logistic regression from Glmnet (Friedman et al. 2010) with lasso regularization. Regularization prevents overfitting by controlling the model complexity with a penalty term λ .

To answer *RQ1*, we use three different NLP techniques (bag-of-words, latent Dirichlet allocation, and word embeddings), to predict SATD from commit messages.

We compare the models built with different techniques against each other in order to determine the best one for SATD prediction. The bag-of-words produced the best result, reaching a median AUC of 0.7411.

RQ2 is answered by replicating the work done by Yan et al. (2018) with logistic regression, and creating another model where their message features are substituted with the automatically selected features from our best performing NLP technique. The two models are compared against each other for AUC performance. When comparing only the message features, our model achieved an improvement of 0.15 of AUC score. When substituting the manually selected predictors with automatically selected ones, the AUC-ROC is improved by 0.03 ($p < 0.0001$).

RQ3 is used to evaluate how generalizable our model is to new projects. We examine the predictor terms chosen by logistic regression, and see what terms were deemed most important predictors across all the 100 models. The most generalizable model has 368 predictors, from which 201 have statistical significance ($p < 0.05$, Wilcoxon rank-sum test) for predicting SATD appearance.

For *RQ4*, we test the best model’s performance in a cross-project environment, where we utilize different projects for training and testing. This validates how well our model can predict SATD from completely new project. This can also help detect defects in projects, which lack a long development history (He et al. 2012).

The main contributions of this paper are determining the most accurate NLP technique for SATD detection from commit messages. This is bag-of-words (AUC 0.7411), which outperforms both topic modeling and word embeddings. We also show how automatic feature selection from commit message contents improves performance in SATD prediction performance over manually selected predictor words of past work. Looking at only message features, the improvement is 0.15 in AUC ($p < 0.001$). When using the full model from Yan

et al. (2018) and changing the manual message features with automatically selected ones, the improvement is 0.03 ($p < 0.001$). We also show how to generate generalizable results when using commit messages from several repositories. This is done by demanding that the word appears in several repositories, and that it appears a certain number of times. Lastly, we produce a list of the predictor words which can predict the appearance of SATD from commit message contents.

The remaining paper is structured as follows. Section 2 first describes the methodology used to extracting, labeling, and processing commit messages, followed by description of the machine learning classifier, and ending with the different methodologies relating to research questions. In Section 3, the results are presented, starting from performance analysis of different models and ending with analysis of the predictor terms related to the appearance of SATD. Finally, in Section 4, we discuss the results in relation to previous work, and also look into threats of validity. We end with Section 5, which offers conclusions and introduces ideas for future work.

2 Methodology

In this section, we explain the methodology used for this paper, which is presented in Fig. 1. The first step is to create the dataset. In this paper, we mine five software repositories, and match them with pre-labeled datasets from a previous work of Maldonado et al. (2017). The second step is to perform basic preprocessing steps for the commit messages, including translating the messages to lower case, removal of stop words, and so on. These basic preprocessing steps increase the quality of the data (Sun et al. 2014). The third step is to apply advanced preprocessing steps to address the problem of very specific project words and spelling errors. This both reduces further noise, and ensures we are left with words that can be used outside specific projects. The fourth and final step is to predict SATD from commit messages using three different NLP techniques (bag-of-words, topic modeling, and word embeddings) utilizing logistic regression as our machine learning classifier. The prediction results are then compared against a previous work of Yan et al. (2018) to examine how the results obtained with automated features selection differ from manually selected message features.

2.1 Step 1: Dataset creation

The dataset for this study was created by combining repository mining with existing datasets. We mined the data needed to answer the research questions from five open-source repositories on Github: Camel,² Log4J,³ Hadoop,⁴ Tomcat,⁵ and Gerrit.⁶ The information was extracted from the projects in commit level, and it included the commit hashes and commit messages.

These repositories were chosen, as we need to have pre-labeled data to determine which commits have included SATD into the development. In order to label the mined commits

²<https://github.com/apache/camel>

³<https://github.com/apache/log4j>

⁴<https://github.com/apache/hadoop>

⁵<https://github.com/apache/tomcat>

⁶<https://github.com/GerritCodeReview/gerrit>

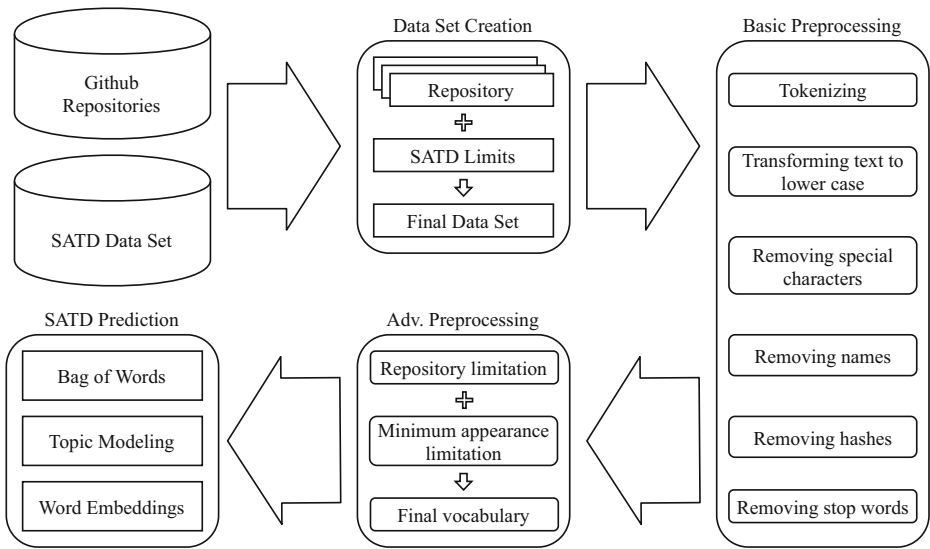


Fig. 1 Overview of methodology

data according to if they included SATD or not, the dataset created by Maldonado et al. (2017) was used. This dataset contains information such as commit hash, the author of the commit, and the code-level comment messages that are identified as SATD inducing changes. However, as this paper is interested only in commit messages and not code-level analyses, only the commit hashes and date information were kept from this dataset.

The hash information from the dataset by Maldonado et al. (2017) was matched against all the commits retrieved from the projects in order to find the SATD introducing commits. The first and last dates from Maldonado et al. (2017) were used as boundaries when matching commits. This is necessary, as the projects are still in active development, and we cannot know if commits that are submitted after the last one listed by Maldonado et al. (2017) would contain SATD.

The total amount of commits was 73,625, from which 1876 were labeled as ones associated with SATD. The SATD percentage between different repositories is between 0.82–3.61%, and for the total dataset 2.55%. The repository information is summarized in Table 1. Overall, the percentage is in line with previous research (da Silva Maldonado et al. 2017; Maldonado E.D.S. and Shihab 2015).

Table 1 Repository commit information

Project name	Total commits	Commits with SATD	Commits without SATD	SATD percentage
Camel	23,190	836	22,354	3.61
Log4J	2815	74	2741	2.63
Hadoop	13,080	333	12,747	2.55
Gerrit	17,708	145	17,563	0.82
Tomcat	16,832	488	16,344	2.90
Total	73,625	1876	71,749	2.55

While SATD percentage from all comments is not high, it can have negative effects on software quality. Wehaibi et al. (2016) show that SATD introduction can lead to increase in software quality, while Kamei et al. (2016) show that SATD gathers interest at least when measured with LOD and Fan-In metrics. When introduced to a project, SATD can survive for a very long time (Bavota and Russo 2016). In that study was also shown that SATD tends to increase over projects lifetime. Once introduced, it takes substantial effort to rework SATD out of the project (Mensah et al. 2016).

2.2 Step 2: Basic preprocessing of the commit messages

After data extraction, we perform basic preprocessing steps for all commit messages, as similar preprocessing is shown to increase the quality of the data (Sun et al. 2014). For all model configurations, the following steps are taken:

- Translating texts to lower case
- Removal of hashes and their references
- Removal of numbers, non alpha-numeric tokens, and single-letter words
- Removal of stop words
- Removal of developer names
- Tokenizing

First, all the commit messages are translated into lower case, as this increases the uniformity of different writing styles. We then remove commit hashes, and similarly to previous work (Movshovitz-Attias and Cohen 2013), we remove numbers, non alpha-numeric tokens, and single-letter words. This cleans the dataset from noise and reduces its size.

