

Self-admitted technical debt classification using natural language processing word embeddings

Ahmed F. Sabbah¹, Abualsoud A. Hanani²

¹Software Engineering Department, Faculty of Engineering and Technology, Birzeit University, Birzeit, Palestine

²Electrical and Computer Engineering Department, Faculty of Engineering and Technology, Birzeit University, Birzeit, Palestine

Article Info

Article history:

Received Sep 3, 2021

Revised Sep 30, 2022

Accepted Oct 24, 2022

Keywords:

Bidirectional encoder representations from transformers
Convolutional neural network
FastText
Self-admitted technical debt
Software engineering
Technical debt Word2Vec

ABSTRACT

Recent studies show that it is possible to detect technical debt automatically from source code comments intentionally created by developers, a phenomenon known as self-admitted technical debt. This study proposes a system by which a comment or commit is classified as one of five debt types, namely, requirement, design, defect, test, and documentation. In addition to the traditional term frequency-inverse document frequency (TF-IDF), several word embeddings methods produced by different pre-trained language models were used for feature extraction, such as Word2Vec, GloVe, bidirectional encoder representations from transformers (BERT), and FastText. The generated features were used to train a set of classifiers including naive Bayes (NB), random forest (RF), support vector machines (SVM), and two configurations of convolutional neural network (CNN). Two datasets were used to train and test the proposed systems. Our collected dataset (A-dataset) includes a total of 1,513 comments and commits manually labeled. Additionally, a dataset, consisting of 4,071 labeled comments, used in previous studies (M-dataset) was also used in this study. The RF classifier achieved an accuracy of 0.822 with A-dataset and 0.820 with the M-dataset. CNN with A-dataset achieved an accuracy of 0.838 using BERT features. With M-dataset, the CNN achieves an accuracy of 0.809 and 0.812 with BERT and Word2Vec, respectively.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Abualsoud A. Hanani

Department of Electrical and Computer Engineering, Birzeit University

Birzeit, Palestine

Email: ahanani@birzeit.edu

1. INTRODUCTION

Technical debt (TD) is a metaphor, coined by Cunningham [1]. It reflects the additional cost that is implied by rework caused by a sub-optimal solution instead of using the better approach in the software development life cycle. The concept of TD is derived from financial debt, as the interest results from the late payment. TD has an interest rate, and the cost increases if the developer does not pay the debt early, by refactoring the code at a suitable time, to avoid interest in the future.

Technical debt is extremely correlated with immature software and issues in software development, such as requirement debt, which measures the difference between the requirement specification and the actual software implementation [2]. Design debt refers to the violation of the good design principle, where code debt includes poor maintenance and readability, and it needs to be refactored. The documentation debt expresses the lack of information that describes the code, and the testing debt describes the shortage of accepted testing [1].

Some previous studies have shown that technical debt is spreading widely in software, which is inevitable and may have an effect on software quality [3].

The developers' accumulation of technical debt may be deliberately or inadvertently; often it is inadvertently [4]. Inadvertently, TD occurs when the developers afford the debt without intentionality. For example, when the developer writes low-quality code because of insufficiency of experience. Furthermore, deliberately TD occurs with the intention of developers, in a particular situation when the project manager decides to release the software early. When the developers admit these issues and these are documented, often by comments in the source code files, technical debt takes the name self-admitted technical debt (SATD). This term was first coined by Potdar and Shihab [5]. It is a technical debt that is created by the developers deliberately, through comments or commits messages, with the knowledge that the implementation is not an optimal solution for this part of the code [6].

Recently, and following the introduction of the self-admitted technical debt term [6], research has focused more on this direction. The majority of the directions, as introduced in [7], can be divided into three categories. The first direction is detection: that focuses on identifying or detecting the SATD in source code comments. The second is comprehension: those studies focus on the relation of SATD with different aspects of the software process; and the third is repayment, which includes studies with the aim of investigating tools and techniques to remove the TD by "fully repaying" or mitigating "partially repaying". Wehaibi *et al.* [8] assert that the proportion of SATD in the project may have a negative effect on the complexity of software. In addition, they discovered that files of the source code that included self-admitted technical debt had more bug fixing changes, whereas files that did not contain SATD had more defects.

