Received July 8, 2019, accepted July 22, 2019, date of publication August 7, 2019, date of current version August 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2933318

Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding

JERNEJ FLISAR AND VILI PODGORELEC, (Member, IEEE)
Faculty of Electrical Engineering and Computer Science, University of Maribor, SI-2000 Maribor, Slovenia

Corresponding author: Vili Podgorelec (vili.podgorelec@um.si)

This work was supported by the Slovenian Research Agency (Research Core Funding No. P2-0057).

ABSTRACT Self-admitted technical debt (SATD) is annotated in source code comments by developers and has been recognized as a great source of discovering flawed software. To reduce manual effort, some recent studies have focused on automated detection of SATD using text classification methods. To train their classifier, these methods need labeled samples, which also require a lot of effort to obtain. We developed a new SATD identification method, which takes advantage of a large corpus of unlabeled code comments, extracted from open source projects, to train a word embedding model. After applying feature selection, the pre-trained word embedding is used for discovering semantically similar features in source code comments to enhance the original feature set. By using such enhanced feature set for classification, our goal was to improve the identification of SATD when compared to existing methods. The proposed feature enhancement method was used with the three most common feature selection methods (CHI, IG, and MI), and three well-known text classification algorithms (NB, SVM, and ME) and was tested on ten open source projects. The experimental results show a significant improvement in SATD identification over the compared methods. With an achieved 82% of correct predictions of SATD, the proposed method seems to be a good candidate to be adopted in practice.

INDEX TERMS Feature enhancement, feature selection, self-admitted technical debt, text classification, word embedding.

I. INTRODUCTION

In software development, there has always been a trade-off between software code quality and timely software release [2]. A metaphor describing this phenomenon is known as technical debt [4]. Technical debt has a negative impact on future development, like the increased future cost for code refactoring or architectural redesigning [43], [46].

In order to manage technical debt effectively and efficiently, it first needs to be identified properly. However, detecting technical debt has always been a challenging task [7], [47]. The tools used are especially useful for identifying defect debt but cannot help in identifying many other types of debt, including technical debt, so involving humans in the identification process is still necessary. Manual inspection of the code, however, is very demanding and requires a lot of effort. In a constantly-pressing push for timely delivery of software, an adequate manual inspection is frequently neglected. In this manner, some automatic approaches to the identification of technical debt have arisen. Such methods generally use static code analyzers to detect different types of anomalies in the source code, like design problems, defects, and others [48].

Recently, a new aspect of technical debt emerged - self-admitted technical debt (SATD). SATD refers to deliberate technical debt, that was introduced in source code intentionally by developers with their documentation of the debt, using source code comments [29]. Developers introduce SATD to the source code for different reasons. Mostly it is because of lack of time to develop quality code, last minute changes, quick fixes etc. [1]. A tool for suggesting when developers should have marked source code fragments as SATD was proposed in [44].

Just like technical debt, SATD also tends to have a negative impact on software quality and appears frequently in projects [21], [29], [41]. A survey in a recent study has

The associate editor coordinating the review of this manuscript and approving it for publication was Xi Peng.

shown that 88% of developers participating in that survey introduce SATD in source code with the intent of future refactoring [38].

While the manual classification of comments can provide good results, it is far too inefficient, especially if millions of lines of code from several projects have to be analyzed. There are some approaches described in the literature, which have focused on the identification of SATD from source code comments. Potdar and Shihab inspected comments of open source Java projects manually on GitHub, to detect SATD [29]. They identified 62 text patterns, which were later used for the identification of SATD [20], [41]. Most recently, two approaches were proposed based on natural language processing and text mining techniques for automated detection and classification of SATD comments [12], [19]. Both of these studies used manually annotated source code comments, from different open source java projects, as training data, to build their classification model.

In these studies, comments are considered as short texts, represented as bag-of-words (BOW) model [33]. Comments can be viewed as short text documents since their length is short and text is usually written in an informal language [34]. Each comment is represented as a feature vector. The feature values in a vector for a single instance (i.e. a comment) are counts of word occurrences within the comment. Despite good overall performance, the BOW model has its own drawbacks [9]. The main drawbacks are high dimensionality (a corpus of all possible words, which are used in source code comments, is very large) and extreme sparsity (in a single comment, only a very small amount of words are used) since every unique word is treated as a feature. The number of unique words can be very high; but as many of them are irrelevant for text classifiers, they can decrease the performance of text classification.

To overcome the problem of high dimensionality in text classification tasks, feature selection (FS) methods, such as Information Gain (IG), Chi-square (CHI) and Mutual Information (MI), are usually applied [42], and have already been applied to improve the performance of SATD classification [12]. On the other hand, the BOW presentation model has problems with extreme sparsity in the case of short texts. Additionally, short texts are usually noisier, informal and do not provide enough contextual information [3], [27], which lead to degradation of performance in short text classification tasks. Classifiers, constructed from BOW models, can fail in capturing proper differences of high dimensional and sparse document vectors. To overcome those problems, many recent studies have applied word embedding in order to improve performance of text classification tasks [14], [17], [40]. Word embedding is a distributed representation, where words are presented as low-dimensional and real-valued vectors. In embedded space, semantically similar words tend to have similar vectors [11].

The method proposed in this study builds on the combination of FS for reducing the number of features and thus reducing the dimensionality of the problem and then enhancing the selected features with the most similar not-used features from the constructed word embedding vector space model, and thus lessening the problem of sparsity.

Word embedding can be adapted to different tasks using different corpora. Despite that, to the best of our knowledge, no such model has been created from source code comments. Therefore, in our recent work, we constructed such word embedding model from source code comments, using word2vec implementation, to create a word vector space model for the software engineering domain [8]. By enhancing the IG FS method, using similarity measures from the word embedding model to select the best features, and applying the method to the identification of SATD, we were able to improve the classification of SATD comments.

Encouraged by the promising results, in this paper we present a new, expanded method for SATD detection from source code comments, which is based on enhanced feature selection using word embedding, and discuss the obtained results in detail. The main difference of our SATD identification approach, presented in this paper, in regard to similar existing approaches [12], is that we use a large amount of unlabeled source code comments to train a word embedding model first, from which we next enhance the prepared BOW model in order to obtain a richer, more informative set of words for classification. As we are using classification algorithms for classifying source code comments into two classes (whether they contain SATD or not), we are using the term classification of SATD throughout the paper. The main contributions of this paper are fourfold:

- First, a word embedding model is constructed to detect semantically related terms from source code comments (within a software engineering domain). The model is built from 1.2 million lines of source code comments, extracted from more than 200.000 source files, representing more than 360 open source java projects on GitHub, using the word2vec tool.
- Second, a two-step FS method is proposed, using the three most common FS methods (CHI, IG and MI) in combination with the word embedding model. The method is used to select a set of features, which are then used for a learning algorithm, in order to improve the classification results.
- Third, the results obtained from the conducted empirical experiments are presented and discussed in detail. To evaluate the effectiveness of our proposed FS method objectively, we performed comprehensive statistical tests, which show a significant improvement of our proposed method when compared to existing methods.
- Finally, we applied our method to classify source code comments in several popular open source projects, to detect and identify comments with SATD.

The remainder of this paper is organized as follows. Section 2 describes the background of the word embedding idea and FS methods for text classification. In Section 3, the proposed method for SATD identification is described. Experimental setup and results are described in detail in

TABLE 1. Sample of comments with SATD.

comment
// Yuck! This is not quite good enough, it's a quick hack around the problem
// TODO: is there a more elegant way than downcasting?
// XXX ignore attributes in a different NS (maybe store them ?)
// Ugly hack for cases like elements(foo.bar.collection)
// Quick hack check for a common case
// Below is a funny looking workaround

Sections 4 and 5. In Section 6, we discuss the obtained results and their implications. Conclusions and some plans for future work are provided in Section 7.

II. BACKGROUND

SATD is technical debt that is described in source code comments, and is introduced intentionally by developers. Table 1 provides a sample of comments that were identified as indicating self-admitted technical debt. It was first described by Potdar and Shihab [29]. In their study, they examined four open source projects manually and extracted textual patterns which were likely indicating SATD comments. In a follow-up study [20], different types of SATD were identified, namely, defect debt, design debt, documentation debt, requirement debt and test debt. They found out, that the most common type of SATD comments are design debt (ranging between 42%-84%) and requirement debt (ranging between 5% and 45%).

The impacts of SATD on software quality was examined by Wehaibi et al. [41]. They analyzed five open-source projects, and found out that changes in code when SATD was introduced were more difficult to perform. Removal of SATD was studied in [18], [45]. A study from Maldonado et al. [18] showed that the majority of SATD is removed, and that the average time to remove it was between 82 to 613 days. Since the removal of SATD was based just on the removal of comments reflecting SATD, Zampetti et al. [45] performed an in-depth study of SATD removal with relation to the affected source code. Their findings showed that a large percentage of SATD removal occurs unintentionally, since only 8% of SATD removal was acknowledged in commit messages.

The diffusion rate of SATD comments in open source projects was studied by Bavota and Russo [1]. In their automated classification of comments, they identified, on average, 51 SATD comments per project. Additionally, they found out that survivability of SATD was long, with an average of over 1000 commits per project, before SATD was removed. To automate the process of SATD identification, they used textual patterns like *todo*, *fixme*, and *workaround*, which were proposed in [29]. The accuracy of such heuristics can be quite low, which can affect the results of the studies [1]. In recent works [12], [19], [21], text mining techniques were used to improve identification of SATD comments, where significant improvement of text classification approaches over the pattern-based approach was reported [19].

In literature, all of the studies using text mining methods for automated SATD classification were based on a BOW representational model. Since every unique word from the provided dataset can be used as a feature, we are dealing with a high-dimensional space, which can cause degradation of performance of a given text classifier. To overcome this problem, dimension reduction techniques such as FS are applied in the pre-processing step of text classification. The FS approaches aim to reduce the number of features by selecting the most discriminating features in order to improve the classification performance, independently of the used classification algorithm. With a smaller set of features, the text classification performance can be improved, as well as the computational cost of a classifier can be reduced [33].

A. FEATURE SELECTION

Previous studies have shown that FS methods could improve the performance of text classification [42]. FS methods rank features based on evaluation criteria to identify features that are most useful in differentiating classes. Usually, a threshold is set to select the top k% of features as a feature set, where k is set experimentally.

Since FS methods select a feature subset from the original feature space according to an evaluation criterion using only a labeled dataset, its selection relies only on the provided dataset. This can be problematic, in our case, since source code comments can be very diverse, as they are from different domains of applications. Additionally, comments are written as informal text, using different words and jargon, by various developers from different backgrounds. For example, developers in one project can comment on unimplemented methods with the word "todo" and others with "xxx". Another example can be the words "hack" and "workaround" which are typical words when SATD is introduced [29]. Those words are very similar semantically but are treated as separate features when an FS method calculates their contribution score. If their score is too low, they can both be excluded from the feature subset list, which can lead to degradation of performance. To address this problem, we select additional features based on similarity. Since BOW representations ignore the dependency among words in context and cannot be used to measure words similarity, we used the learned word embedding.

B. WORD EMBEDDING

Word embeddings are based on the hypothesis that words appearing in similar context tend to have similar meaning [11]. Word embedding is a learned representation for text, where words with similar meaning have similar representation as real-valued vectors in a predefined vector space. Unlike BOW representation, distributed representation of words in embedding space can capture the semantics of a word. They are usually learned from neural language models, and have been applied successfully in natural language processing tasks such as information retrieval, sentiment analysis, paraphrase detection and text classification [5], [37]. One of the most established word embedding model is word2vec where the model is trained from large unlabeled text corpora [22].