Next, removal of stop words is done to further decrease noise in the text. These include words such as “to,” “the,” “she,” and “from.” They do not usually contribute new information for the sentences in which they appear (Kanakaraj and Guddeti 2015). To create a comprehensive stop word list, we combined the base stop word lists from CoreNLP (Manning et al. 2014), Natural Language Toolkit (NLTK) (Loper and Bird 2002), and SnowballC (Bouchet-Valat M. and Bouchet-Valat M.M. 2015) tools.

After these removals, we delete from the messages all human names. These names are obtained by combining several methods. All the developer names that appear as authors in the commits are automatically retrieved. We add to this initial list all the female, male, and common names from Moby Word Lists.⁷ Finally, we add names and nicknames that we found by going over manually the preliminary vocabulary built from the whole dataset.

The last step before building a vocabulary is to tokenize all commit messages. Tokenization refers to separating individual word from the sentences. These individual words are considered features by our machine learning classifier, and they form the basic unit for the analyses.

After this basic preprocessing, the number of terms and word occurrences drop to 25% compared with the size of the original tokenized dataset as seen in Table 2. We consider our basic preprocessing steps and word removals conservative.

2.3 Step 3: Advanced preprocessing to address word generalizability

We initially created our vocabulary by joining all words from different repositories. However, after initial testing, it became clear that this does not lead to the best results with the

⁷<http://www.gutenberg.org/ebooks/3201>

Table 2 Number of different terms and their frequency counts in the final dataset with different models

Model name	No. of different terms	Terms % from original	Total terms	Dataset size % from original
All included, tokenized	108,748	100	3,105,768	100
Basic preprocessing	27,449	25.24	768,563	25.75
Basic preprocessing + 3	4320	3.97	633,464	20.40
≥ 0.001% + 3	3956	3.64	630,140	20.29
≥ 0.005% + 3	1745	1.60	551,769	17.77
≥ 0.01% + 3	1116	1.03	487,542	15.70

generalization of the results. The best predictor words were project specific, relating to individual technologies or code element names. These are shown in Table 3 on column “Basic preprocessed.” Obviously, mentions of specific technologies code element names for one project are worthless as predictors outside of that project. We overcame this deficiency by adding two additional restraints. Firstly, a word has to appear in several repositories, and secondly, it has to appear at least a certain number of times in each of them.

The first restraint ensures that project-specific words are not chosen as predictors. We tested with all the possible combinations, starting from that word has to be in only 1 project’s vocabulary, and ending to one where it has to appear in all of them. As we increase the threshold, we can see that more and more generic terms start to appear. We found that using three repositories as the minimum avoids overfitting of individual repositories and offers a relative balance between number of terms and performance.

The second constraint excludes terms appearing only few times. Rare words do not make good predictors, as they refer to very specific special cases. Defining a minimum appearance limit for terms also eliminates spelling errors (Oliveira et al. 2014). Instead of choosing a single specific number like that which has been done in previous research (Newman et al. 2011; Oliveira et al. 2014), we count the total number of words for each individual repository and use percentile limits to weed out rare words.

Table 3 10 best predictor terms after basic and advanced preprocessing steps with different percentage thresholds

Basic preprocessing	Basic preprocessing + 3	0.001% + 3	0.005% + 3	0.01% + 3
fsccommand	plumb	rough	todo	todo
invokations	rough	bsd	cleanups	happens
colouring	redesign	simulating	checkin	initial
conflicting	bsd	conform	javms	workaround
usign	simulating	powerful	aggregated	progress
startings	conform	todo	pluggable	along
resloveruti	communicating	xp	happens	work
aggregat	powerful	wonder	decoding	think
camelsourceapdater	sorter	tells	outputs	lifecycle
todo	reversion	transition	initial	marked

The limit when a word would be accepted from a repository was calculated by formula:

$$Inclusion = \sum_{i=1}^n x_i \times PT$$

where n is the number of individual words in a repository; x_i is the number of occurrences for an individual word; PT is the percentage threshold. For example, Camel repository contains after basic preprocessing a total of 8858 words which occur a total of 148,146 times. With a PT of 0.01 %, it would mean that a term would have to appear in Camel a total of $148,146 \times 0.0001 = 14$ times to be included from that repository. Fractions are rounded down in all cases. These thresholds are applied individually to all repositories, and using a percentage from the total number of words in them takes into account their different sizes.

Since it was unclear what is the correct percentage threshold, we experimented with several different values to find the one that produces the most generalized results with the largest number of predictor terms. We tested with three percentage threshold limits for inclusion: 0.001%, 0.005%, and 0.01%. Increasing the threshold caused technical terms slowly to disappear from the top predictor words. This can be seen in Table 3, which lists the top 10 words after basic preprocessing and with models made with different percentage thresholds. The models are named after the percentage threshold used in the models, except for the column listed “Basic preprocessing + 3” which applies only the 3 repository constraint. The suffix “+ 3” is used throughout this paper to describe models which have gone through an advanced preprocessing step, and which use the 3 project vocabulary minimum constraint, while the percentage before the + sign is used to mark the PT in that model.

Our heuristic of choosing a cutoff point was to choose the percentage threshold, where we do not anymore see technical or project specific terms on the top 10 predictors. Thus, we chose 0.01% threshold as our limit, as it is the earliest point where terms relating to specific technologies disappear from the 10 best predictor words as shown in Table 3.

The final generic model that we use throughout the rest of our work has therefore the following restraints imposed on it:

- 0.01% percentage threshold
- 3 vocabulary minimum

This model is afterwards referred to as “0.01% + 3” through the rest of the article.

Comparing predictor terms after basic and advanced preprocessing steps shows clear improvement when looking at the generalizability in terms. Looking at Table 3 shows how the best predictor terms change between basic preprocessing and the final 0.01% + 3 model.

After the advanced preprocessing step, we had eliminated project-specific words, spelling errors, and rare terms, reducing our dataset even further in size. Looking at Table 2, we see that all “+ 3” models have only a fraction of different terms remaining compared to the original and basic preprocessed datasets. The total number of terms does not decrease as sharply, which means that we can reduce the number of terms considerably while still retaining a high total in the number of word occurrences.

We also looked how the message length changes between different models, and notice that the median length remains quite constant after applying the 3 repository restraints. There are no big differences in the median message lengths between SATD inducing and non-SATD commit messages. Looking at the average length of commit messages, the SATD commit messages are longer. This information is summarized in Table 4.

Table 4 Message length information for commit messages with different models

Model name	Non-SATD commit messages		SATD commit messages	
	Median	Average	Median	Average
All included, tokenized	7.00	10.08	8.00	12.63
Basic preprocessing	7.00	10.37	8.00	12.96
Basic preprocessing + 3	5.00	8.641	6.00	10.55
$\geq 0.001\% + 3$	5.00	8.603	6.00	10.47
$\geq 0.005\% + 3$	5.00	7.635	5.00	9.135
$\geq 0.01\% + 3$	4.00	6.843	5.00	8.183

2.4 Fourth step: Different NLP techniques, machine learning classifier, and statistical methods

The final step in our method consists of comparing three different NLP techniques on their SATD prediction accuracy from commit messages using logistic regression. The following sections offer an overview of each NLP technique, the machine learning classifier used in this study, and the statistical methods used to compare our results.

2.4.1 NLP technique 1: Bag-of-words

We choose the bag-of-words (BoW) method as our first modeling technique. It is based on distributional semantics introduced already in the 1950s by Harris (1954). Bag-of-words presents every document as a collection of words and the number of times they appear; e.g., if a document d is made of one sentence: “Do you know good software to measure software quality?”, we can express d in a BoW format in the following way:

$$d = [to : 1, do : 1, good : 1, measure : 1, know : 1, quality : 1, software : 2, you : 1]$$

BoW does not care about word order. The method is a relatively straightforward one, and that is one of the reasons it retains its popularity today (Cambria and White 2014).