For any purpose involving the treatment of self-admitted technical debt, identification is imperative as the first step. When the type of TD is known, the problem can be handled quickly and with less interest. Three areas of research found in the literature include pattern-based approaches that depend on identifying textual patterns in comments. The machine-learning approach with natural language processing finally, deep learning and natural language processing (NLP) approaches are based on more advanced techniques such as neural networks.

In this paper, we propose an automatic system for classifying the SATD comments and commits written in the source code into five SATD classes, namely, defect, design, documentation, requirement, and test. Some of the NLP techniques are investigated for the SATD classification task, such as term frequency-inverse document frequency (TF-IDF), Word2Vec, GoVe, bidirectional encoder representations from transformers (BERT), and FastText. In conjunction with the NLP techniques, several machine learning techniques are investigated for the SATD classification task, such as naive Bayes (NB), random forest (RF), support-vector machines (SVM), and convolutional neural network (CNN).

Two datasets were used to train and test the proposed systems. Our collected dataset (referred to as A-dataset hereafter) includes a total of 1,513 comments and commits, each manually labeled as one of the five self-admitted technical debt classes. Additionally, an existing dataset, consisting of 4,071 labeled comments, which was used in some previous studies (referred to as the M-dataset hereafter), was also used in this study. By evaluating our proposed systems with an existing dataset, the obtained results can be directly compared with the published results of similar studies.

The main contribution of this paper can be summarized in three folds. First, most of the previous studies applied a binary classification technique to predict whether or not a comment contains SATD. However, the proposed system used multi-classification techniques to identify the SATD categories. Second, to our knowledge, none of the previous studies investigated the effectiveness of the pre-trained and fine-tuned language model transformers such as BERT, Word2Vec, and FastText. Third, to our knowledge, this is the first study that applied the NLP and machine learning (ML) techniques for detecting the SATD classes from the commits as well as the source code comments.

This paper is structured as: section 2 presents related works. Section 3 discusses the research method which includes the proposed system overview, dataset description, feature engineering, and machine learning classifiers. Section 4 presents results and discussion for all experiments. The last section shows the conclusion.

2. RELATED WORK

Potdar and Shihab [5] were the first to analyse the comments in the source code in order to identify the technical debt and introduced the concept of "self-admitted technical debt". Differently from common static

analysis code-based tools, which depend on predefined rules, metrics, and thresholds to expect debt, technical debt refers to the code defective, incomplete, smelly, or temporary, and is written by the developers intentionally (self-admitted), with an obvious recognition that the implementation is not optimal. Those developers admitted that a piece of code is technical debt, and they documented it through comments. The authors explore source code comments in four open-source projects, to study the amount of SATD used in these projects. They also investigated why the developers used this debt in the projects, and how the SATD was removed from the projects. The result was that SATD exists in 2.4%-31% of the files. The majority of SATD were introduced by more experienced developers, and there is no relationship between SATD and time constraints or code complexity. Finally, 26.3%-63.5% of SATD is removed from projects after being presented. Moreover, in this study, Potdar and Shihab introduced 62 patterns to indicate the SATD, through manually reading 101,762 comments to define patterns that indicate SATD [5].

Farias *et al.* [9] presented the contextualized vocabulary model (CVM-TD) to identify the different types of SATD in source code comments. The evaluation of the model showed that comments that were returned by the model were different from the comments that were evaluated to contain the SATD. This results in low-performance detection and the model needs to be enhanced in how word classes are mapped to various SATD types to enhance the model.