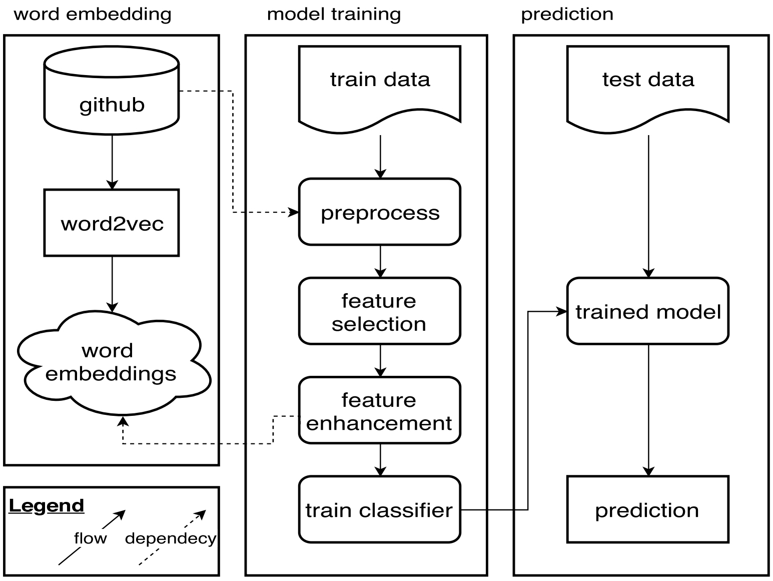


FIGURE 1. An overview of our SATD classification approach.

Word2vec is a neural network-based method for efficient learning of word embeddings from a text corpus [22]. It contains two different approaches. Continuous bag of word (CBOW) learns embedding by predicting current word based on its context. The second approach is the skip-gram model, which predicts the surrounding word in a current context. Both methods learn about words from their surrounding context, where context is defined as the window size of neighboring words and is a configurable parameter in the model. Another configurable parameter in the model is the size of a vector, where each dimension encodes different aspects of a word. This model has already been used on short texts classification tasks with much success. Le and Mikolov combined word embedding using simple vector addition to create paragraph vectors to classify movie reviews [15]. Kim et al. used the word2vec model and combined similar words to clusters to design a bag-of-concept model, where clusters are represented as concepts [14]. Wang et al. expanded short texts with related words, defined in word embedding space, to improve classification performance [40].

III. METHOD

In this section, the proposed method for SATD identification is described, as shown in Fig. 1. Our method aims to enhance a FS process by taking advantage of the pre-trained word embeddings for detecting similar features in source code comments to improve SATD classification. The main difference of our method with regard to classical text classification process, is that we use a large amount of unlabeled text (source code comments) to train a word embedding model first. Then, in the process of SATD classification model training, source code comments are first pre-processed and represented in a form of BOW. After selecting the most informative features for classification, for each of the selected feature its most similar features, not already contained in the prepared BOW model, are taken from the trained word embedding model and added to the BOW model in order to obtain a richer, more informative set of words for SATD classification. The classifier is then trained upon such enhanced BOW representation using a common classification algorithm.

A. OVERALL SYSTEM

Fig. 1 presents an overall framework of our approach. The framework is composed of three separated phases: The word embedding phase, the model training phase, and the prediction phase. In the word embedding phase, a *word2vec* model is built to support the feature enhancement method for the model training phase. In the model training phase, the SATD classifier is built within the next four steps: Comments' preprocessing, feature selection, feature enhancement, and classifier training. In the prediction phase, the built SATD classification model can be used to perform a SATD prediction upon new, previously unseen code comments.

In the model training phase, we first preprocess the text of comments, from the corpus of labeled comments (i.e., a comment with or without SATD) from different open source projects, to extract features (i.e., words), to represent each comment. Then, the FS method is applied to compute the score for each extracted feature. Here, generally, any known FS method can be applied. Features are then ranked by their scores, from the highest to the lowest, to create a ranked list of features. Next, feature enhancement is applied, where the top *k*% of features are selected (from the list of previous step), and, for each of those features, we search for *t* most similar words in the trained word embedding model, generated in the word embedding phase, using the cosine similarity measure between two word vectors, which is computed as:

$$sim(w_t, w_u) = \frac{w_t \cdot w_u}{||w_t|| \cdot ||w_u||} \tag{1}$$

where w_t and w_u are words in the provided data set, where words are embedded into a semantic space model.

This word embedding model is built using the gensim library [30], which implements Mikolov's *word2vec* skip-gram model. The skip-gram model learns the vector representation of words that are useful for predicting surrounding words in a text.

In the last step, we train the text classifier using an enhanced set of features, provided by our enhanced FS method in the feature enhancement step.

B. PREPROCESSING

Source code comments are texts written in a natural language. The initial phase in any classification process is usually text preprocessing which is composed of three steps: tokenization, stop words removal and stemming. We use preprocessing methods, to clean up text comments of some Java language syntax and strip of all punctuations and numeric characters.

1) Tokenization. Tokenization is the process that splits text into words, phrases or other meaningful elements so-called tokens. Comments are stripped of all punctuations, non-alpha and numeric characters and extensive whitespaces, so only English letters remains in a token. Finally, all words all converted to lowercase.
2) Stop-words removal. Some words like preposition don't carry much meaning and are used frequently text documents. Typically, they are filtered out, since they don't carry much meaning in distinguishing different categories of text. We manually build a small list of stop words, which contains prepositions like ''the'', ''to'', ''at'', etc. We didn't use the standard stop-word list since it is not built for software engineering domain and some words in this list can provide some useful information [12].
3) Stemming. Stemming is the process of reducing a word to its base or root form. We apply Porter stemmer [28] algorithm to reduce word to its root form. For example, the words ''develop'', ''developer'' and ''development'' would all be reduced to their base form ''develop''.

The described preprocessing of source code comments is performed completely the same in both the word embedding phase and in the model training phase.

C. FEATURE SELECTION

After the preprocessing step, we use the BOW model to represent each comment with a word vector. Altogether, we have a large vector dimension, since every unique word is a feature. In this manner, we are faced with the problem of high dimensionality [39]. To address this problem, we apply an FS method, which selects the most relevant features f in a given data set S. FS methods can be divided into three classes [36]: Filter models, wrapper models, and embedded models. In this paper, we focus on the filtering approach. This approach aims to reduce the number of features by selecting the most discriminative features, independent of the selected classification algorithm. Many effective filtering FS methods have been applied for text classification. The typical FS methods include information gain (IG), chi-square (CHI), and mutual information (MI) [42]. These FS methods calculate the contribution of each independent feature to identify the category. Based on this contribution, only the features with high scores in identifying different categories are selected. With a small set of features, we can improve the text classification performance, as well as reduce the computational cost of training the classifier [33]. Our work can be utilized with any FS method since it just uses a final feature set provided by the used FS method. In this paper, we consider the three most common FS methods, i.e., IG, CHI, and MI.

1) INFORMATION GAIN

Information gain (IG) measures the amount of information a feature have about the class prediction, where only information available is the presence of the feature [24]. IG is one of the most popular and widely used FS technique in text mining [33]. It measures the number of bits of information required for category prediction by knowing the presence or absence of a word in the text document. The higher the score, the more important is the word for category prediction. The score for word w is computed as [39]:

$$\begin{aligned} IG(w) = &-\sum_{i=1}^{|C|} P(c_i) log P(c_i) \\ &+ P(w) \sum_{i=1}^{|C|} P(c_i, w) log P(c_i, w) \\ &+ P(\bar{w}) \times \sum_{i=1}^{|C|} P(c_i, \bar{w}) log P(c_i, \bar{w}) \end{aligned} \tag{2}$$

Here, $P(c_i)$ represents the probability of the category, c_i, $P(w)$ is the probability that word w appears and $P(\bar{w})$ the probability that word w doesn't appears in documents. $P(c_i, w)$ is the conditional probability of category c_i that word w has appeared, and $P(c_i, \bar{w})$ is the conditional probability of category c_i that word w has not appeared.

2) CHI-SQUARE

Chi-square (CHI) measures independency between two variables, category c and feature w. The greater the value of the CHI, the more information about category c the feature w contains. The CHI formula is defined as follows [39]:

$$CHI(w, c) = \frac{N(AD - CB)^2}{(A + C)(B + D)(A + B)(C + D)} \tag{3}$$

Here A is the number of times word w is in category c, B is the number of times w occurs without c, C is the number of times c occurs without w, D is the number of times where neither w and c appear, and N is the total number of comments.

3) MUTUAL INFORMATION

Mutual information (MI) measures the dependency between the feature w and the category c, where $P(w, c)$ is the probability that feature w occurs in category c. The higher value means more information about category c the feature contains [39]:

$$MI(w, c) = \log \frac{P(w, c)}{P(w)P(c)} \tag{4}$$

D. FEATURE ENHANCEMENT

Our feature enhancement method contains three steps. In the first step, the top k% of features are selected with the highest FS score. In the second step, the cosine similarity score between each feature f and all other features from the unlabeled data set from github is calculated using the prepared *word2vec* model. In the last step, the top t semantically most similar words for each feature are selected and added to the feature list.

We ran some preliminary experiments, to determine the best possible value for t. We examined how many features would be added to the original set, using a different value for t (ranged from 1 to 7,), using CHI as the FS method. Figure 2 shows the percentage of additional features added to the original feature set. The percentage of added features is determined as the total number of additional features not contained in the original feature set, divided by the theoretical maximum for t, where $max = k * t + k$. As we were trying to select the value for t, which would still add enough new features to make an implication in our method, we did not consider $t = 1$, since the addition of features in such manner would be extremely low – less than 30% of additional features would be added even for extremely small initial feature sets (for $k = 0.1\%$) and then dropping below 20% already for $k = 1\%$. All other values ($t > 1$) add a much greater percentage of new features, more than 40% for $k = 0.1\%$, while the percentage drops similarly for larger k. Finally, we chose 2 and 5 as values for t, since $t = 2$ adds the highest percentage of features for $k >= 10\%$, while $t = 5$ adds the most features at the smallest initial set ($k = 0.1\%$) and in other cases a very similar percentage of features as $t = 6$ and $t = 7$. As the addition of too many features with lesser similarities induces noise, which could hurt the performances of classification, we decided to use only $t = 5$. At last, $t = 2$ and $t = 5$ add a number of features, distinct enough, so that the comparison of these two would make sense. We examined the impact of using these different values for t in our experiments. A detailed pseudo code for our method is shown in Algorithm 1.

For example, let's have a list of five most important features, as ranked by an FS method: *todo*, *fixm*, *stupid*, *hack*, *ugli*. For each of these features, we search for the most semantically similar features from the *word2vec* model. Table 2 shows two ($t = 2$) most similar words for each feature, ranked by their cosine similarity score. Our base feature set is enhanced with the following additional features: *xxx*, *here*, *ugli*, *better*, *actual*, *workaround*, *ineffic*.