We use BoW as our baseline NLP method. We consider commit messages as our documents, and mark the frequencies of every word in every document. This way we create a matrix, where every row is an individual commit message and every column an individual word. We use *text2vec*-package for transformations, which utilizes sparse matrices to save memory.

2.4.2 NLP technique 2: Topic discovery

Latent Dirichlet allocation (LDA) is a popular topic modeling technique in software engineering research, which aims at finding topics from texts (Blei et al. 2003). It has been used to study various areas such as developer discussions in Stack Overflow (Barua et al. 2014), code evolution through topic modeling (Thomas et al. 2010), and prioritizing test cases (Thomas et al. 2014; Hemmati et al. 2017). Utilizing topic models instead of BoW for SATD prediction could lead to better results, and it can also shed light on the topics the developers were talking about when they were submitting commits containing SATD.

LDA is based on the idea that documents can be expressed as a collection of different topics. We can classify a document d as $d = \{t_1, t_2, \dots, t_n\}$, where t is a topic probability and n is the number of topics in that document. Similarly every topic t is defined as $t = \{w_1, w_2, \dots, w_n\}$, where n is the number of words in that topic. Each document is therefore a collection of topics with probabilities, and every topic is comprised of collection of words with probabilities.

LDA has three main components, which are number of topics, how many topics a document is expected to have (hyper parameter α), and how many words are expected to be in a single topic (hyper parameter β). All of these have to be decided in advance, and choosing wrong values can have detrimental effect on the results.

Determining the number of topics is a critical parameter when performing LDA (Fu et al. 2015). Estimating the correct number can be difficult, and previous works (Song et al. 2009; Mantyla M.V. et al. 2018; Newman et al. 2011) have set a fixed topic number that has been decided by different criteria. However, this might not provide the optimal number of topics, which can lead to either too broad topics with few clusters, or too complex topics with too many clusters (Zhao et al. 2015).

One way of solving the problem of topic numbers is to use different coherence measures, which evaluates how understandable generated topics are by assigning them a numerical value. There exist several different ways of measuring the coherence (Röder et al. 2015). We follow this approach based on coherence, and use normalized pointwise mutual information (NPMI) as our measurement, as it has been shown to correlate well with human evaluators (Bouma 2009). We apply evolutionary algorithm from *DEoptim* package by Mullen et al. (2009) to discover optimal number of topics, and to find optimal values to the hyper parameters α and β . This is evaluated by choosing the values that produce the highest coherence value calculated with NPMI.

DEoptim is designed to be used for optimizing a specific function. In our case, the function is the LDA model. We use normalized pointwise mutual information (NPMI) technique that is implemented in the *text2vec* package. The following parameters are used in the *DEoptim*, following the guidelines given in the package itself:

- Population = 30
- CR = 0.9
- $F = 0.8$
- Iterations = 200

DEoptim package also requires as an input the boundaries it has to take into account while performing optimization. Defining these boundaries can be difficult, and recent research by Treude and Wagner (2019) suggests that very wide scales should be used for the values. However, using wide scales led to a very unusual result, where all of the topics were practically identical. This causes them to be worthless as predictors. After this discovery, we reverted back to more traditional boundaries:

- Number of topics = 10–100
- $\alpha = 0$ –1.0
- $\beta = 0$ –0.1

We ran the optimization algorithm 200 times to ensure that the results are not achieved because of luck. The LDA model inside the *DEoptim* was run from *text2vec* package with 1000 iterations per run using the vocabulary from 0.01% + 3 model. The optimized values to be used in LDA model were as follows:

- Number of topics = 11
- $\alpha = 0.211534$
- $\beta = 0.079917$

We used these optimized parameters in our LDA model to predict the appearance of SATD.

2.4.3 NLP technique 3: Word embeddings

Word embeddings have likewise been used in several studies related to text analyses in software engineering. They have dealt with myriad of topics such as finding links between questions and API documentation (Ye et al. 2016), named entity recognition (Seok et al. 2016) and finding similar bug reports (Yang et al. 2016).

In this technique, word representations are shown as semantic vectors with multiple dimensions (Pennington et al. 2014). When these vector space models are expressed as distributed representations, they are called *word embeddings* (Turian et al. 2010). Each of the dimensions in these embeddings corresponds to some latent attribute that can hold semantic and syntactic information (Turian et al. 2010; Liu et al. 2015). One benefit of word embeddings is the possibility to compare words to find similarities between them. As these embeddings are compact, they enable fast comparisons (Levy and Goldberg 2014). One of the drawbacks between models created as vector space models, is that a model trained in one domain can include dimensions that are unnecessary in another domain leading to non-optimal results (Efstathiou et al. 2018). We looked into predicting SATD utilizing word embeddings developed for software engineering domain.

We tested word embeddings suitable for software engineering domain, which were created from Sateck Overflow posts in a previous research by Efstathiou et al. (2018). The original model has 1,787,145 words and their vectors. The vectors have been limited to 200 dimensions. We transform the original model in several steps. First, we match all of these word embeddings against our vocabulary, and then create a commit message-vector matrix ($m \times v$), where every row corresponds to a commit message m and every column to one of the 200 word vector dimensions v . Using this cross-referencing cuts down the size of the original matrix considerably.

2.4.4 Machine learning classifier

For our machine learning classifier, we implement a binomial logistic regression with lasso penalty from *glmnet* package. Logistic regression can be used to predict categorical outcomes (classification) and has been shown to perform well in text categorization (Genkin et al. 2007). Logistic regression has also performed well in predicting software faults when compared against random forest or SVM classifiers (Hall et al. 2012).

Besides these, the main key benefits for using logistic regression with lasso penalty are its interpretability and automated feature selection. Logistic regression models are so-called “white-box” models, which allow us to interpret the results better than with “black-box” models. Logistic regression with lasso penalty performs word selection automatically. This allows us to find the best predictor words for SATD, which can include words that might have been missed if they were chosen manually.

Lasso penalty seeks to avoid overfitting by adding a penalty term λ to the logistic regression analysis. Penalty term λ is defined as the sum of the magnitudes from all of the coefficients from the model. Choice of λ can have a big impact on the results, but *glmnet* does this automatically by selecting the one which gives the least amount of prediction error.

The penalty term causes some the coefficients of some features to become 0, eliminating them completely from the equation. This in turn produces smaller models with less features.

We train the classifier with a combined dataset which includes all the individual repositories. To avoid bias in the training and testing, we create 100 models for each configuration by using specific random seeds and stratified 10-fold cross-validation. This approach is known as *J-K*-fold cross validation, in our case 100-10-fold, which reduces performance variance and is particularly recommended for developing models to be used in NLP (Moss et al. 2018). The 100 repetitions also protects us from particularly lucky or unlucky folding of the data in the 10-fold cross-validation. The use of specific random seeds ensures the possibility to replicate the study in the future.

We chose AUC-ROC as our machine learning performance measure, as it has been argued in previous studies to be more robust than F1, precision, and accuracy when performing analyses on class distribution (Lessmann et al. 2008) or comparisons of different models (Yan et al. 2018). AUC-ROC is also shown to work with unbalanced datasets (Prati et al. 2011).

2.4.5 Statistical analysis

To validate our results against previous work by Yan et al. (2018), we compare AUC curves using DeLong's test *pROC*⁸ package created by Robin et al. (2011). It follows the process described by DeLong et al. (1988), but uses the algorithm from Sun and Xu (2014) to considerably speed up the processing time. In all cases, we run DeLong's test 10 times with randomly selected seed numbers, and report the median *p* value from the results. This ensures that the results are not due to luck. DeLong's test is shown to be conservative in some cases as demonstrated by Demler et al. (2012), so we validate our results utilizing also bootstrapping, while using the same alternate hypothesis.

We evaluate statistical significance ($p < 0.05$) of the automatically selected individual predictors by calculating Wilcoxon rank-sum test (Mann and Whitney 1947) for the predictor word appearance and SATD labels. Wilcoxon rank-sum has been utilized earlier with machine learning to determine statistical significance of features selected for machine learning classifiers (Wu et al. 2020; Fan et al. 2018; Yan et al. 2018).