Maldonado *et al.* [10] used the NLP maximum entropy classifier (Stanford Classifier) approach to automatically identify SATD from the comments, including design and requirement TD. The authors used 10 open source projects, extracted 62,566 comments, and classified them manually to create a dataset with five types of TD: requirement, design, defect, documentation, and test debt. The experiment used 10-fold cross-project validation, nine open-source projects for training, and one project for validation. The results showed that the NLP improved the identification accuracy compared with previous pattern-based detection. The classifier scored an average F1-measure of 0.620 for design debt, 0.403 for requirement debt, and 0.636 for technical debt without types. Additionally, the study also provided top-10 lists of textual features that the developers used as SATD, which means there is a variety of styles of expression for the SATD.

Bavota and Russo [11] introduced differentiated replication of the work by Potdar and Shihab in [5] on a large scale. The study was run on 195 software projects, with 600K commits and 2 billion comments, to investigate the spread and evolution of SATD and the relationship between these debts and software quality. The main results showed that SATD is distributed in an average of 51 instances for each system. Moreover, even when this debt is fixed, it survives for a long time, on average more than 1,000 commits. Additionally, there are other studies examining the relationship between the SATD and the quality of software. Wehaibi *et al.* have been investigating whether the files that include SATD have a greater chance of including defects in comparison to the files that do not include SATD. Additionally, the changes in the SATD introduce defects in the future. The result of this study showed that the self-admitted technical debt, in addition to the negative impact on the system, is related to defects, and it makes the change more complicated in the future [8].

Farias *et al.* [12] proposed the work that applies, evaluates, and improves previous work of contextualized patterns to detect self-admitted technical debt using source code comment analysis in the studies [9], [13]. The results of empirical studies show that more than 0.50 of the new patterns are critical to technical debt detection. The new vocabulary succeeded in finding items related to code, design, defect, documentation, and requirement debt. The main contribution of this study was to identify self-admitted technical debt depending on the knowledge embedded in the vocabulary [12].

Wattanakriengkrai *et al.* [14] introduced an approach using N-gram IDF, with multi-classification techniques, and built a model that can identify a comment as design debt, requirements debt, or non-SATD. The dataset's Maldonado *et al.* [10] that contains 62K Java source code comments used. A RF machine learning algorithm was applied to classify target comments. The result of the experiment was that N-gram IDF outperformed traditional techniques like bag of words (BoW) and TF-IDF, with an average F1-score value of 0.6474.

Yu *et al.* [15] proposed a Jitterbug framework with two methods for identifying SATD. The dataset proposed by Maldonado *et al.* [10] used. The first type, "easy to find", can be detected automatically without human intervention because the comment has explicitly denoted that it includes the keywords or patterns that relate to the type of SATD such as "Todo", "Fixme". This approach can find 0.20-0.90 of SATD automatically. The precision of identifying SATD close to 0.100, when using a pattern recognition technique. On average, those comments cover 0.53 of the total SATD. The second type: "hard to find" is not easy to classify and needs experts to accurately decide. Only humans can make the final decisions, and it is still hard for algorithms. In

this approach, supervised machine learning is used to present the comment to the experts for identifying SATD that is not identified automatically. After that, the comments identified by the experts are reused by updating the model and through continuous training.

Some studies examine the commit messages to determine the effect and relationship with SATD. Yan *et al.* [16] introduced the level of change in self-admitted technical debt determination. This model determines whether the change introduces SATD by using the versions of the code comments, identifying the SATD at the file level for each version, and analyzing and extracting information from a message in commits that were written by the developers, to predict if the commit is related to SATD. Maipradit *et al.* [17] introduced “on-hold self-admitted technical debt”, to identify the on-hold instance debt, that makes the developer wait for other events or functionality elsewhere to fix this on-hold instance.

Rantala and Mäntylä [18] replicating and extending the work introduced by Yan *et al.* [16], they used 1876 commits messages extracted from five repositories (Camel, Log4J, Hadoop, Gerrit, and Tomcat) that were pre-labeled as SATD, and three techniques of NLP (bag-of-words, latent Dirichl *et al.* location, and word embedding), to predict self-admitted technical debt from commit messages. The main contribution of this study, the bag-of-words technique, is the best performance with a median (AUC 0.7411). Automatic feature selection from the commit message improved the prediction performance for SATD.