By employing this method, we are able to select and add more representative features since we use semantic information about features from additional unlabeled corpora of

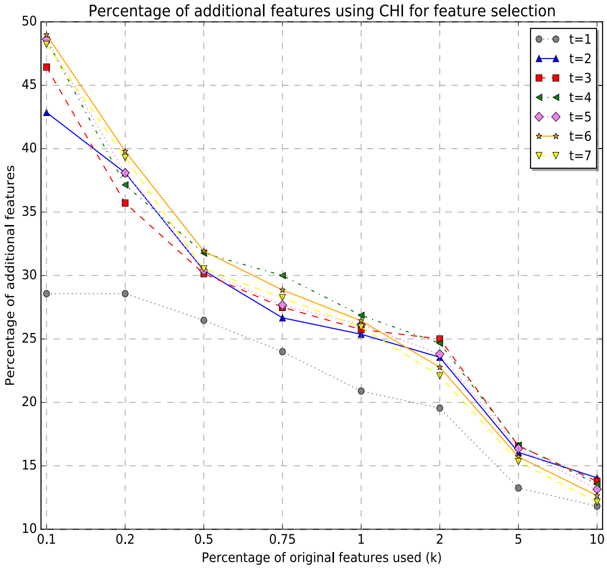


FIGURE 2. Comparison of the percentage of added features using a different value for *t*.

Algorithm 1 FS Enhancement

```
Input:
    $word2vec$: a learned word embedding model
    $features$: a list of features selected by FS method
    $t$: number of most similar words
Output: $features_{enhanced}$
procedure FEATUREENHANCEMENT
    $features_{enhanced} \leftarrow features$
    for each word $f$ in $features$ do
        $w_{sim} \leftarrow$ find $t$ most similar words in $word2vec$
        for each word $w$ in $w_{sim}$ do
            if $w$ is in not in $features$ then
                add $w$ to $features_{enhanced}$
            end if
        end for
    end for
    return $features_{enhanced}$
end procedure
```

TABLE 2. Most similar words from trained *word2vec*.

Feature	Two (*t*=2) most similar features
todo	xxx, here
stupid	ugli, better
fixm	todo, actual
hack	fix, workaround
ugli	ineffic, hack

comments using *word2vec*. These features are then used for training a text classifier.

E. TRAINING CLASSIFIER

The last stage in the text classification process is to train a text classifier. A text classifier is trained with the provided training documents. We train a classifier with the labeled source code comments [19]. We use BOW to represent each comment with a word vector, where the commonly used *tfidf* term-weighting scheme [32] is applied to adjust the count of a word based on its frequency in the entire corpus. We only consider the features, provided by our feature selection and enhancement method. Our work can be utilized with any supervised text classification algorithm. In this paper, we investigated the performance of three very common classification methods, Naive Bayes (NB), Support Vector Machine (SVM) and Maximum Entropy (ME).

F. WORD EMBEDDINGS

In order to implement our proposed FS enhanced method, we first trained word embedding, using the Gensim Pyhon library, which implements Mikolov's word2vec skip-gram model [30]. The skip-gram model learns vector representation of words that are useful for predicting surrounding words in a text. To provide the learning corpora to train the *word2vec* model, we retrieved the most popular 360 open source projects from GitHub[1] using Github API.[2] To select the most popular projects, we filtered out all non Java projects,[3] and projects that didn't have at least 3 forks and 100 rating stars, since it is very likely that repository with a high number of stars contains relevant software engineered project [26]. We ordered projects by the number of stars and selected the top 360 projects. Examples of selected projects are: $ReactiveX - RxJava$, $elasticsearch$, $square - retrofit$, $square - okhttp$, $google - guava$, $spring - boot$, $bumptech - glide$, $JakeWharton - butterknife$, $kdn251 - interviews$, $airbnb - lottie$, etc. Python's `comment_parser` library[4] was used for extracting text comments from .java source files. Using simple heuristic, license comments, and comments with less than three words were ignored, since this type of comments would not provide quality training data to train word embeddings. Fig. 3 shows the main steps in the comments extraction process.

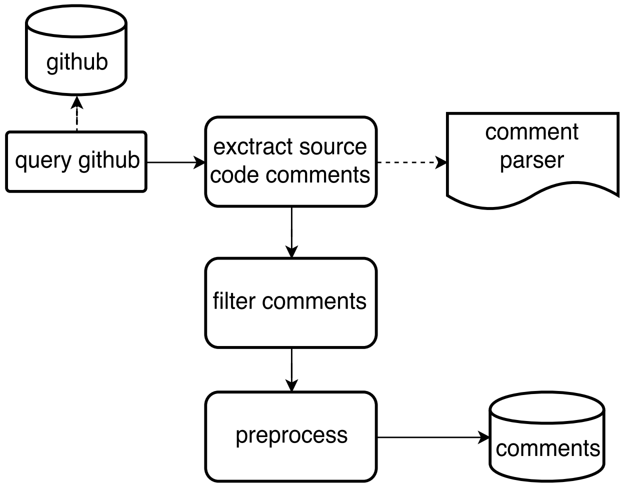


FIGURE 3. Comments extraction process.

Each comment represents a single text document in our learning corpora. There were approximately 1.2 M comments in the retrieved 360 open source Java projects. Before all comments were passed into the *word2vec* model, we preprocessed comments to strip them of all punctuations, non-alphanumeric characters, and extensive white spaces, using the preprocessing method described above.

Vector size for representing a word in the *word2vec* model was set to 400. All the words were ignored with term frequency lower than two. The window size for the surrounding word context was set to four. These settings are based on some limited preliminary experiments, and are similar to previously reported configurations used to train the word embeddings [14], [22], [40], and were not optimized.

After training, we obtained an embedded model with 110,760 unique tokens (words), where words with similar meanings are embedded into neighboring semantic space. Once word vectors were embedded into semantic space, we could apply our enhancement method to add additional features to the feature set. It takes about 20 minutes on a Windows 10 based computer, 3.60 GHz Intel Core i7 CPU and 32GB RAM, to train a word embedding model and save it to the hard disk. The size of the model file is 1.4GB, and it can fit in the main memory.

1) PRE-TRAINED WORD EMBEDDINGS

Instead of training our own word embedding model, we could use some of the existing pre-trained word embedding models. The advantage of pre-trained word embedding models is generally the huge amount of text, upon which the model was built. The disadvantage of pre-trained word embeddings, on the other hand, is that the words contained within may not capture the peculiarities of language in a specific application domain. For example, although being huge, the Wikipedia may not have great word exposure to particular aspects of legal doctrine, religious texts, or source code comments for that matter, so if an application is specific to a domain like this, the results may not be optimal due to the generality of the downloaded model's word embeddings.

An often used pre-trained word embeddings model is GoogleNews.[5] This model was trained on 100 billion words from Google News dataset using 300-dimensional

[1] https://github.com/

[2] https://github.com/PyGithub/PyGithub

[3] projects without .java source files

[4] https://pypi.python.org/pypi/comment_parser

[5] https://code.google.com/archive/p/word2vec/

vector [23]. Although this model is useful in a variety of NLP tasks it is not specific enough in the domain of code comments, since, despite 3 million word vocabulary, many of the words like *todo*, *fixme*, *hack*,... were not found in this model. Other studies also suggest that training a word embedding on a specific domain is better than using pre-trained models [13].

To confirm this, we run some preliminary experiments where we compared the performance of pre-trained GoogleNews model with our model trained on source code comments using our feature enhancement method. Figure 4 shows the classification results, using our feature enhancement method with different word embeddings. Our word embedding model, trained on source code comments, outperformed publicly available word vectors trained in a different domain.

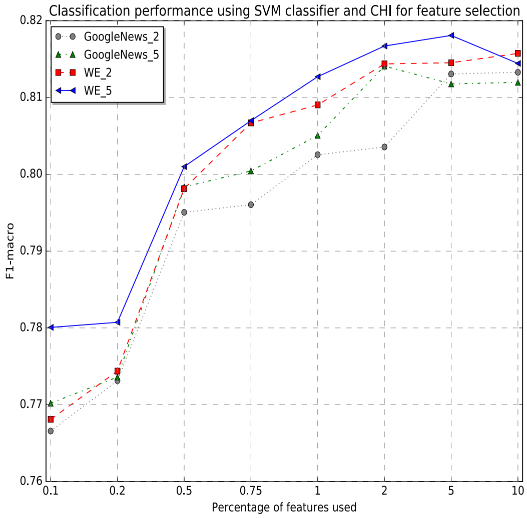


FIGURE 4. Comparison of classification performances of different word embedding models.

IV. EXPERIMENT SETUP

In this section, we describe the experimental setup that we followed to evaluate the performance of the proposed approach. We first describe the data collection that was used for performing the experiments. Then we list the baseline FS methods, feature enhancement settings, and classification algorithms, which were used to assess our proposed method. Finally, we explain the evaluation method and classification performance metric. The obtained experimental results and research questions are presented and discussed in the next section.

A. DATASET

To evaluate our proposed method, we obtained the SATD data provided by the authors in [19]. This publicly available dataset contains comments from ten open source software projects: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and SQuirrel. Projects vary in size and the number of contributors and belong to different application domains. The dataset contains 39,033 unique comments and their labels. Comments are labeled as five different types of SATD. Since our aim is to detect SATD (i.e. to identify whether a comment contains an indication of technical debt or not), we joined all SATD types into one single category – whether a comment includes SATD or not. There are a 3,298 comments that contain SATD, which is 8% of all comments. Table 3 lists detailed information about the used dataset.

B. BASELINE AND COMPARED METHODS

In order to assess the performance of the proposed method and its applicability to the identification of self-admitted technical debt, a series of tests has been carried out, in which we measured text classification performance. As baseline methods, we applied three different FS methods: CHI, IG, and MI.

We tested all three baseline methods with various numbers of initially selected features, ranged from 0.1% to 100% of all features. Feature sets, selected by the baseline FS methods, were then enhanced using our proposed method with additional features using a similarity measure based on the trained word embedding vector space model. We used two different enhancing values and set the parameter t as 2 and 5, which means that either 2 (WE2, word embedding method with 2 additional words) or 5 additional words (WE5), most similar to the words obtained by the baseline FS method, were added as additional features to enhance the set of selected features. Those features were then used by a text classifier.

To validate the proposed feature enhancement method using our word embedded model, we also used two existing methods, that compute the semantic relatedness of words in a different manner. The first is WS4J (WordNet Similarity for Java) library,[6] which implements several known algorithms for computing semantic relatedness of words using WordNet.[7] For the similarity algorithm, we selected Resnik [31], which uses information content of shared parents of words in WordNet, where two concepts are more similar if they present more shared information. For a fair comparison, we used this method to also add either 2 (R2) or 5 most similar words (R5). The second method is SEWordSim [35], which is a software specific word similarity database, constructed from StackOverflow.[8] It contains similarity information of more than 5M word pairs, related to the software engineering domain. Again, also using this method, we added either 2 (SE2) or 5 most similar words (SE5).

The baseline FS methods do not depend on the learning model. Thus, as a text classifier, we used the support vector machine (SVM), Naïve Bayes multinomial (NB) and Maximum Entropy (ME), since they are some of the most effective and widely used classification algorithms in text classification tasks [39]. The SVM, NB and ME were employed

[6] https://github.com/Sciss/ws4j
[7] https://wordnet.princeton.edu/
[8] https://stackoverflow.com/

TABLE 3. SATD dataset statistic.

Project	Comments	SATD comments	% SATD	Avg. tokens per doc.	
				preprocess	original
Ant	3138	124	3.95	10.13	13.36
ArgoUML	5706	1149	20.13	13.62	18.30
Columba	4329	163	3.77	7.32	10.64
EMF	2843	82	2.88	11.22	16.13
Hibernate	2590	428	16.15	11.75	15.10
JEdit	4759	235	4.94	6.41	11.30
JFreeChart	2629	109	4.15	7.70	11.11
JMeter	4254	318	13.4	9.71	13.15
JRuby	3822	462	12.09	8.87	11.93
SQuirell	4962	228	4.59	11.55	15.39
Total	**39,033**	**3298**	**8.45**	**9.83**	**13.64**

to investigate the contributions of the selected features to classification performance.