3 Results

In this section, we present the results to our research questions. We start by examining which NLP technique is best suited for NLP detection. Next, we investigate if utilizing this model with automatically selected features can help improve results from previous literature. Thirdly, we look at what predictors were deemed as the most important ones for SATD prediction. Finally, we look at how our model handles a cross-project environment.

3.1 RQ1: What NLP technique predicts most accurately SATD from commit messages?

We compared three different NLP techniques (BoW, LDA, and word embeddings) to find the technique, which most accurately predicts SATD appearance from commit messages.

⁸<https://cran.r-project.org/web/packages/pROC/pROC.pdf>

Here we present all the techniques and their results, and finally we compare them against each other. For each technique, we used the 0.01% + 3 model with 1116 words vocabulary, in order to make the results comparable with each other.

3.1.1 Bag-of-words

Bag-of-words is a popular method used still in NLP today due to its straightforward nature. It sets the baseline to which we can compare our more advanced methods: topic discovery with LDA and word embeddings.

The technique reached median AUC-ROC of 0.7411, while the median number of predictors chosen for a single model is 368. Lasso penalty manages to eliminate a large number of the words from the vocabulary, leaving less than 40% of them as predictors. Small number of predictors means a smaller model which is preferred in machine learning.

As an extra step we wanted to validate our results against different vocabulary sizes. The tested configurations were:

- Basic preprocessing + 3
- 0.001% + 3
- 0.005% + 3

We tested against these configurations to see how our generalized model performs against models which include more technological terms in their vocabularies. An example of the best predictor terms present in these vocabularies was presented earlier in Table 3.

The results for different configurations are shown in Table 5 along with our most generalized 0.01% + 3 model. Vocabulary size is the number of unique words in the final vocabulary. Predictor words column shows the median result on how many words were chosen for a single model. Finally, the last column lists the median result in terms of AUC-ROC for that configuration.

We observe that the most generic model 0.01% + 3 achieves almost the same results as the models built with more extensive vocabularies. The difference between its median AUC to the best performing configuration is only 0.0127 (0.7538–0.7411). At the same time, the 0.01% + 3 model uses only around half as much predictor words. We conclude that using 0.01% + 3 is preferable, as it has several advantages. The trained model has less predictor words resulting in a smaller model. Many technological terms are dropped from the vocabulary offering more generalizable results. And finally, the drop in performance is not big enough to warrant the use of larger models.

3.1.2 Topic discovery

As an alternative to the BoW model, we investigated topics discovery as a way of predicting SATD appearance. We selected LDA as our technique, and optimized its parameters as

Table 5 SATD prediction results for different BoW models

Model name	Vocabulary size	Predictor words	Median AUC
Basic Prep. + 3	4320	652	0.7538
0.001% + 3	3956	611	0.7524
0.005% + 3	1745	433	0.7453
0.01% + 3	1116	368	0.7411

described in Section 2.4.2. For SATD predictions, we build 100 different LDA models, using 1000 iterations in each of them. The models are fitted into commit message topic matrices ($m \times t$), where every row corresponds to a commit message m and every column to a topic t . These matrices are then fed into the logistic regression classifier. The end result is an abstract representation of the best topics that correlate with SATD.

Utilizing the values suggested by the optimization algorithm, and the vocabulary from 0,01% + 3 model, we were able to achieve AUC-ROC values between 0.5587 and 0.6652. The median result was 0.6294.

Similarly like individual predictor terms obtained through BoW method can show what words and therefore subjects, the topics obtained with LDA can offer an insight into larger themes that can cause SATD. Even when the prediction value of LDA clusters does not reach the level obtained from the bag-of-words method, the clusters themselves can be analyzed to see if any larger topics emerge.

In order to find out what kind of topics can describe SATD appearance, we performed K-medoid clustering using R-package *cluster*,⁹ and counted topic stability using guidelines presented in Mantyla M.V. et al. (2018) using the RBO implementation from R-package *gespeR*.¹⁰

Topic stability varied between 0.838 and 0.364 between the 11 clusters. Two clusters rated best to predict SATD had RBO of 0.838 and 0.507. Inside these two topic clusters, we examined the top words present in them to discover emerging themes. The most stable cluster has words such as unit, tests, bug, fixes, patches, and errors. It can mean that actions relating to these activities are more closely related to SATD; e.g., a developer makes a patch or fixes a bug, but the resulting code is not yet polished. The second most stable cluster has words like file, change, user, ui, and bug. This might indicate file changes made to code relating to ui because of some user needs.

These topic analyses would warrant a more in-depth look in the future, even when their predictive power for SATD was worse than the bag-of-words models. However, they could offer an extra insight when used in qualitative analyses between SATD inducing messages and other messages.

3.1.3 Word embeddings

Word embeddings utilizes multidimensional vectors to capture word meanings. We experiment using vectors built from a similar domain to see how well they perform in SATD predictions. This method is also an alternative to BoW and topic modeling.

We used only the word embeddings, which matched the vocabulary from the 0.01% + 3 model. The median AUC-ROC result for predicting with the word vectors was 0.6631. The result is slightly better than with topic modeling, but still not as good as it is with BoW.

We also tested if the number of words might be a limiting factor to the performance. Therefore, we also tested by matching the word embeddings against the vocabulary from 100% + 3 model with its 4320 words. The median AUC result this time was 0.643, so increasing the vocabulary actually led to poorer performance.

Again, even when using word embeddings for SATD prediction does not guarantee better results than BoW, the technique itself can be useful for qualitative analyzing of the contents and should be investigated further.

⁹<https://cran.r-project.org/web/packages/cluster/>

¹⁰<https://rdr.io/bioc/gespeR/>

3.1.4 Comparison of different NLP techniques

Looking at the results from these different techniques, we can see that bag-of-words outperforms the other two techniques, reaching median AUC of 0.7411. The second best result is obtained from word embeddings (median AUC 0.6631), and the LDA shows weakest performance (median AUC 0.6294). The results are summarized in Table 6, which lists each of the techniques along with their median AUC results.

To verify if the best achieving classifier (BoW) was statistically better than the others, we tested the BoW results against the ones from LDA and word embeddings. We utilized *pROC* package, and compared 10 randomly chosen classifier models with the same seed numbers against each other. We ran the test with the alternative hypothesis that word embeddings outperforms the other compared technique. In all cases, the p value from BoW against LDA and word embeddings was much lower ($p < 0.0001$) than the normally considered statistical limit ($p < 0.05$). Therefore, we will accept the alternative hypothesis that BoW outperforms word embeddings and LDA.

Result The best performing NLP technique for predicting SATD from commit messages is bag-of-words, which outperformed both topic modeling and word embeddings. The use of generalized 0.01% + 3 model is advised, as it offers a smaller and more generalizable model with only a minimal loss in performance.

3.2 RQ2: Can we improve with our models over previous work on message analysis?

We compare our model to the baseline approach established by Yan et al. (2018). This is done in two parts, where the first one examines the differences in performance when only the commit message is utilized for prediction. This is done by replicating their message analyzing part and comparing it to the performance of our 0.01% + 3 model. Secondly, we replicate their full model using logistic regression. We then investigate if substituting their manually selected commit message features with our automated predictors leads to improved performance.

3.2.1 Comparison of manually and automatically selected message features

In order to validate our results against our ground truth suggested in Yan et al. (2018), we build a similar dataframe as seen in the original study. This is done by checking if the message contains one of the following words or their conjugations: bug, feature, improve, document, refactor. Then, every commit message is labeled as either containing one of these words or not, and we also measure the length of individual messages by counting how many words they contain. Message length was found to be an important predictor (Yan

Table 6 Comparison of different NLP techniques

Model name	Median AUC
Bag-of-words	0.7411
Word embeddings	0.6631
LDA	0.6294

et al. 2018), so in order to validate its effect to our model, we include it into our 0.01% + 3 model's matrix as an extra column.

The median AUC-ROC in our replication of Yan et al. (2018) was 0.5891. Our 0.01% + 3 model with message length added achieved median result of 0.7411. Adding the message length column did not improve the result of our classifier. The median AUC was precisely the same with or without the message length column. Our model achieves an improvement of + 0.152 to the baseline. We tested the result for statistical significance, using DeLong's test. We ran the test with the alternative hypothesis that the model with the automatically selected features outperforms the original model. In all runs, the p value was significant ($p < 0.0001$). Therefore, we will accept the alternative hypothesis that automatically selected features will improve the performance over manually selected ones.