3. RESEARCH METHOD

The main goal of this study is to identify the types of TD based on the comments and commits written by the developers. An empirical study has been conducted to measure the relationship between dependent and independent variables in order to satisfy the research goal. Two independent variables are considered pre-trained models and machine learning algorithms, and one dependent variable is the classification accuracy. The proposed approach consists of three main phases: preparing the comments or commits by pre-processing the text using NLP techniques. The aim of the pre-processing phase is to focus more on the words of a sentence that give meaning. Usually, the comments and commits are written in natural language, and they include noises that donnot affect the semantic meaning of the sentence. These sentences need to be handled through pipeline processes to keep the important words and remove the noise. We used some of the NLP techniques [19], [20] to perform this phase, such as tokenization, text cleaning, normalization, and lemmatization. The second phase is features engineering, which converts tokens of text into features. In the final phase, different machine learning techniques are used to classify the SATD into one of five classes. Figure 1 shows the system design diagram.

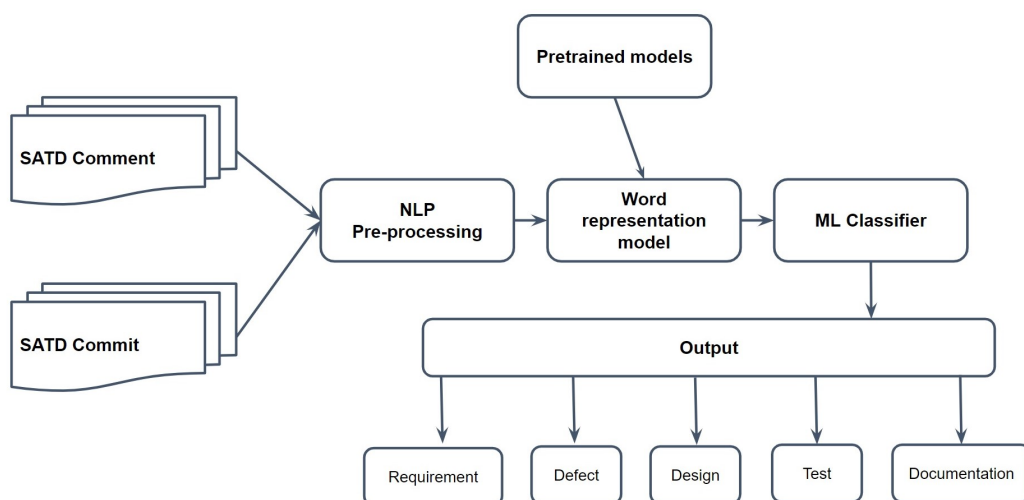


Figure 1. System design block diagram

3.1. Dataset description

To perform our study, we used two datasets. One is publicly available and used in many previous studies (referred to as the M-dataset throughout this paper), and one is manually collected and annotated specifically

for this study (referred to as the A-dataset throughout this paper). We collected two types of expression that the developers write, which can be considered as SATD: source code comments and commit messages. Most of the sentences are comments, and the commits are used to add more variety to the sentences that are used in the A-dataset, with the aim of generalising the results of the proposed approach.

3.1.1. Source code comments

The first dataset used in this study is the dataset (M-dataset) of source comments that is publicly available and introduced by Maldonado *et al.* [10]. This dataset consists of 62k comments extracted from 10 open-source projects (Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and Squirrel SQL). Each comment is manually classified into one of five types of SATD: requirement: 757, design: 2,703, defect: 472, test: 85, documentation: 54, and other comments, which are unclassified. Most of the reviewed studies in this field adopted this dataset, which was introduced by Maldonado *et al.* [10] for SATD identification.