In this manner, if the proposed method can truly enhance a baseline FS method, it should perform well in this task and improve the baseline classification results. Furthermore, the comparison with the existing methods for computing semantic relatedness of words, should show how well the proposed method shall perform in practice. By using three different baseline FS methods (CHI, IG, and MI), three different classification algorithms (NB, SVM, and ME), and two different enhancement values (2 and 5), we should be able to evaluate the proposed method comprehensively.

C. EVALUATION METHOD AND METRIC

Considering the imbalance of the data set, we performed the evaluation using the $F1$ measure averaged over two classes – the $F1$-macro. The $F1$ score is computed for each class within the data set and then the average is obtained over all classes. In this way, equal weight is assigned to each class regardless of the class frequency. The $F1$ score can be interpreted as a harmonic mean of the precision and recall, where precision is the number of correct positive predictions divided by the number of all positive predictions, and recall is the number of correct positive predictions divided by the number of all positive instances:

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{5}$$

To test the proposed method objectively, we adopted the 10-fold cross validation approach, where each fold contains all comments from a single open source project. In this manner, within one fold, we trained the classification model using the comments from 9 out of 10 folds (the comments from 9 projects) and then tested the model on the remaining fold (the comments from the remaining project, which were not used for training). All the results reported are the averaged $F1$-macro scores, obtained on the test comments, over all 10 folds, if not specified otherwise.

V. RESULTS

In this section, we present the obtained experimental results, which are divided into subsections in accordance with the defined research questions. In our study, we investigated the following research questions:

- RQ 1: Can the classification performance of SATD be improved by enhancing the features obtained with a selected FS method, with our proposed method?
  - RQ 1.1: What is the classification performance when using different feature selection approaches (CHI, IG, and MI) and different classification algorithms (NB, SVM, and ME)?
  - RQ 1.2: How does our proposed feature enhancement method compare with other existing methods?
  - RQ 1.3: Are there any differences in identification of SATD between different feature enhancement settings?
- RQ 2: How much of SATD is reported in open source projects?
  - RQ 2.1: Is there a set of words that determines the SATD in source code comments?
  - RQ 2.2: Which words are the most similar in source code comments?

A. ANALYSIS OF CLASSIFICATION PERFORMANCE USING DIFFERENT FS APPROACHES AND DIFFERENT CLASSIFICATION ALGORITHMS

In order to evaluate our proposed method objectively, we compared its two variants (either adding 2 or 5 most similar features) with the baseline (when no feature enhancement is used) and two existing methods, already shown to perform well on software engineering problems (SEWordSim and WS4J-Resnik; these two methods have been used for exactly the same purpose as our proposed method – to add either 2 or 5 most similar features). For this purpose, we performed a series of experiments with three major feature selection approaches (CHI, IG, MI) and three classification algorithms (NB, SVM, ME) – for each combination of FS approach and classification algorithm, all the compared methods have been used to identify SATD in 10 selected software engineering projects.

1) USING CHI AS THE UNDERLYING FS APPROACH

Fig. 5 shows SATD classification results, obtained with the NB classifier when CHI wa used for feature selection.

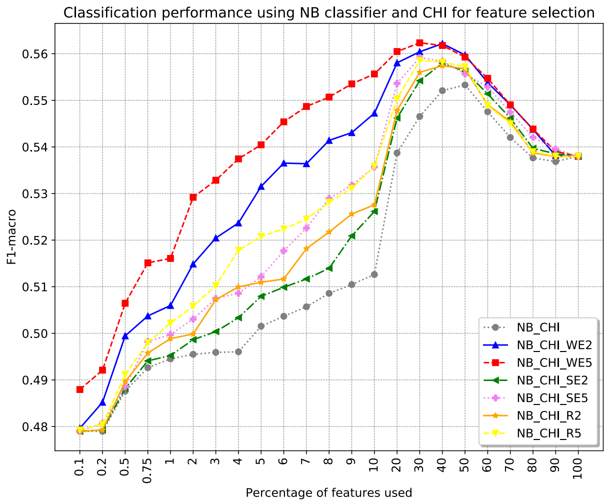


FIGURE 5. **Comparison of seven methods when using NB classification algorithm and CHI for feature selection.**

The seven methods are compared – baseline CHI (NB_CHI), baseline CHI enhanced with 2 most similar words for each selected feature using our proposed method (NB_CHI_WE2), baseline CHI enhanced with the 5 most similar words for each selected feature using our proposed method (NB_CHI_WE5), baseline CHI enhanced with 2 (NB_CHI_SE2) and 5 (NB_CHI_SE5) most similar features using the SEWordSim method, and baseline CHI enhanced with either 2 (NB_CHI_R2) or 5 (NB_CHI_R5) most similar features using the WS4J-Resnik method.

We can see that all seven methods behave very similar to the varying amount of features that are used to train the classifier. The classification performance ($F1$-score) increases continuously from the beginning (0.1% of features used) and up to approximately 20% of used features. After that, the predictive performance starts to drop continuously until all the features are used, with the highest peak at around 30% of used features.

Fig. 5 shows that our two enhancement methods (NB_CHI_WE2 and NB_CHI_WE5) outperformed the baseline NB_CHI across the whole range, especially NB_CHI_WE5 seems to have the advantage. We can see that the four compared enhancement methods (NB_CHI_SE2, NB_CHI_SE5, NB_CHI_R2, NB_CHI_R5) also outperformed the baseline NB_CHI, but at the same time lag behind our proposed method. The best overall $F1$-score was achieved by NB_CHI_WE5, while using the top 30% of features. After this point, classification performance starts to decline a bit. When all features (100%) are used, the result of all the methods is the same, since all methods use the complete set of all features and there is nothing to enhance. As can be seen, the classification performance of our two proposed enhanced methods is improved over the rest in almost every measurement.

To evaluate the statistical significance of these results, we first applied the Friedman test as suggested by Demšar [6] by calculating the asymptotic significance for the seven compared methods for the whole range of measurements (0.1% to 100% of features used). As the results are not normally distributed, a non-parametric statistical test was chosen. Furthermore, the obtained results of each method are related regarding the initial set of selected features. Namely, the same baseline FS approach is used to provide the initial set of features, on which the methods operate next (by adding the most similar features in their own, different ways). Consequently, the Friedman test was applied, which is a non-parametric statistical test used to detect differences in the results of various methods across multiple test attempts. The results of the performed Friedman test show that differences between methods are statistically significant ($p < 0.001$).

To test further whether the results of our proposed method NB_CHI_WE5, which show the biggest improvement, are indeed significant, the Wilcoxon signed-rank test was applied next, as suggested by Demšar [6], to compare NB_CHI_WE5 with the remaining six methods (the Holm-Bonferroni correction was applied). The Wilcoxon signed rank test is a non-parametric alternative for the paired T-Test, which can be used to compare the statistical equality of two methods over the same sample. It tests whether the difference of achieved ranks of the two methods is statistically significant [25]. In our case, the Wilcoxon test was used to compare the results, achieved by our proposed method NB_CHI_WE5, with each of the remaining methods, one by one. If the Wilcoxon test resulted in statistically significant difference between the two methods, the one with the higher average rank has been regarded as the better method. The method NB_CHI_WE5 achieved the best average rank among seven methods ($rank = 6.78$ on the scale from 1 being the worst result up to 7 being the perfect score), and it turned out that the results of NB_CHI_WE5 are indeed significantly better than the results from the other methods ($p = 0.0001$ when compared to NB_CHI_WE2, and $p < 0.0001$ for all the rest).

Similarly, Fig. 6 shows a comparison of SATD classification results of seven methods, obtained with the SVM classifier, and Fig. 7 the results obtained with the ME classifier, in both cases when CHI has been used for feature selection. We can see that in both cases all seven methods behave very similarly with the varying amount of features that are used to train the classifier, and a bit different than when using the NB for classification. In these two cases, the classification performance ($F1$-score) increases rapidly at the beginning, when only a fraction of features are used (from 0.1% to approx. 5%). After that, the predictive performance remains quite stable until all the features are used, with the highest peak again at around 30% of used features.

In these two cases, the dominance of our proposed method is not so obvious across the whole range of measurements (from 0.1% to 100% of features used). Nevertheless, the Friedman tests show that the results of different methods are again statistically significant ($p < 0.001$ in the case of SVM_CHI, and $p = 0.039$ in the case of ME_CHI).

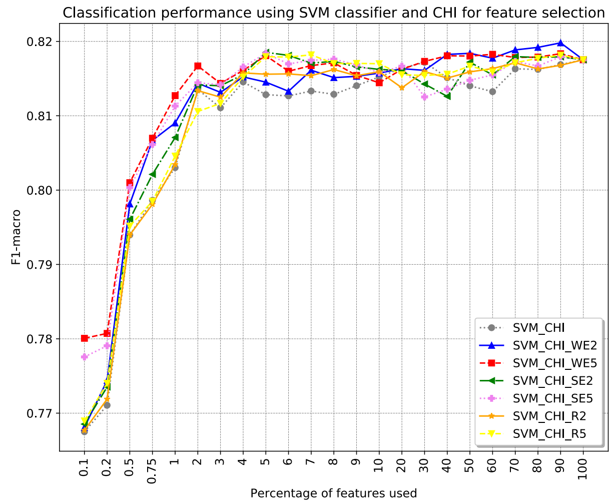


FIGURE 6. Comparison of seven methods when using SVM classification algorithm and CHI for feature selection.

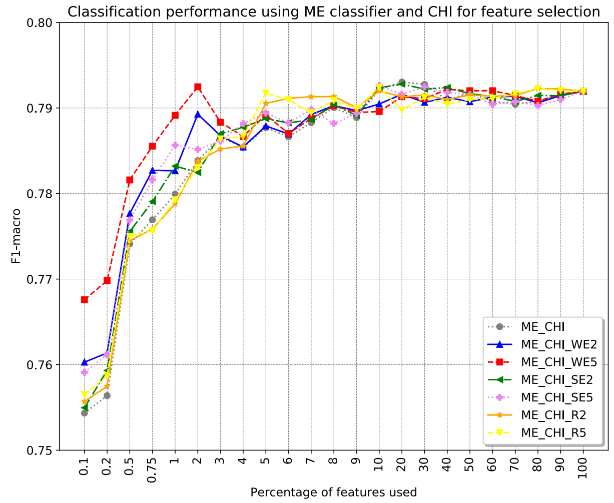


FIGURE 7. Comparison of seven methods when using ME classification algorithm and CHI for feature selection.

In both cases, again our proposed method that adds the 5 most similar features achieved the highest rank among the seven compared methods (*rank* = 5.70 for SVM_CHI_WE5, and *rank* = 5.00 for ME_CHI_WE5, see Table 4 for details). In the case of using CHI as an FS approach and SVM for classification, the results of SVM_CHI_WE5 are significantly better than those from all other methods: baseline ($p < 0.0001$), SVM_CHI_WE2 ($p = 0.024$), SVM_CHI_SE2 ($p = 0.0225$), SVM_CHI_SE5 ($p = 0.048$), SVM_CHI_R2 ($p = 0.0001$), and SVM_CHI_R5 ($p = 0.0074$). In the case of using CHI as a FS approach and ME for classification, the results of ME_CHI_WE5 are significantly better than those from baseline ($p = 0.0074$), ME_CHI_WE2 ($p = 0.0027$), and ME_CHI_SE5 ($p = 0.0392$), while not being significantly different from those from ME_CHI_SE2 ($p = 0.1396$), ME_CHI_R2 ($p = 0.2234$), and ME_CHI_R5 ($p = 0.1485$).