The results show that our classifier achieves better AUC-ROC than reported in the baseline paper, when considering only the commit message analysis. Instead of manually choosing predictor words, the words should be selected automatically. Counting message length does not improve the results, when the predictor terms are automatically selected.

3.2.2 Comparison of the original model and one with automatically selected features

To see how our automatically selected features would impact the overall result of the model from Yan et al. (2018), we ran a replication study with their full model using logistic regression, and then a comparison with our automatically selected features.

The original model by Yan et al. (2018) had 25 features, which were grouped into three dimensions: Diffusion, History, and Message. We describe here shortly the contents of each dimension, and the full list can be found in Table 7. The Diffusion Dimension has 16 features, and they all relate to code or file changes. History Dimension has 3 features, which deal with developers and change history of the files. Finally, Message Dimension has 6 features, from which 5 check the appearance of manually selected word and one reports the commit message length.

We ran the replication with logistic regression, using both the whole original model, and one where we substituted the manually selected message features with our automatically selected message features. The overview of the method on substituting manually selected features with automatically selected ones is shown in Fig. 2. In both cases, we end up with a matrix, where every row is an individual commit message, and every column a feature.

The median AUC-ROC in the replication study for the original model was 0.7685. Replacing the manually selected message features with our automatically selected features, we obtained median AUC of 0.7979, so there was an improvement of 0.03 in the AUC-ROC score. To validate the results statistical significance, we again compared classifier models against each other with DeLong's test. We ran the test with the alternative hypothesis that the model with the automatically selected features outperforms the original model. In all runs, the p value was significant ($p < 0.0001$).

To further validate the results, we tested the classifier models with the same alternative hypothesis using bootstrapping in place of DeLong's test. Again, in all runs, the p value was significant ($p < 0.0001$). Therefore, we will accept the alternative hypothesis that automatically selected features will improve the performance over manually selected ones.

Result Substituting manually selected message features with automatically selected ones increased the model's performance by 0.03 in AUC-ROC score ($p < 0.0001$).

Table 7 Full list of features used in the baseline model (adapted from Yan et al. (2018))

Name	Definition
Diffusion Dimension	
NS	How many subsystems are modified
ND	How many directories are modified
Entropy	Distribution of modified code across the files
LA	How many lines of coded are added
LD	How many lines of coded are deleted
FA	How many files are added
FM	How many files are modified
FD	How many files are deleted
FR	How many files are renamed
FC	How many files are copied
LCC	How many code changes are made (low significance)
MCC	How many code changes are made (medium significance)
HCC	How many code changes are made (high significance)
CCC	How many code changes are made (crucial significance)
<i>language_num</i>	How many programming languages are changed
<i>file_type_num</i>	How many file types are changed
History Dimension	
NDEV	How many developers did the changes
NUC	How many unique changes are made to the files before this
EXP	How experienced are the developers
Message Dimension	
<i>msg_length</i>	How many words are in the message
<i>has_bug</i>	Check if the word “bug” appears in the commit message
<i>has_feature</i>	Check if the word “feature” appears in the commit message
<i>has_improve</i>	Check if the word “improve” appears in the commit message
<i>has_document</i>	Check if the word “document” appears in the commit message
<i>has_refactor</i>	Check if the word “refactor” appears in the commit message

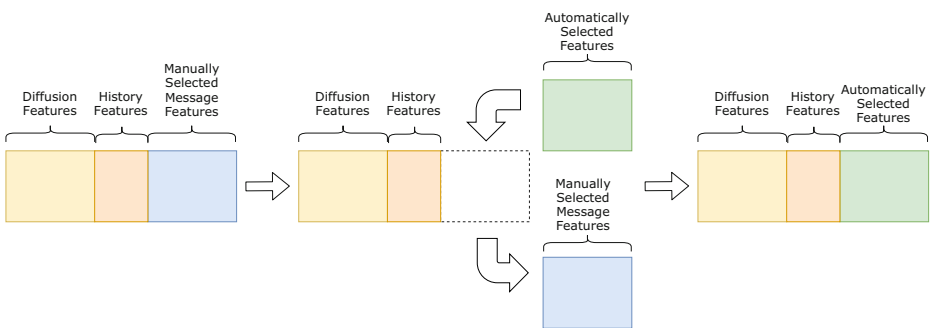


Fig. 2 Extending (Yan et al. 2018) by substituting their manually selected features with our automatic feature selection

3.3 RQ3: What are the best generalizable terms that can predict appearance of SATD?

To look into the automatically selected predictor terms by our classifier, we collect them from all the 100 versions of our 0.01% +3 model, and normalize their values by following formula:

$$\text{median}(x) \times \frac{n_x}{n}$$

where $\text{median}(x)$ is the median coefficient value for the predictor term across all models where it was chosen by logistic regression; n_x is the number of models the predictor term x appeared; n is the total number of models ($n = 100$). Using normalized values guarantees that we do not select terms that happened to get a lucky high number in some of the models. As our machine learning classifier is not deterministic, we need both multiple models and normalization to make sure that we do not select predictors based on luck. This way we instead consider their overall usefulness in terms of predicting SATD.

The most prevalent word was “todo”. Most of the specific technological terms were removed by applying advanced preprocessing step, which imposed specific restrictions on how many repositories and how many times a word has to appear to be included in the vocabulary. There are still technical terms remaining, but in a much smaller quantity.

Table 8 lists the 50 predictor terms with the highest normalized coefficient values for predicting SATD from the 0.01% + 3 model, and Fig. 3 shows top predictors as a word cloud. All of the depicted terms have a positive coefficient for predicting SATD, and their size corresponds to their normalized value. The largest value was for the word todo, followed by happens, initial, and workaround.

To evaluate the predictor terms’ statistical significance, we did the following steps. We checked the appearance of every predictor terms individually in the commit messages and marked if the commit message included this word or not. These appearances are then tested against whether the message is labeled as SATD or not using Wilcoxon rank-sum test. With this method, we found out that from the 368 predictor terms chosen by the logistic regression, 201 terms were deemed as statistically significant ones ($p < 0.05$). The 50 most highly related terms presented in Table 8 were all statistically significant.

Table 8 50 Predictor terms for SATD detection from the most generic model (0.01% + 3)
p-value: *(< 0.05), **(< 0.01), ***(< 0.001)

1. todo***	2. happens***	3. initial***	4. workaround***
5. progress***	6. along***	7. work***	8. think***
9. lifecycle***	10. marked***	11. incomplete*	12. remaining***
13. improved***	14. refactored***	15. copy***	16. temporary***
17. processing***	18. various***	19. applied***	20. first***
21. component***	22. model***	23. future***	24. protocol***
25. interfaces***	26. renaming**	27. merging**	28. integration***
29. yet***	30. introduced***	31. implement***	32. fixes***
33. going***	34. depending***	35. handling***	36. services***
37. implementation***	38. support***	39. working***	40. refactoring***
41. builder***	42. nice***	43. pipeline***	44. though***
45. probably***	46. part***	47. impl*	48. blocking***
49. provide***	50. annotations**		

Table 9 10 Predictor terms and example sentences labeled containing SATD

1.	todo	todo : gsm 0338 pattern <i>needs</i> to be checked
2.	happens	and tried to <i>fix</i> a failing <i>unit test</i> on the bamboo, that happens rarely
3.	initial	initial version of <i>component</i> utilities
4.	workaround	<i>upgrade</i> to sshd 0.10.1 and provided a <i>temporary</i> workaround
5.	progress	deprecate the xxxref options and <i>provide</i> the xxx option if it not exists yet - <i>work</i> in progress
6.	along	added an <i>initial implementation</i> of a debugging interceptor along with integration with the spring <i>based</i> main
7.	work	more work on new camel-netty-http <i>component</i>
8.	think	i think there should be basic multipart <i>handling</i> in the core
9.	lifecycle	it was not correctly invoking the lifecycle <i>methods</i> , nor was it actually evaluating the predicates
10.	marked	marked code for @deprecation

test without this latter limitation, as it can bring out differences on how project-specific terms can impact the prediction accuracy.