To make the identification of SATD more general with a large variety of sentences and to investigate the accuracy of our methods for classifying SATD, we collected a new dataset from various projects. We extracted 222 source code comments from two open-source Android mobile applications, namely K9 and WordPress (WP). To extract the comments, a SATD detector was used, which automatically detects SATD using text mining-based algorithms described in [21]. In the K9 app, there are 339 comments extracted by this tool, 170 of which are classified as SATD. After removing the duplicated comments, we used 145 comments classified as SATD. In the WordPress app, 94 comments are extracted as SATD and reduced to 77 after removing the duplicate comments. Additionally, 3,102 comments were taken from five open source code projects (Gerrit, Camel, log4j, Tomcat, Hadoop) that were used by Maldonado *et al.* in [22], and they are classified as SATD. We manually labeled these comments into the main five types of SATD, including requirement, design, defect, documentation, and test. Table 1 summarizes the details of the A-dataset comments.

Table 1. Number of new collected comments

Project Name	No. of comments	Removed duplicate comments
Gerrit	272	172
Camel	4,332	1,162
Log4j	136	91
Tomcat	1,318	1,052
Hadoop	1,165	625
K9 App	339	145
WordPress App	94	77
Total	7,317	3,324

3.1.2. Commits messages

For the commit messages, we used the same dataset that was used in the study described in [18]. This dataset consists of 73,625 messages, of which 1,876 are classified as SATD. After removing the duplicated commits and the URL of conduit for change sets between subversion and Git for most of the commits, we get 1758 commits classified as SATD.

3.1.3. Manual annotation

The manual annotation process was performed in two phases. In phase 1, we follow the same classification method described by Yikun *et al.* [23] and Deng [24]. Both are based on a framework proposed by Alves *et al.* [2]. Additionally, Alves *et al.* in [25] conducted systematic mapping in around 100 studies, dated from 2010 to 2014. In our study, we followed the same taxonomy and definitions, at least for the five types of SATD that we used.

Alves *et al.* [2] proposed an ontology for the definitions and indicators of technical debt that were spread across the literature. In other words, the factor that leads to introducing technical debt. Alves *et al.* provide 13 different types of TD with definitions that include: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation, and test debt. In the case of self-admitted technical debt, Maldonado and Shihab in [26] manually analyzed 33,093 comments and classified them. The main finding was that most of the technical debt types that are self-admitted in source code are requirement, design, defect, test, and documentation debt. The other 8 types were not found in this approach,

since the developers do not like to express them in the comments, i.e., infrastructure, people, and process. Additionally, some of the TD may overlap, such as design and architecture.

After preparing a sufficient number of comments and commits, a website [27] is created to simplify the annotation process and, hence, get the maximum number of annotations by a maximum number of expert annotators. The website includes three pages (home, registration, and classification). On the home page, we provide a set of guidelines to help the participants complete the task and a set of definitions related to the technical debt of the target categories of the SATD. Before starting the manual annotation process, each participant needs to provide some details about him/herself, such as email address, the years of experience (1–5 years, 5–10 years, and more than 10 years), and a job description that includes: software engineer, programmer, software architect, QA-testing, project manager, team lead, academic student, and academic teacher. On the classification page, a set of randomly selected comments appear to the participant one at a time. The participant is asked to read carefully the presented comment and then select the appropriate SATD category that best matches the comment. A criterion is applied to the comment selection to guarantee that no comment is repeated for the same participant twice and that the comment is not classified by more than two different participants. The web application also provides the participants with a review summary of the definition of each SATD category. By this, we make sure that each participant is confident about the types of debt that he or she can choose from the list that embeds the five types of self-admitted technical debt. The participant has an option to skip any “not sure” comments. The first author of this study, who has good experience in software development and engineering, annotated a total of 1,513 comments and commits out of the available 5,082. To improve the reliability of the annotations, the random selection of the comments for the other participants is forced to be from the annotated subset by the author. Hence, each selected comment is annotated by the author and by at least one other annotator. The source code of the website is found in [27].