2) USING IG AND MI AS THE UNDERLYING FS APPROACHES

Similar analysis has also been performed for the other two FS approaches, namely IG and MI. Generally, the findings are very similar to those, as presented in the previous subsection. For this purpose, only an overview of results will be presented in the next sub-section.

3) COMPARISON OF OUR PROPOSED METHOD WITH BASELINE AND THE EXISTING ONES

Table 4 presents the average ranks of the seven compared methods as achieved on each of 9 possible combinations of FS approach and classification algorithms. The average rank for each method is calculated as an average across all ranks, achieved by that method on all data points (SATD classification results when from 0.1% to 100% of features are used). The best rank is 7 (when a method achieves the best score among all methods), while the worst rank is 1. The last row of Table 4 presents an overall average of ranks, achieved by each of the seven compared methods, on all data points with all combinations of FS approaches and classification algorithms. It can be seen, that our proposed method WE5 achieved the best overall average rank and the best average rank in 6 out of 9 combinations, while achieving the second best average rank in the other three testing combinations. The second best overall average is achieved by the method SE5, followed narrowly by WE2. All other methods lag quite far behind, with the baseline being the least successful of all methods, as expected.

Another interesting presentation of SATD classification results of seven compared methods, based on achieved ranks, is provided in Fig. 8. It can easily be seen that our proposed method WE5 achieved the most wins (it was the best in 83 out of 207 possible cases). It is also interesting, that the frequency descends steadily with the descending ranks, which is not the case for any other method. On the other hand, the baseline is the only method, which ranks frequencies steadily ascend with the descending ranks. When looking at the SE5 method, as the one that is closest to WE5, we can see that it has quite some wins (51, compared to 83 wins of WE5), but also quite a lot of the worst results (20, which is twice as much as WE5).

The statistical tests, as described above for the case of using CHI and NB, have been performed to validate the significance of the results. In Table 5, the results of performed statistical tests are presented. The first row denotes the testing situation (combination of FS approach and classification algorithm). The second row presents the asymptotic significance (p-value) of the performed Friedman test – if the p-value is less than 0.05, then there are significant differences between seven methods. The rows 3 through 8 present the asymptotic significance (p-value) of the performed post-hoc Wilcoxon test for comparing our proposed WE5 with the other six methods – if the p-value is less than 0.05, then there are significant differences between WE5 and the compared method. In case of significant differences, the average ranks of the

TABLE 4. The average ranks of seven methods when classifying SATD using different FS approaches and classification algorithms (the best average ranks are shown in bold).

	baseline	WE2	WE5	SE2	SE5	R2	R5
CHI+NB	1.22	6.04	**6.78**	2.57	4.28	2.96	4.15
CHI+SVM	2.04	4.65	**5.70**	4.24	4.80	2.67	3.89
CHI+ME	2.78	3.83	**5.00**	4.13	4.13	4.13	4.00
IG+NB	1.35	5.43	**6.83**	3.26	4.83	2.24	4.07
IG+SVM	1.83	4.61	5.35	4.67	**6.02**	2.59	2.93
IG+ME	3.43	3.39	4.39	4.13	**4.57**	3.87	4.22
MI+NB	3.50	4.59	**5.33**	3.59	4.07	3.59	3.35
MI+SVM	3.48	4.70	**4.83**	4.54	4.80	2.52	3.13
MI+ME	3.22	2.93	4.59	4.35	**5.74**	3.52	3.65
overall average	2.54	4.46	**5.42**	3.94	4.80	3.12	3.71

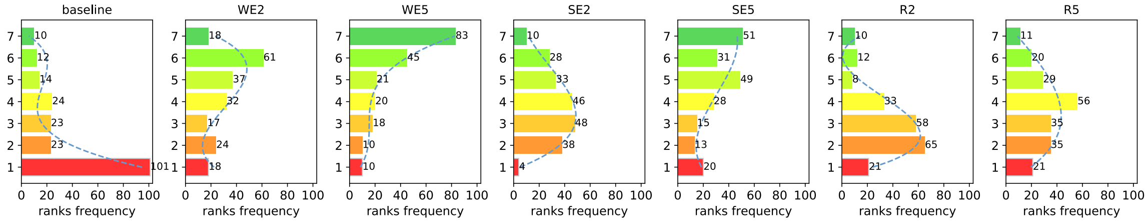


FIGURE 8. Comparison of seven methods regarding the achieved ranks on all 9 testing combinations – using three FS approaches and three classification algorithms.

TABLE 5. The results of the statistical comparison of seven methods – Friedman's asymptotic significance (2nd row) for comparing differences between methods, and Wilcoxon's asymptotic significance (rows 3–8) for comparing our proposed method WE5 with other methods (significant differences are marked with *).

	Friedman	baseline	WE2	SE2	SE5	R2	R5
CHI+NB	*0.0000	*0.0000	*0.0001	*0.0000	*0.0000	*0.0000	*0.0000
CHI+SVM	*0.0000	*0.0001	*0.0240	*0.0225	*0.0480	*0.0001	*0.0074
CHI+ME	*0.0399	*0.0074	*0.0027	0.1396	*0.0392	0.2234	0.1485
IG+NB	*0.0000	*0.0000	*0.0000	*0.0000	*0.0000	*0.0000	*0.0000
IG+SVM	*0.0000	*0.0000	0.1154	0.1361	0.0727	*0.0002	*0.0007
IG+ME	0.3790	–	–	–	–	–	–
MI+NB	*0.0072	0.0537	*0.0355	*0.0015	0.6894	*0.0013	*0.0004
MI+SVM	*0.0002	*0.0074	0.1677	0.3615	0.3458	*0.0011	*0.0014
MI+ME	*0.0000	*0.0071	*0.0012	0.3219	0.2586	*0.0033	*0.0078

compared methods show which method is better (the one with the highest rank, see Table 4).

As we can see, the only testing situation, where there are no significant differences between the seven methods, is the combination IG + ME – otherwise, the methods differ significantly. For the combinations CHI + NB, CHI + SVM, and IG + NB, the WE5 is significantly better than all other methods. For the combination CHI + ME, WE5 is significantly better than baseline, WE2, and SE5. For combination IG + SVM, WE5 is better than baseline, R2, and R5. For the combination MI + NB, WE5 is better than WE2, SE2, R2, and R5. For combination MI + SVM, WE5 is better than baseline, R2, and R5. Finally, for the combination MI + ME, WE5 is better than baseline, WE2, R2, and R5. In all the testing situations, our proposed method WE5 is never significantly worse than any other compared method.

B. COMPARING FEATURE ENHANCEMENT SETTINGS ON SELECTED OPEN SOURCE PROJECTS

In the previous section, all the presented results provide enough evidence in order to consider our proposed method WE5 as the most successful for identifying SATD among the compared methods in general. In this section, we will compare all the methods on a set of 10 open source projects, using the general findings from the above performed analysis. This should equip a software engineer with further insights into how different feature enhancement settings influence the identification of SATD.

WE5 achieved the best overall result using MI as the FS approach and SVM for classification when 30% of features were used ($F1$-score = 82.12%). In general, the SVM classifier achieved slightly better results than ME and much better than NB, regardless of the used FS approach. The differences between different FS approaches, when using the same classification algorithm, were only marginal. Furthermore, generally, the best results were achieved with the feature set sizes of approx. 30%. In this manner, we constructed a classification model for each open source project using 30% of features as selected by each baseline FS approach and trained it with all the comments from the remaining nine projects.

1) THE INFLUENCE OF THE CLASSIFICATION ALGORITHM

First, we analyzed the influence of different classification algorithms (NB, SVM, and ME) on the results. For each testing situation (different classification algorithm), the SATD

TABLE 6. The average ranks of seven methods when classifying SATD using different classification algorithms (NB, SVM, ME) on ten selected open source projects (the best average ranks are shown in bold).

	baseline	WE2	WE5	SE2	SE5	R2	R5
NB	2.75	4.72	**5.33**	3.75	4.45	3.22	3.78
SVM	3.82	4.22	**4.63**	4.45	3.68	3.73	3.47
ME	4.30	3.75	4.12	4.10	**4.53**	3.80	3.40
overall avg	3.62	4.23	**4.69**	4.10	4.22	3.58	3.55

TABLE 7. The average ranks of seven methods when classifying SATD using different underlying FS approaches (CHI, IG, MI) on ten selected open source projects (the best average ranks are shown in bold).

	baseline	WE2	WE5	SE2	SE5	R2	R5
CHI	3.37	4.17	**4.68**	3.93	4.12	3.85	3.88
IG	2.83	4.72	**5.13**	3.93	4.92	3.08	3.38
MI	**4.67**	3.80	4.27	4.43	3.63	3.82	3.38
overall avg	3.62	4.23	**4.69**	4.10	4.22	3.58	3.55

classification results for all three underlying FS approaches (CHI, IG, MI) were used, for all 10 open source projects.

When using NB for classification, the results of our proposed method WE5 outperformed all other methods. The Friedman test confirmed the significant difference ($p < 0.001$). As the best average rank (5.33) was achieved by the WE5, we performed the post-hoc Wilcoxon test to compare WE5 with all other methods. The results show that WE5 is indeed significantly better than all other methods: baseline ($p = 0.0001$), WE2 ($p = 0.0081$), SE2 ($p = 0.0014$), SE5 ($p = 0.0201$), R2 ($p = 0.0006$), and R5 ($p = 0.0039$). On the other hand, when using either SVM or ME for classification, the differences between methods on 10 selected projects are only marginal. This was further confirmed by the Friedman tests, which show no significant difference in the case of SVM ($p = 0.2731$) and in the case of ME ($p = 0.2989$).

Next, we calculated the average ranks of all seven methods on three classification algorithms, which are summarized in Table 6. We can see, that our proposed method WE5 achieved the highest rank with two classification algorithms (NB and SVM) while being the second best with ME. Consequentially, WE5 also achieved the highest overall average rank. In the case of the ME classifier – the only one where WE5 did not achieve the top rank – the differences of average ranks between the methods were the smallest. It is also interesting, that the second highest overall rank was achieved by WE2, even slightly in front of the best of the compared existing methods SE5. The comparison of seven methods regarding the achieved average ranks on ten selected open source projects using three classification algorithms is presented visually in Fig. 9. Interestingly, when using ME for classification, all the methods tended to have the average rank around the mid-point at rank 4.00, which confirms that differences in this case are indeed marginal.

2) THE INFLUENCE OF THE FS APPROACH

Similarly, we analyzed the influence of different FS approaches (CHI, IG, and MI) on the results. For each testing situation (different FS approach), the SATD results of all three classification algorithms (NB, SVM, ME) were used, for all 10 open source projects.

When using IG as the underlying FS approach, the results of our proposed method WE5 outperformed all other methods. The Friedman test confirmed the significant difference ($p < 0.001$). As the best average rank (5.13) was achieved by the WE5, we performed the post-hoc Wilcoxon test to compare WE5 with all other methods. The results show that WE5 is indeed significantly better than the following methods: Baseline ($p = 0.0004$), SE2 ($p = 0.0225$), R2 ($p = 0.0014$), and R5 ($p = 0.0057$), while the results were not significantly better than those from WE2 ($p = 0.1442$) and SE5 ($p = 0.3001$). On the other hand, when using either CHI or MI as the underlying FS approach, the differences between methods on 10 selected projects are only marginal. This was further confirmed by the Friedman tests, which show no significant difference in case of CHI ($p = 0.3706$) and in case of MI ($p = 0.0905$).