The models achieved median AUC-ROC between 0.614 and 0.6998 depending on what repositories were selected for training and testing. The best result was achieved testing against Gerrit using other as training repositories.

Against Log4J and Tomcat repositories, the results got better if the limit that a word has to appear in 3 repositories was removed. With Log4J the increase was noticeable, with the AUC going up from 0.614 to 0.6826. With the other three repositories, the results were always worse with the limitation removed. This increase with Log4J occurs because the best predictor terms happen to be again related to very specific technologies such as “spi” (service provider interface) and “jakarta,” which deals with logging events. Here, “spi” is present only in Camel and Log4J repositories, and “jakarta” is present in Log4J and Tomcat repositories. These two terms are therefore always excluded from 0.01% + 3 model. The results for both models are reported in Table 10.

The overall weaker performance for cross-project testing is expected especially when only a small amount of repositories are available for training. With five repositories, the 3 repository minimum limit can exclude terms, which could be present in the vocabulary if we had a larger pool of repositories to train on.

Result Cross-project testing shows weaker performance compared with the full model. Including project-specific terms can in some cases improve performance.

Table 10 Cross-project median AUC results using 0.01% + 3 model and 0.01% model

Test project	0.01% + 3 model	0.01% model
Camel	0.6459	0.6401
Log4J	0.6140	0.6826
Hadoop	0.6289	0.6080
Gerrit	0.6998	0.6809
Tomcat	0.6578	0.6829

4 Discussion

In this paper, we created and compared NLP classifiers with automated feature (word) selection to detect SATD based on the content of commit messages. All the commit messages were preprocessed in order to exclude word that could carry little informational value. Additionally, we introduced advanced preprocessing step that included new constraints, which defined limits on both how many times a word has to appear and on how many individual repository vocabulary it should appear to be included. Using these makes the results more generic when building a vocabulary from several sources.

Here, we discuss our findings and compare our results to other previous works, where the one we chose as our baseline had selected predictor terms manually for analyzing commit messages (Yan et al. 2018), and others looked into predictors built from source code comments (Huang et al. 2018; Potdar and Shihab 2014). Finally, we look into possible threats to the validity of our work.

4.1 Comparison of models and their performance

We utilized three different NLP methods: BoW, LDA topic modeling, and word embeddings. We wanted to find out which one produced the best results which could still be generalized. Comparing topic modeling with LDA, utilizing ready-made word embeddings and predicting by bag-of-words using plain words, the best results are achieved with bag-of-words.

Bag-of-words model achieved the best prediction results, although it required using the advanced preprocessing step to make the results generalizable. Utilizing two different constraints in the advanced preprocessing step to control both how many times and in how many repositories a word has to appear leads to results that are more easily generalized without great loss in performance. The best model for future purposes is therefore 0.01% + 3 model, which includes 368 predictor words and achieves 0.7411 AUC score.

Word embeddings performed slightly worse than BoW model, even when the word vectors were created from the software engineering domain and contain similar words. The creation of word embeddings from Stack Overflow was done by using a 5-word window for word occurrences (Efstathiou et al. 2018). However, it seems likely that words that are related in Stack Overflow are not necessarily related to each other in developer commit messages. The same words could contain different partners in their 5-word window when looking at Stack Overflow writings or commit messages.

LDA model showed the weakest median performance. Instead of using LDA for making predictions, it can offer other benefits compared with the other methods. Examining the topics generated by LDA can yield information about the generic themes in the commit messages, which in itself can give extra value to research.

4.2 Comparing our model performance with previous work with manually chosen predictor words

The work by Yan et al. (2018) utilized the appearance of 5 manually selected words in commit messages as well as the length of commit messages. They reported an average area under curve ROC of 0.57 when considering only predicting SATD appearance from the commit messages. Our replication study with their method produced a similar result, with median AUC being 0.5891.

Looking at only the message dimension, our 0.01% + 3 model achieved median AUC-ROC of 0.7411, which outperforms both the original result as well as our replication. The improvement in AUC is + 0.15 ($p < 0.001$) compared with the replicated original work.

Substituting their manually selected words with our automatically selected features to complete the original model, we were able to get an improvement of 0.03 in AUC-ROC score, which was deemed as a statistically significant improvement ($p < 0.001$).

Therefore, the performance of using automatically selected words offers better results, and should therefore be preferred over using manually selected predictor words.

4.3 Comparing the predictor terms with manually selected predictors from previous work

In an earlier research (Yan et al. 2018), the predictor terms used to match against commit messages were selected manually. The terms were bug, feature, refactor, improve, and document. These were chosen because in another work (Herzig et al. 2013) these were chosen as the categories for labeling and classifying issue reports.

It is important to note that in Yan et al. (2018), they check the appearance of these five manually selected terms to label the commit message belonging to that category. So if the message contains word “refactor,” the commit message is labeled to belong to this activity. However, this sort of classification has the possibility to miss the messages that do not explicitly use these specific words, even when they might deal with that activity; e.g., refactoring code does not mean that the developer would state explicitly that he or she did refactoring, rather just writing to the message that the code has been improved or even just changed. This sort of weakness has been discussed in earlier literature by Cambria and White (2014). Therefore, relying in the appearance of a single word to categorize the commit messages might not be enough to discover their actual category.

Another possible problem relying on these five predictors is that in the original work of Herzig et al. (2013) the labeling was done to issue tracker tickets and not to commit messages. Therefore, it might be that these categories do not necessarily reflect the commit messages categories.

Comparing these five terms with our automatically selected predictors, we see that one of the words is missing completely. The word “feature” does not appear at all in the best median results of the total 369 predictor terms that the 0.01% + 3 model had chosen.

As one of the terms is missing as predictors, we can conclude that relying on manually chosen predictor terms does not necessarily guarantee the best result. Also, relying on the presence of a single term to classify commit messages to fit a certain activity category might not guarantee to catch all the messages that are about that activity.

4.4 Comparing the predictor terms with source code level predictors from previous work

Finding predictors by analyzing messages at source code level has been looked into at earlier research (Huang et al. 2018; Potdar and Shihab 2014). This practice has also been employed in the industry to find technical debt (Laitila 2019; SonarQube 2019). Looking at the predictors in Huang et al. (2018), we can see both similarities and differences between our work. Here, the authors have identified different features from source code comments, which are all single stemmed terms. They listed the top words after automated feature selection, but acknowledge many of the terms were too generic to offer meanings such as

“should,” “need,” “here,” etc. Therefore, they manually labeled from these terms the ones that appear frequently and are shared between different projects.

Most of our predictors are not mentioned in their lists, although there are some similarities. The shared words are listed in Table 11 with our matching predictors. As can be seen, our automatic word selection did find few similar words, which shows some connection between code level comments and commit messages. The connection is not very strong, as our predictors are mostly different and we are missing words like “yuck,” “ugli,” and “stupid.” It is likely that commits contain more formal language and thus missing words like “stupid” could be expected.

Another study (Potdar and Shihab 2014) listed 62 different hack patterns from source code comments that they examined to be indicators for SATD. As most of these patterns are longer than a single word, we can not directly compare them against our predictors. However, we can see how many of these patterns actually appear in the commit messaged labeled with SATD. Twenty-one out of 62 patterns appear in commit messages, albeit many of them do no match exactly. There are conjugation differences, as well as differences with wording.

The end results is that predictors derived from source level code comments do not necessarily make good predictors when examining commit messages. Developers use different language when they are writing source code comments and when they are writing commit messages. This can be due to different choice of words and conjugations, from which the latter especially lowers the value of using rigid word patterns and expressions.

4.5 Threats to validity and limitations

We will shortly discuss here about threats to validity and limitations relating to our research. Choice of machine learning algorithm can be seen as a threat to validity, as it is possible to use several different machine learning algorithms for NLP purposes. Logistic regression has performed well in text categorization (Genkin et al. 2007), but it has also been shown that at least in the context of detecting code smells the choice of the algorithm does not give a meaningful difference in performance (Fontana et al. 2016).

The choice to repositories used in training and testing can also be seen as limitations. Although the number of commits was over 73 thousand, there might still be some issues with generalization of the results. The projects were all open source projects from different domains, but the results cannot be necessarily generalized to work between other repositories or projects.