If we take a look at the calculated average ranks of all seven methods using three FS approaches (Table 7), we can see that our proposed method WE5 achieved the highest rank with two FS approaches (CHI and IG), while being the third best with MI. As already shown above, WE5 also achieved the highest overall average rank. In the case of MI as the FS approach – the only one where WE5 did not achieve the top rank – the differences of average ranks between the methods were the smallest. It is very interesting, that in the case of using ME the best average rank was achieved by the baseline method. The comparison of seven methods regarding the achieved average ranks on ten selected open source projects using three FS approaches is presented visually in Fig. 10.

C. HOW MUCH OF SATD IS REPORTED IN OPEN SOURCE PROJECTS?

For the identification of the SATD comments, we used our proposed method. Since the best performances were obtained with the SVM_MI_WE5, we used this model to classify comments as SATD (or not SATD). We mined the same projects that were used for creating the word embedding model. We filtered out projects, from our analysis, which contained less than 10 source files, have less than 100 comments and less than 1000 LOC. That leaves us with 246 projects. The obtained projects varied in size (LOC), in the number of source files and in number of comments. The detailed information about the analyzed projects is reported in Table 8.

On average, the analyzed projects have around 90k LOC and 4.7k comments. The largest project is *Intellij-community*[9] with 3.4M LOC which also has the most .java files. Despite being the largest, it has fewer comments than the second largest project *platform_frameworks_base*,[10] which has 126,835 comments.

[9] https://github.com/JetBrains/intellij-community

[10] https://github.com/aosp-mirror/platform_frameworks_base

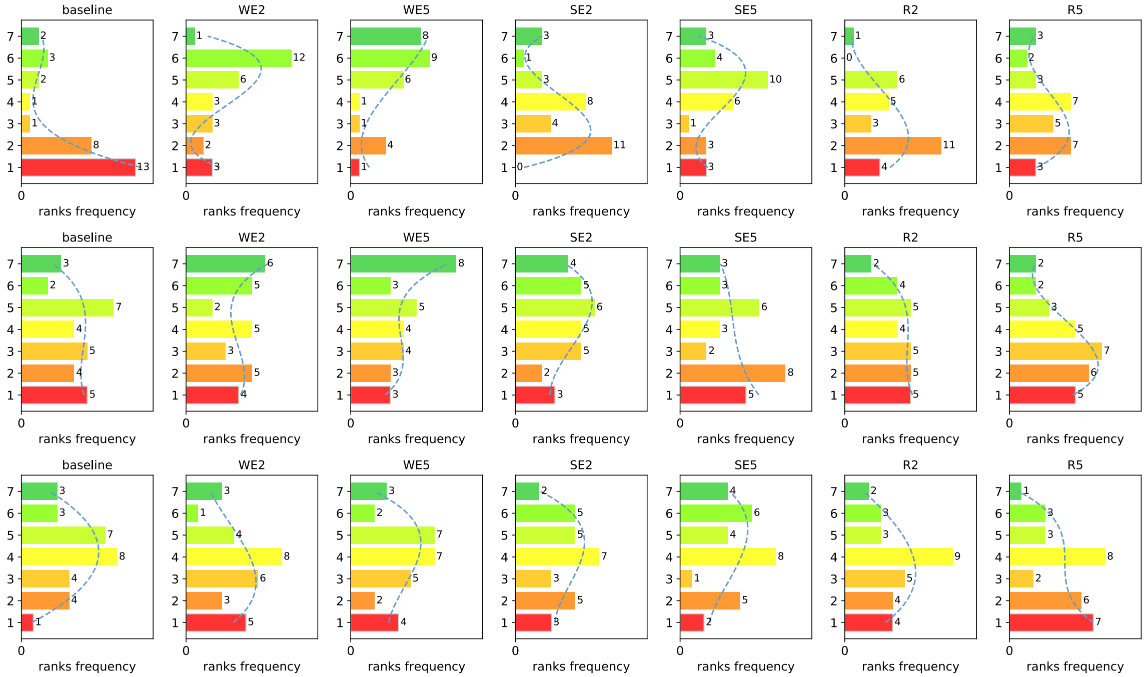


FIGURE 9. **Comparison of seven methods regarding the achieved ranks on ten selected projects using NB (top), SVM (middle), and ME (bottom) for classification.**

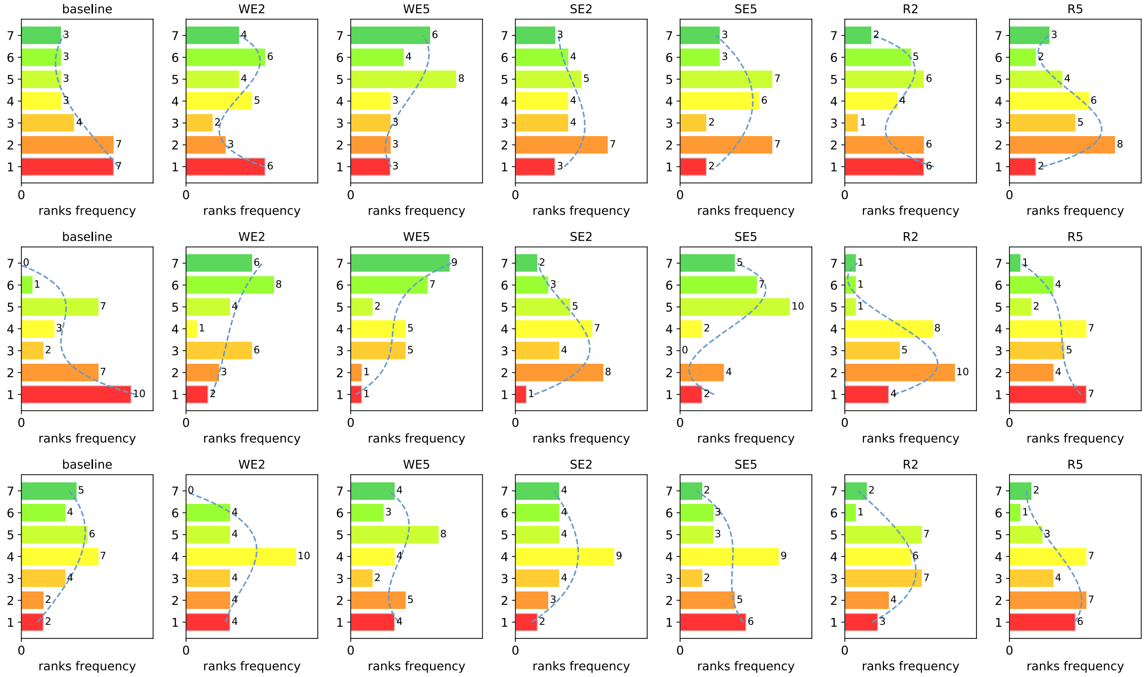
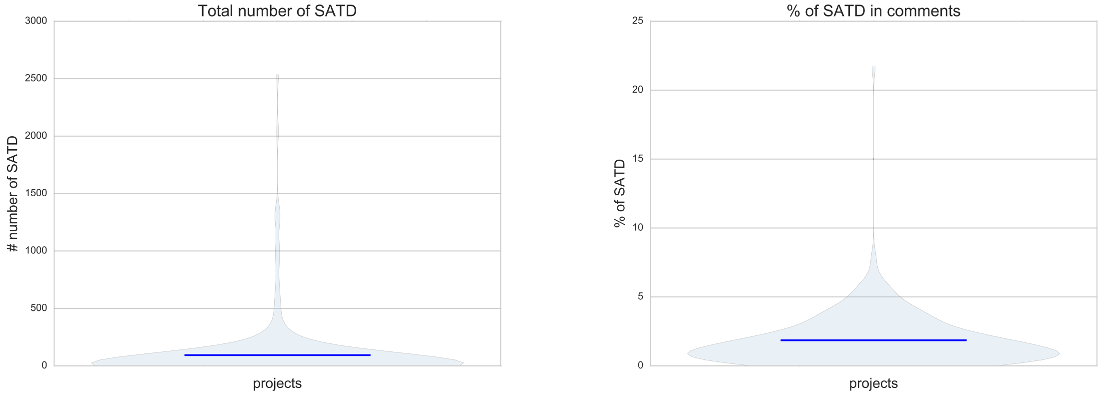


FIGURE 10. **Comparison of seven methods regarding the achieved ranks on ten selected projects using CHI (top), IG (middle), and MI (bottom) as the underlying FS approach.**

There was a total of 1.2M comments which we classified and identified 23,368 of those comments as SATD. That represents 2% of all comments in analyzed GitHub projects. On average, percentage of SATD comments in each project was 1.86% (SD = 1.98). Fig. 11 depicts the diffusion of SATD comments found in the 246 analyzed open source projects. Since different projects may have a substantially different number of comments, we report both the discrete number of SATD found in projects as well as the percentage of comments containing SATD. The first plot (a) reports the

TABLE 8. Characteristics of analyzed GitHub projects.

LOC			.java files			comments		
max	avg	sum	max	avg	sum	max	avg	sum
3 447 064	89 782.17	22M	54 969	897.6	220k	126 835	4 728.7	1.2M



(a) The number of identified SATD in projects.

(b) The percentage of SATD in comments

FIGURE 11. Diffusion of SATD comments. a) Absolute number of SATD, and b) Percentage of comments reporting it.

distribution of the absolute number of SATD comments found in the projects, while (b) shows the percentage of SATD in comments.

On average, analyzed projects had 94.9 SATD comments. The most identified SATD comments were found in the *Platform_frameworks_base* and *Intellij – community* projects, with 2,535 and 2,038 SATD comments, respectively. Although such values look very high, it is important to consider that these two projects contain a total of 126,865 and 61,708 comments, which means that 1.9% and 3.3% of comments report SATD, which is not very far from average. On the other hand, projects with the most percentage of reported SATD in comments are *EffectiveAndroidUI*[11] and *Android – classyshark*.[12] Both are much smaller projects, containing only 2,225 and 9,267 LOC with 106 and 1,639 comments in 54 and 133 java source files, respectively. Their percentages of SATD comments are 21.6% and 7.9%, which are high above average (which is 1.86%). The overall observed trend is in line with some other research reporting SATD in open source projects [1], [29].

It should be emphasized that we did not fully examine whether the SATD comments actually discuss the detected technical debt in source code or not; yet we did look through some of identified SATD comments. Examples of identified SATD, with underlying source code are the following:

- Android-classyshark (FilesTree.java)

```
// hack for manually stripped APKs
    with one flat package
  if (packageNode != null &&
      currentClassesDex.isLeaf()) {
     currentClassesDex.add(packageNode);
  }
```

- Platform_Framework_base (PowerManagerService.java)

```
    // XXX should WorkSource have a
        way to set uids as an int[]
        instead of adding them
    // one at a time?
    ws = new WorkSource();
    for (int i = 0; i < uids.length;
        i++) {
       ws.add(uids[i]);
    }
```

- Intellij-community (ItentionManagerImpl.java)

```
//todo temporary hack, need smarter
    logic:
// * on the first request, wait until
    all the initialization is finished
// * ensure this request doesn't come
    on EDT
// * while waiting, check for
    ProcessCanceledException
if (ApplicationManager.
    getApplication().
    isUnitTestMode()) {
  runnable. run();
}
else {
  myInitActionsAlarm.addRequest(runnable,
      300);
}
```

[11] https://github.com/pedrovgs/EffectiveAndroidUI

[12] https://github.com/google/android-classyshark

- EffectiveAndroidUI (TVShowViewModel.java)

```
 /* This class could be a interface
     implementation if the
 * TvShowViewModel has more than one
 * implementation.
*/
public class TvShowViewModel {
    ...
```

The classification process for identification of SATD comments took about 13 minutes,[13] which is a reasonable time, considering the number of analyzed projects, since the average time was around 3 seconds per project. For example, just to set up software quality tools, like *Sonarqube*[14] or *JDeodorant*,[15] could take up hours, just for one project. With this classification process, we have also created a new large labeled dataset (comments with or without SATD) which would serve for answering RQ 2.1. The detailed information about created dataset is provided in Table 9.

TABLE 9. Results for identification of SATD comment in open source projects.

Comments	SATD comments	% of SATD	Avg. numb. of tokens
1 163 258	23 368	2.00	10.62

D. IS THERE A SET OF WORDS THAT DETERMINES THE SATD IN SOURCE CODE COMMENTS?

To identify common words, used in SATD comments, across multiple GitHub projects, we trained a new SVM classifier with the same settings as used in previous steps. Since SVM is a supervised learning method, we needed labeled information whether comments in those projects contain SATD or not. We used the newly created dataset (see Table 9) from the previous step, to train this classifier. Since an SVM classifier assigns weights to features, where more relevant features, for a specific category, have higher score, we were able to extract the most important features (words) for both category of comments. Table 10 shows comparison of the top-10 features, learned by the SVM classifier, to classify comments to those with or without SATD. The first column shows the list of most important features for identifying SATD comments; the second column lists the features to identify comments without SATD. From the results we can see, that most important features are indeed those that are describing SATD comments. For example, words like *todo*, *xxx* indicate incomplete or missing implementation of code (from *Processing*[16]):

- *//**todo** still not sure why category would be come back null*
- *//**xxx** we can't replace stuff soooooo do something different*

TABLE 10. Top 10 features used by the text classifier to classify comments.

SATD	Not SATD
hack	boolean
todo	close
workaround	construct
fixm	charset
perhap	target
ugli	retain
bug	stub
better	timeout
refactor	word
xxx	archiv

Other words like *workaround*, *fixm*, *refactor*, *hack* indicates poor code quality and temporary solutions (from *Jadx*[17]):

- *//**workaround** for compile bug, see test duplicate cast*
- *//**fixm** add this for equal method in scope,*
- *//**refactor** this boilerplate code*

Words *perhap* and *better* in comments like (from *Small*[18]):

- *//**perhap** start with number*
- *//fixm remove follow thread if you find the **better** place to kill process*

are questioning current solution, and indicate that improved solution should be applied.

Features from the second column are common words that appeared frequently in comments not containing SATD.

E. WHICH WORDS ARE THE MOST SIMILAR IN SOURCE CODE COMMENTS?

Our enhancement method is based on the *word2vec* model and its ability to retrieve words with the most similar meaning since similar words are embedded closely together in semantic space. We constructed word embedding model using source code comments from 360 open source projects on GitHub. With this model, we can retrieve similarity of words, using the cosine similarity distance (1), since they are embedded in the same vector space. To provide some insights into the constructed model, we presented some common words from the software engineering domain, extracted from source code comments, with their most similar words, in Table 11.

When analyzing the similarity of words in a learned *word2vec* model, we found out that the model indeed found similar words with regard to the software engineering and software development domains. For example, the word *private* is used in java programming language as an access modifier. Other access modifiers are *public* and *protected* which our model also finds as similar words. Numeric types in Java like *long*, *short*, *double*, *int* are also identified as similar words. Another such example could be the word *workaround* with its similar words being *fix* and *hack*, and the word *ineffici* with the most similar words *better* and *ugli*.

[13] Windows 10 based computer with 3.6 GHz (i7) and 32GB RAM
[14] https://www.sonarqube.org/
[15] https://github.com/tsantalis/JDeodorant
[16] https://github.com/processing/processing
[17] https://github.com/skylot/jadx
[18] https://github.com/wequick/Small

TABLE 11. Some word enhancements (5 most similar resulting words for a given input word) as provided by our constructed word2vec model from source code comments.

assert	check verifi expect valid test
bug	issu problem situat test todo
buggi	bug crash bad sometim broken
code	return param string object prog
hack	todo better workaround fix realli
hashtabl	linkedhashmap hashset enummap treemap hashmap
inconsist	unexpect corrupt invalid bad problem
improv	better optim help impact fix
issu	problem messag error fail
logo	icon actionbar drawabl thumbnail titl
long	short int doubl integ valu
polygon	shape triangl rectangl circl polylin
private	public static protect final class
ugli	ineffici hack better strang weird
workaround	fix hack avoid detect prevent
xxx	todo but these we comment

All those words are identified in the literature as words used commonly for SATD [19].

Some words may not seem related at first. For example, some would argue that the words *ugli* and *better* are not related. However, these two words are related in the context of a source code comment. For instance, comments "*this method is **ugly***" and "*should find **better** implementation*", both suggest that the implementation of the method should be improved. In this case, you can see how words tend to be semantically closely together regarding context.

VI. DISCUSSION

The obtained results show that the classification performance has been improved over baseline significantly with the help of the proposed enhancement method WE5 in practically every tested situation. The only two exceptions, where improvements did not prove to be statistically significant, were IG + ME and MI + NB. In both, however, the ranks analysis showed the advantage of our proposed method, as well as in all other testing situations. Similarly, the performed comparison of our proposed method with the existing feature enhancement methods showed the clear overall advantage of WE5 over all the compared methods, with significant advantage in most of the testing situations. The most competitive method to our WE5, regarding the obtained test results, turned out to be SE5. However, while WE5 was significantly better in 4 out of 9 cases, in the other 5 testing situations there were no significant differences between the two methods, and thus we can pronounce the WE5 as the dominant method.

Based on these results, we may say that using the proposed feature enhancement method will most likely improve the prediction of SATD in practically any case or settings. In this manner, software developers, software project managers, and other practitioners can benefit from applying the proposed method to identify the SATD. The experiments have shown that it is generally the best option to use the SVM as a classification algorithm, to select initially approx. 30% of features and then use our proposed method WE5 to add the top 5 most similar features to each of the selected ones. The use of the FS approach is not so important as all three FS approaches performed very similarly.

Let us remember once again that the features in our case are the words from comments in the source code. By analyzing the words, selected initially by a FS method, and the sets of most similar words, added by our proposed feature enhancement method, we may derive some insights, which can help software practitioners to evaluate their projects better, thus making better decisions.

First of all, the calculated similar words are in fact semantically very related to the given input words (look at Table 11). By enhancing the features, we have a much better chance to include different words with practically the same meaning. As the source code comments can be very different, depending on the application domain, cultural background of a developer and similar, the enhancement includes such different words, which would be otherwise overlooked, to be used to train the model. In detecting SATD, such examples are: "improve" is similar to "better" and "optimize", "fix" is similar to "workaround" and "hack", "bug" is similar to "issue", "problem" and even "todo", etc. In addition to our feature enhancement method, the trained word embedding model for computing word similarity could also help to improve many tasks related to information retrieval. Word similarity is already used successfully in a variety of tasks related to the natural language processing (NLP) community. In the software engineering domain, there are many such tasks, like duplicate bug identification, code search, bug localization, etc. [35]. For example, word similarity information from our embedded model could be used for query expansion with additional similar words to improve the accuracy of information retrieval based solutions (e.g., code search, bug classification, bug localization, etc.) [10].

Second, the improvement in accuracy of the constructed model for SATD identification would benefit the tools like SATD detector [16]. It is a recently developed tool that is able to detect and manage SATD comments to support software development in an integrated development environment (IDE), like Eclipse. Since it uses a learned classifier, this tool can analyze and classify source code comments in real time. The identified SATD comments are then highlighted to remind the developers and project managers of the existence of SATD and a need for future refactoring. With the utilization of our approach, project managers can have a quick insight into the quality of code and discover potential problems related to technical debt, which can help to improve evaluation of project quality and thus making a better decision, regarding the release date of the software. Usually, project managers are supervising multiple projects, with many different developers. Since our method is trained on comments from many different projects, it is robust to be used across different projects. For a developer, our method can provide a better reminder of the existence of a possibly forgotten SATD. This is especially useful for new-coming developers since their knowledge of the previously developed code is limited, and the vocabulary they used for code commenting elsewhere was different.

Finally, with our presented feature enhancement method, which is based on a vector space model built from a huge amount of unlabeled comments in various open source software projects, we have shown that the use of word embedding can improve the classification performance of SATD detection. Additionally, we took advantage of an abundance of available source code. Although the comments were not labeled, we were able to utilize the information contained within the content and structure of available source code. As we have demonstrated that our approach can quickly analyze hundreds of thousands of comments in source code to identify SATD, we believe that our work highlights an opportunity for researchers to extensively study the code structure marked as SATD to gain more insights.

A. THREATS TO VALIDITY

We are aware of possible threats to the validity of the presented results and implications they bring.

1) CONSTRUCT VALIDITY

Primarily they relate to the diversity, quality, and quantity of the data. The amount of used unlabeled source code comments and their verified source (all are open-source projects on GitHub) can be considered as very adequate. On the other hand, source code with labeled source code comments is much more difficult to obtain. In this manner, we used a set of 10 open source projects that varied in size, application domain, developers and the number of comments. Open source projects are highly transparent in which developers are more likely to admit technical debt in comments [12]. Thus, we believe the threat to be reasonable. In the future, however, we plan to reduce this threat further by analyzing even more labeled source code comments from additional software projects. Another threat of using only comments for the identification of SATD is that code comments may not be updated consistently with source code. Thus, identifying SATD just using comments would sometimes be misleading. However, some previous works show that changes in code and comments are consistent [29]. Finally, this study relies only on source code comments. In this manner, the detection of SATD in projects with limited code comments, or projects where developers do not express themselves in code comments, would not be possible using the presented method (or would at least be very hard to perform).

2) RELIABILITY

The dataset for training and testing that we used heavily relies on manual analysis and classification of the source code comments made in a previous study [19]. In that study, to reduce human errors and personal bias, the dataset was validated by Master students. Besides, our approach depends on the correctness of the underlying tools we utilize. To mitigate this risk, we used tools that are commonly used in machine learning community, such as scikit-learn,[19] which contains multiple methods for text classification and feature selection, used in our approach.

3) INTERNAL VALIDITY

The potential threat to internal validity could be the selection of projects we analyzed, since not all projects on GitHub are software projects. To avoid such projects, a threshold for minimal number of .java files, and total LOC was considered. Imprecision of our automated SATD in detection method could also effect results of the analysis. Still, we believe that the vast amount of data and high result scores on the validation dataset make us confident about our findings.

4) EXTERNAL VALIDITY

All of our findings were based on comments, written in English, derived from open source Java projects. To minimize the threat to external validity, we chose open source projects from different domains. Nevertheless, our results may not be simply generalized to non-Java projects, commercial projects or projects in different languages. Particularly, our results may not generalize to projects with a low number of comments or comments that are not written in English.

VII. CONCLUSION AND FUTURE WORK

The detection of technical debt is an important task in the software development process. Our work was focused on improving the identification of SATD, which is technical debt that is described in source code comments. Current state–of–the art approaches only use supervised learners to automate the classification of SATD comments. Thus, we proposed the use of word embeddings to improve the detection of SATD. We obtained and preprocessed more than a million unlabeled comments to build a *word2vec* model. In this model, words are embedded into semantic space so that semantically more similar words are closer to each other. We applied the semantic similarity measure to find features, most similar to those from the feature set previously selected by a FS method. These additional features were then used to enhance the original feature set. With the enhanced set of features, we were able to improve the detection of SATD.