For training the model, we require labeled data. We used prelabeled data from previous work by Maldonado et al. (2017). This data was then connected to commit level messages. It is possible that this prelabeling contains some errors, causing us to mislabel some commits.

Table 11 Common terms found in Huang et al. (2018) and our top predictors

Source level words	Our similar words
Todo	Todo
Implement	Implement
Fixm	Fixes / Fix
Workaround	Workaround
Fix	Fixes / Fix

Our choice of using J - K -fold cross validation does not take into account the temporal dimension. Therefore, it might be possible that such commit messages are used to train the model and identify technical debt that are written after the technical debt appeared. However, using J - K -fold cross-validation does protect us from the effects of exceptionally good or bad splits while training the classifiers. Furthermore, we used stratified splitting to keep the ratio of different repositories and messages with and without SATD similar to the whole dataset.

5 Conclusion and future work

We created and tested different NLP techniques to find the best performing one when detecting SATD from commit messages, and compared the results with previous research. Our work has four main contributions:

- Comparing three different NLP techniques to find the best performing one for SATD detection from commit messages
- Improving SATD detection over previous work relying on manually chosen predictor words
- Showing how to generate generalizable prediction models/predictor terms when using commit messages from several repositories
- Producing a list of words that correlate with SATD appearance

Firstly, we examine how LDA and word embeddings perform against our baseline technique relying on bag-of-words. The end result is that even with its relative simplicity, bag-of-words was able to outperform the more advanced LDA and word embedding techniques.

Secondly, we report a higher AUC result than a previous work by Yan et al. (2018) that relied on five manually selected terms and message length. We performed a replication study utilizing their method, and show that utilizing BoW with automatically chosen predictor words outperforms the result of the original model. We also show that message length is not an important attribute when predictor words are selected automatically.

Thirdly, we solve the problem on how to create generalizable results from texts built from multiple sources. The key is to define two thresholds, one for how many times the word has to appear in a repository to be included in its vocabulary, and second one is to set a limit on how many of the repository vocabularies the word has to be in. This ensures that words dealing with project specific things like certain technologies or references to classes and methods are weeded out.

Fourthly, we present here the top 50 words that correlate with SATD appearance in commit messages, and release the full list¹¹ for future research.

In the future, we will expand our research beyond manually labeled commits by detecting SATD inducing commits automatically. First, we will use SATD keywords like “TODO” to bootstrap a machine learning detector. Second, we will iterate upon the bootstrapped detector with active learning which queries for human input only for the borderline comments.

¹¹<https://doi.org/10.6084/m9.figshare.11907216>

We also aim to expand our investigations deeper into the topics that arise when developers commit SATD inducing changes. This will be tied in with expanding the classifier to predict not only SATD but also other forms of technical debt. Therefore, we can analyze further changes in communication patterns between intentional and unintentional admittance of technical debt.

This could be done by new implementations of word embeddings and topic modeling. Even when their performance in SATD detection was poorer than bag-of-words, they could be investigated as qualitative analysis techniques for commit message contents. Other possibilities include exploring the contents with commit specific word embeddings, or utilizing deep learning techniques.

Another research topic for the future is further investigation of code level attributes, and their relationship to other predictors, such as predictor terms from commit messages, code level changes, and so on. Utilizing a combination of such predictors could possibly yield even better results, while at the same time offering an expanded view into what kinds of predictors are connected to specific types of technical debt.

Currently, the industry is utilizing technical debt predictors for source code level comments similarly as in Huang et al. (2018). State of the practice services such as SonarQube employ rules (SonarQube 2019) that are derived from recommendations done by MITRE Corporation (MITRE 2019). These include looking for few specific keywords like “TODO,” “FIXME,” and “HACK” from code level comments. However, there is currently no industry-supported tool that would look at commit messages for technical debt detection. One industry blog (Laitila 2019) mentions that such a feature does not exist yet, which might indicate that commit message analysis is a possible trend for the future.

Another topic for the future is to investigate whether there exist differences between projects, which make large commits less frequently and projects implementing small commits at a rapid pace.

One problem to overcome is to ensure that the developed techniques would be adopted to practice. Even when this is outside the scope of this paper, possible solutions would be to investigate an automated SATD predictor, which utilizes active learning to learn. This could be done e.g. by creating non-intrusive reports about potentially fault-inducing commits, which the developers could then check and either flag the report either false or true. This feedback could be then inserted back into the machine learning tool to update the model. Another option is to build a tool, which makes the reports and checks automatically on code level, if the developer made any changes and updates the model based on this information. These are good directions for future research, and we are currently engaging in talks with an industry partner for developing a solution for an automatic technical debt prediction to their actual monitoring tools.

Acknowledgments The authors are thankful for Meng Yan and other authors of Yan et al. (2018), who provided their dataset for replication.

Funding information Open access funding provided by University of Oulu including Oulu University Hospital. The authors have been supported by Infotech Oulu and Academy of Finland (grants 298020 and 328058).

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- AlOmar, E., Mkaouer, M.W., Ouni, A. (2019). Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd international workshop on refactoring (IWor)* (pp. 51–58). IEEE.
- Barua, A., Thomas, S.W., Hassan, A.E. (2014). What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, *19*(3), 619–654.
- Bavota, G., & Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th international conference on mining software repositories*, pp 315–326.
- Besker, T., Martini, A., Bosch, J. (2018). Technical debt cripples software developer productivity: a longitudinal study on developers' daily software development work. In *Proceedings of the 2018 International Conference on Technical Debt*, pp 105–114.
- Blei, D.M., Ng, A.Y., Jordan, M.I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, *3*(Jan), 993–1022.
- Bouchet-Valat M., & Bouchet-Valat M.M. (2015). Package 'snowballc'.
- Bouma, G. (2009). Normalized (pointwise) mutual information in collocation extraction. In *Proceedings of GSCS*, pp 31–40.
- Cambria, E., & White, B. (2014). Jumping nlp curves: a review of natural language processing research. *IEEE Computational Intelligence Magazine*, *9*(2), 48–57.
- da Silva Maldonado, E., Shihab, E., Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, *43*(11), 1044–1062.
- DeLong, E.R., DeLong, D.M., Clarke-Pearson, D.L. (1988). Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. *Biometrics*, pp 837–845.
- Demler, O.V., Pencina, M.J., D'Agostino, Sr. R.B. (2012). Misuse of delong test to compare aucs for nested models. *Statistics in Medicine*, *31*(23), 2577–2587.
- Efstathiou, V., Chatzilenas, C., Spinellis, D. (2018). Word embeddings for the software engineering domain. In *Proceedings of the 15th international conference on mining software repositories* (pp. 38–41). ACM.
- Fan, Y., Xia, X., Lo, D., Hassan AE (2018). Chaff from the wheat: characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering*.
- Fontana, F.A., Mäntylä, M.V., Zaroni, M., Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191.
- Friedman, J., Hastie, T., Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, *33*(1), 1.
- Fu, Y., Yan, M., Zhang, X., Xu, L., Yang, D., Kymer, J.D. (2015). Automated classification of software change messages by semi-supervised latent dirichlet allocation. *Information and Software Technology*, *57*, 369–377.
- Genkin, A., Lewis, D.D., Madigan, D. (2007). Large-scale bayesian logistic regression for text categorization. *Technometrics*, *49*(3), 291–304.
- Guzman, E., Azócar, D., Li, Y. (2014). Sentiment analysis of commit comments in github: an empirical study. In *Proceedings of the 11th working conference on mining software repositories*, pp 352–355.
- Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, *38*(6), 1276–1304.
- Harris, Z.S. (1954). Distributional structure. *Word*, *10*(2-3), 146–162.
- He, Z., Shu, F., Yang, Y., Li, M., Wang, Q. (2012). An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, *19*(2), 167–199.
- Hemmati, H., Fang, Z., Mäntylä, M.V., Adams, B. (2017). Prioritizing manual test cases in rapid release environments. *Software Testing, Verification and Reliability*, *27*(6), e1609.