Then, we applied our enhanced model for SATD identification to different open source projects. The results of an exploration for the introduction of SATD comments in open source projects have shown that, on average, 1.9% of comments in projects contain SATD. Finally, we explored which words are most common to describe SATD comments in open source projects. Additionally, we have shown that our learned word embedding model can find similar words in the software engineering domain.

In the future, we plan to perform more analyses on how to optimize all the parameters for improving the word embedding model, so that the search for most similar features will be even more accurate. Additionally, an improved word embed-

[19] http://scikit-learn.org/

ding model could be used to construct a software specific word similarity database, to help and solve different specific software engineering problems based on software textual artifacts, like code search and bug localization. We also plan to further analyze the source code, identified with our proposed method as containing technical debt, in order to extract information about code structure and some common patterns of such code.

Since multiple text classification methods have been developed and tested in literature, we plan to conduct additional experiments on applying our enhancement method to different classifiers. We also plan to perform additional experiments on different textual datasets from different domains, to validate our approach.

REFERENCES

[1] G. Bavota and B. Russo, ''A large-scale empirical study on self-admitted technical debt,'' in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, May 2016, pp. 315–326.

[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, ''Managing technical debt in software-reliant systems,'' in *Proc. FSE/SDP Workshop Future Softw. Eng. Res. (FoSER)*, 2010, pp. 47–52.

[3] M. Chen, X. Jin, and D. Shen, ''Short text classification improved by learning multi-granularity topics,'' in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, 2011, pp. 1776–1781.

[4] W. Cunningham, ''The WyCash portfolio management system,'' *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[5] C. De Boom, S. Van Canneyt, T. Demeester, and B. Dhoedt, ''Representation learning for very short texts using weighted word embedding aggregation,'' *Pattern Recognit. Lett.*, vol. 80, pp. 150–156, Sep. 2016.

[6] J. Demšar, ''Statistical comparisons of classifiers over multiple data sets,'' *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Jan. 2006.

[7] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, ''Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study,'' *J. Syst. Softw.*, vol. 124, pp. 22–38, Feb. 2017.

[8] J. Flisar and V. Podgorelec, ''Enhanced feature selection using word embeddings for self-admitted technical debt identification,'' in *Proc. 44th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2018, pp. 230–233.

[9] E. Gabrilovich and S. Markovitch, ''Overcoming the brittleness bottleneck using Wikipedia: Enhancing text categorization with encyclopedic knowledge,'' in *Proc. 21st Nat. Conf. Artif. Intell. (AAAI)*, Jul. 2006, pp. 1301–1306.

[10] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, ''Automatic query reformulations for text retrieval in software engineering,'' in *Proc. Int. Conf. Softw. Eng. (ICSE)*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 842–851.

[11] Z. S. Harris, ''Distributional structure,'' *Word*, vol. 10, nos. 2–3, pp. 146–162, 1954.

[12] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, ''Identifying self-admitted technical debt in open source projects using text mining,'' *Empirical Softw. Eng.*, vol. 23, no. 1, pp. 418–451, 2018.

[13] A. Khatua, A. Khatua, and E. Cambria, ''A tale of two epidemics: Contextual Word2Vec for classifying twitter streams during outbreaks,'' *Inf. Process. Manage.*, vol. 56, no. 1, pp. 247–257, Jan. 2019.

[14] H. K. Kim, H. Kim, and S. Cho, ''Bag-of-concepts: Comprehending document representation through clustering words in distributed representation,'' *Neurocomputing*, vol. 266, pp. 336–352, Nov. 2017.

[15] Q. Le and T. Mikolov, ''Distributed representations of sentences and documents,'' in *Proc. 24th Int. Conf. World Wide Web*, vol. 32. New York, NY, USA: ACM Press, 2014, pp. 29–30.

[16] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, ''Satd detector: A text-mining-based self-admitted technical debt detection tool,'' in *Proc. 40th Int. Conf. Softw. Eng., Companion (ICSE)*. New York, NY, USA: ACM, 2018, pp. 9–12.

[17] C. Ma, Q. Zhao, J. Pan, and Y. Yan, ''Short text classification based on distributional representations of words,'' *IEICE Trans. Inf. Syst.*, vol. E99-D, no. 10, pp. 2562–2565, 2016.

[18] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, ''An empirical study on the removal of self-admitted technical debt,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 238–248.

[19] E. D. S. Maldonado, E. Shihab, and N. Tsantalis, ''Using natural language processing to automatically detect self-admitted technical debt,'' *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.

[20] E. D. S. Maldonado and E. Shihab, ''Detecting and quantifying different types of self-admitted technical Debt,'' in *Proc. IEEE 7th Int. Workshop Manag. Tech. Debt (MTD)*, Oct. 2015, pp. 9–15.

[21] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, and Q. Mi, ''On the value of a prioritization scheme for resolving Self-admitted technical debt,'' *J. Syst. Softw.*, vol. 135, pp. 37–54, 2018.

[22] T. Mikolov, K. Chen, S. Gregory Corrado, and J. Dean, ''Efficient estimation of word representations in vector space,'' *CoRR*, vol. abs/1301.3781, pp. 1–12, Jan. 2013.

[23] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, ''Distributed representations of words and phrases and their compositionality,'' in *Proc. 26th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 2. Lake Tahoe, NV, USA: Curran Associates, 2013, pp. 3111–3119.

[24] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, 1997.

[25] D. S. Moore, G. P. McCabe, and B. A. Craig, *Introduction to the Practice of Statistics SPSS Manual*. New York, NY, USA: W. H. Freeman, 2009.

[26] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, ''Curating GitHub for engineered software projects,'' *Empirical Softw. Eng.*, vol. 22, no. 6, pp. 3219–3253, 2017.

[27] X.-H. Phan, L.-M. Nguyen, and S. Horiguchi, ''Learning to classify short and sparse text & Web with hidden topics from large-scale data collections,'' in *Proc. 17th Int. Conf. World Wide Web (WWW)*. New York, NY, USA; ACM Press, 2008, pp. 91–100.

[28] M. F. Porter, ''An algorithm for suffix stripping,'' *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[29] A. Potdar and E. Shihab, ''An exploratory study on self-admitted technical debt,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep./Oct. 2014, pp. 91–100.

[30] R. Řehůřek and P. Sojka, ''Software framework for topic modelling with large corpora,'' in *Proc. LREC Workshop New Challenges NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50. [Online]. Available: http://is.muni.cz/publication/884893/en

[31] P. Resnik, ''Using information content to evaluate semantic similarity in a taxonomy,'' *CoRR*, vol. cmp-lg/9511007, pp. 1–6, Nov. 1995.

[32] G. Salton and C. Buckley, ''Term-weighting approaches in automatic text retrieval,'' *Inf. Process. Manage.*, vol. 24, no. 5, pp. 513–523, Aug. 1988.

[33] F. Sebastiani, ''Machine learning in automated text categorization,'' *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, Mar. 2002.

[34] D. Steidl, B. Hummel, and E. Juergens, ''Quality analysis of source code comments,'' in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 83–92.

[35] Y. Tian, D. Lo, and J. Lawall, ''SEWordSim: Software-specific word similarity database,'' in *Proc. Companion 36th Int. Conf. Softw. Eng. (ICSE Companion)*. New York, NY, USA: ACM, 2014, pp. 568–571.

[36] A. K. Uysal and S. Gunal, ''A novel probabilistic feature selection method for text classification,'' *Knowl.-Based Syst.*, vol. 36, pp. 226–235, Dec. 2012.

[37] A. K. Uysal and Y. L. Murphey, ''Sentiment classification: Feature selection based approaches versus deep learning,'' in *Proc. IEEE Int. Conf. Comput. Inf. Technol. (CIT)*, Aug. 2017, pp. 23–30.

[38] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. Di Penta, and A. Zaidman, ''Continuous delivery practices in a large financial organization,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2017, pp. 519–528.

[39] L. Vinet and A. Zhedanov, ''A 'missing' family of classical orthogonal polynomials,'' *J. Phys. A, Math. Theor.*, vol. 44, pp. 163–222, Nov. 2010.

[40] P. Wang, B. Xu, J. Xu, G. Tian, C. Liu, and H. Hao, ''Semantic expansion using word embedding clustering and convolutional neural network for improving short text classification,'' *Neurocomputing*, vol. 174, pp. 806–814, Jan. 2016.

[41] S. Wehaibi, E. Shihab, and L. Guerrouj, ''Examining the impact of self-admitted technical debt on software quality,'' in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2016, pp. 179–188.

[42] Y. Yang and J. O. Pedersen, "A comparative study on feature selection in text categorization," in *Proc. 14th Int. Conf. Mach. Learn. (ICML)*. Burlington, MA, USA: Morgan Kaufmann Publishers, 1997, pp. 412–420.

[43] J. Yli-Huumo, A. Maglyas, and K. Smolander, "How do software development teams manage technical debt?—An empirical study," *J. Syst. Softw.*, vol. 120, pp. 195–218, Oct. 2016.

[44] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta, "Recommending when design technical debt should be self-admitted," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 216–226.

[45] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? An in-depth perspective," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories (MSR)*. New York, NY, USA: ACM, 2018, pp. 526–536.

[46] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proc. 2nd Workshop Manag. Tech. Debt (MTD)*. New York, NY, USA: ACM, 2011, pp. 17–23.

[47] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proc. 17th Int. Conf. Eval. Assessment Softw. Eng. (EASE)*, 2013, pp. 42–47.

[48] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Softw. Qual. J.*, vol. 22, no. 3, pp. 403–426, 2014.

JERNEJ FLISAR received the B.Sc. and M.Sc. degrees in computer science from the University of Maribor, Slovenia, in 2010 and 2012, respectively, where he is currently pursuing the Ph.D. degree in computer science. From 2009 to 2012, he was a Java EE Information System Developer. Since 2012, he has been a Researcher and also a Teaching Assistant with the University of Maribor, Slovenia.

His research interests include intelligent systems, text mining, semantic web, and software engineering. He is the author of two journal papers, one book chapter, and eight conference papers. He has been involved in several national research projects, both scientific and industrial Research and Development projects.

VILI PODGORELEC (M'18) received the Ph.D. degree from the University of Maribor, Slovenia, in 2001, where he is currently a Professor of computer science. He was a Visiting Professor and a Researcher at several universities around the world, including the University of Osaka, Japan; Federal University of Sao Paulo, Brazil; University of Nantes, France; University of La Laguna, Spain; University of Madeira, Portugal; University of Applied Sciences Seinäjoki, Finland; University of Applied Sciences Valencia, Spain. He has been involved in AI and intelligent systems for 20 years, where he gained professional experience in implementation of many scientific and industrial Research and Development projects related to analysis, design, implementation, integration, and evaluation of intelligent information systems. He has authored more than 50 peer-reviewed scientific journal papers, more than 100 conference papers, three books, and several book chapters on machine learning, computational intelligence, data science, medical informatics, and software engineering. He has the leading role in the field of research and applications of transparent data-driven decision making (especially in medicine), top expertise in AI and machine learning methods and algorithms, in-depth knowledge of systems integration technologies and methods applied to intelligent data analysis, classification and prediction of human-centered data, as well as large experience in designing and implementing information retrieval, natural language processing and text mining solutions for academia, industrial partners and international companies using the state-of-the-art approaches, methods, and tools. He received several international awards and grants for his research activities.