- Herzig, K., Just, S., Zeller, A. (2013). It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 international conference on software engineering* (pp. 392–401). Piscataway: IEEE Press.
- Huang, Q., Shihab, E., Xia, X., Lo, D., Li, S. (2018). Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1), 418–451.
- Kamei, Y., Maldonado, E.D.S., Shihab, E., Ubayashi, N. (2016). Using analytics to quantify interest of self-admitted technical debt. In *QuASoQ/TDA@ APSEC*, pp 68–71.
- Kanakaraj, M., & Guddeti, R.MR. (2015). Nlp based sentiment analysis on twitter data using ensemble classifiers. In *2015 3Rd international conference on signal processing, communication and networking(ICSCN)* (pp. 1-5). IEEE.
- Laitila, V. (2019). Technical debt analysis. <http://help.softagram.com/articles/2887068-technical-debt-analysis>.
- Lessmann, S., Baesens, B., Mues, C., Pietsch, S. (2008). Benchmarking classification models for software defect prediction: a proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4), 485–496.
- Levy, O., & Goldberg, Y. (2014). Dependency-based word embeddings. In *Proceedings of the 52nd annual meeting of the association for computational linguistics (volume 2: short papers)*, (Vol. 2 pp. 302-308).
- Liu, Y., Liu, Z., Chua, T.S., Sun, M. (2015). Topical word embeddings. In *Twenty-Ninth AAAI conference on artificial intelligence*.
- Loper, E., & Bird, S. (2002). Nltk: The natural language toolkit. In *Proceedings of the ACL workshop on effective tools and methodologies for teaching natural language processing and computational linguistics*, pp 62–69.
- Maldonado E.D.S., & Shihab, E. (2015). Detecting and quantifying different types of self-admitted technical debt. In *2015 IEEE 7Th international workshop on managing technical debt(MTD)* (pp. 9-15). IEEE.
- Maldonado, E.D.S., Abdalkareem, R., Shihab, E., Serebrenik, A. (2017). An empirical study on the removal of self-admitted technical debt. In *Software maintenance and evolution (ICSME), 2017 IEEE international conference on* (pp. 238-248). IEEE.
- Mann, H.B., & Whitney, D.R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, pp 50–60.
- Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D. (2014). The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pp 55–60.
- Mantyla M.V., Claes M., Farooq U. (2018). Measuring lda topic stability from clusters of replicated runs. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement* (p. 49). ACM.
- Mensah, S., Keung, J., Bosu, M.F., Bennin, K.E. (2016). Rework effort estimation of self-admitted technical debt.
- MITRE (2019). Common weakness enumeration. <http://cwe.mitre.org/data/definitions/546.html>.
- Moss, H.B., Leslie, D.S., Rayson, P. (2018). Using jk fold cross validation to reduce variance when tuning nlp models. In *Proceedings of the 27th international conference on computational linguistics*, pp 2978–2989.
- Movshovitz-Attias, D., & Cohen, W.W. (2013). Natural language models for predicting programming comments. In *Proceedings of the 51st annual meeting of the association for computational linguistics (Volume 2: Short Papers)*, (Vol. 2 pp. 35-40).
- Mullen, K.M., Ardia, D., Gil, D.L., Windover, D., Cline, J. (2009). Deoptim: An r package for global optimization by differential evolution.
- Newman, D., Bonilla, E.V., Buntine, W. (2011). Improving topic coherence with regularized topic models. In *Advances in neural information processing systems*, pp 496–504.
- Oliveira, N., Cortez, P., Areal, N. (2014). Automatic creation of stock market lexicons for sentiment analysis using stocktwits data. In *Proceedings of the 18th international database engineering & applications symposium* (pp. 115–123). ACM.
- Pennington, J., Socher, R., Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp 1532–1543.
- Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on software maintenance and evolution(ICSME)* (pp. 91-100). IEEE.
- Prati, R.C., Batista, G., Monard, M.C. (2011). A survey on graphical methods for classification predictive performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 23(11), 1601–1618.
- Robin, X., Turck, N., Hainard, A., Tiberti, N., Lisacek, F., Sanchez, J.C., Müller, M. (2011). Proc: an open-source package for r and s+ to analyze and compare roc curves. *BMC Bioinformatics*, 12, 77.

- Röder, M., Both, A., Hinneburg, A. (2015). Exploring the space of topic coherence measures. In *Proceedings of the eighth ACM international conference on Web search and data mining* (pp. 399–408). ACM.
- Sas, D., & Avgeriou, P. (2019). Quality attribute trade-offs in the embedded systems industry: an exploratory case study. *Software Quality Journal*, 1–30.
- Seok, M., Song, H.J., Park, C.Y., Kim, J.D., Ys, K.im. (2016). Named entity recognition using word embedding as a feature. *International Journal Software Engineering Application*, 10(2), 93–104.
- Sinha, V., Lazar, A., Sharif, B. (2016). Analyzing developer sentiment in commit logs. In *Proceedings of the 13th international conference on mining software repositories*, pp 520–523.
- SonarQube (2019). Rules explorer. <https://rules.sonarsource.com/>.
- Song, Y., Pan, S., Liu, S., Zhou, M.X., Qian, W. (2009). Topic and keyword re-ranking for lda-based topic modeling. In *Proceedings of the 18th ACM conference on information and knowledge management* (pp. 1757–1760). ACM.
- Sun, X., & Xu, W. (2014). Fast implementation of delong’s algorithm for comparing the areas under correlated receiver operating characteristic curves. *IEEE Signal Processing Letters*, 21(11), 1389–1393.
- Sun, X., Liu, X., Hu, J., Zhu, J. (2014). Empirical studies on the nlp techniques for source code data pre-processing. In *Proceedings of the 2014 3rd international workshop on evidential assessment of software technologies* (pp. 32–39). ACM.
- Thomas, S.W., Adams, B., Hassan, A.E., Blostein, D. (2010). Validating the use of topic models for software evolution. In *2010 10th IEEE working conference on source code analysis and manipulation* (pp. 55–64): IEEE.
- Thomas, S.W., Hemmati, H., Hassan, A.E., Blostein, D. (2014). Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1), 182–212.
- Tom, E., Aurum, A., Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6), 1498–1516.
- Treude, C., & Wagner, M. (2019). Predicting good configurations for github and stack overflow topic models. In *Proceedings of the 16th international conference on mining software repositories* (pp. 84–95). IEEE Press.
- Turian, J., Ratinov, L., Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics, association for computational linguistics*, pp 384–394.
- Wehaibi, S., Shihab, E., Guerrouj, L. (2016). Examining technical debt on software quality. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, (Vol. 1 pp. 179–188). IEEE.
- Wu, X., Zheng, W., Pu, M., Chen, J., Mu, D. (2020). Invalid bug reports complicate the software aging situation. *Software Quality Journal*, 1–26.
- Yan, M., Xia, X., Shihab E., Lo, D., Yin, J., Yang, X. (2018). Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering*.
- Yang, X., Lo, D., Xia, X., Bao, L., Sun, J. (2016). Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th international symposium on software reliability engineering(ISSRE)* (pp. 127–137). IEEE.
- Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C. (2016). From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering* (pp. 404–415). ACM.
- Zhao, W., Chen, J.J., Perkins, R., Liu, Z., Ge, W., Ding, Y., Zou, W. (2015). A heuristic approach to determine an appropriate number of topics in topic modeling. In *BMC bioinformatics, biomed central*, (Vol. 16 p. S8).
- Zhou Y, & Sharma A (2017). Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp 914–919.



Leevi Rantala is a PhD student of Software Engineering at the University of Oulu, Finland. He received his M.Sc. degree in Software Engineering in 2018 from the University of Oulu. He is interested in researching software development with different machine learning and natural language processing methods, with focus being at detecting and analyzing technical debt.



Mika Mäntylä is a professor of Software Engineering at the University of Oulu, Finland. He received a D. Sc. degree in software engineering from the Helsinki University of Technology, Finland. His research interests include empirical software engineering, software testing, software maintenance, mining software repositories, and behavioral software engineering. He has previously worked as a post-doc at the Lund University, Sweden and as an assistant professor at the Aalto University, Finland. His studies have appeared in journals such as IEEE Transaction on Software Engineering, IEEE Software, Empirical Software Engineering, and Information and Software Technology. He currently serves as an associated editor for IEEE Software. For more information <http://mikamantyla.eu/>