The output of this process is that 1,147 out of 3,324 comments and 366 out of 1,758 commits are annotated. Some of the comments were simply names of functions or methods followed by “TODO” or “FixMe” while others were longer and described more than one type of SATD. These kinds of comments are skipped. Additionally, we skipped the comments that clearly do not belong to any of the considered five types of SATD. Similarly, some commits describe some issues solved by a person; therefore, skipped. Tables 2 and 3 show summaries of the author’s comments and commits annotations.

Table 2. Comments classification

Project	No of comments	Classified	Requirement	Design	Defect	Test	Documentation
Camel	1,162	452	133	182	58	70	9
Gerrit	172	46	19	17	9	1	0
Log4j	91	23	6	11	6	0	0
Tomcat	1,052	334	66	163	77	23	5
Hadoop	625	182	28	81	44	29	0
K9 App	145	75	26	16	26	5	2
WP APP	77	35	14	10	10	1	0
Total	3,324	1,147	292	480	230	129	16

Table 3. Commits classification

Project	No of commits	Classified	Requirement	Design	Defect	Test	Documentation
Camel	739	193	20	34	25	105	9
Gerrit	145	13	4	4	5	0	0
Log4j	74	14	1	8	2	1	2
Tomcat	485	115	10	37	17	45	6
Hadoop	315	31	6	8	5	12	0
Total	1,758	366	41	91	54	163	17

3.1.4. Kappa test

A Kappa statistical test [28] was conducted to measure the reliability of our new dataset. We used Cohen’s kappa coefficient [29] that was used in other studies with the same labelling method [10], [23]. To mitigate the chance of creating a biased annotated dataset, a group session of three experts with various experience in software development participated in the annotation process. One of the participants has a master’s

degree in software engineering and is working as a team leader with 10 years of experience in software development. Two participants are MSc software engineering students at Birzeit University and are working as software developers with 6 and 8 years of experience.

The group session took three hours. In the first hour, we give a short review of the self-admitted technical debt and present the five SATD types with, at least, three examples for each type. In the second hour, we conducted a discussion among the group of participants with questions and answers to make sure that everyone understood the task. Lastly, we view the website and explain the steps for annotation. The three participants interacted with the topic and classified 260 comments and commits selected randomly from the comments and commits that were classified before by the first author. The first expert classified 121 comments and 28 commits. The second expert classified 35 comments and 15 commits, and the third expert classified 49 comments and 12 commits. The majority of the disagreements between the author and the experts concerned the requirements and design types.

We evaluate the level of agreement between the expert's classifications and the author's classifications by calculating Cohen's kappa coefficient [29]. The Cohen's Kappa coefficient is a widely used method to evaluate inter-rater agreement levels for categorical scales, and it calculates the proportion of agreement that is chance-corrected. The result of the coefficient is scaled from -1 to +1, with a negative value indicating worse than chance agreement, zero means exactly chance agreement, and a positive value indicates better than chance agreement [30]. Whenever the value is close to +1, the agreement is stronger. The level of agreement was calculated between two observers (author and experts), and five categories (requirement, design, defect, test, and documentation). We used an online kappa calculator [31]. We achieved a level of agreement measured between the author and the experts +0.82 based on a sample including 17% of all technical debt types, which is considered almost perfect agreement according to Fleiss [28] values larger than +0.75 are characterized as excellent agreement. Table 4 presents the input data for the Kappa test. Each cell in the table is defined by its row and column. The rows specify how each SATD type was classified by the author. The columns specify how the experts classified the subjects. For example, in the second row of the first column, 10 comments were classified by the author as "design" whereas the experts classified them as "requirements". In the second column of the second row, 89 comments were classified by both as design.

Table 4. Input data for Kappa test

		Experts					Total
		Requirement	Design	Defect	Test	Documentation	
Author	Requirement	43	12	2	0	0	57
	Design	10	89	7	0	0	106
	Defect	0	3	45	1	0	49
	Test	1	0	1	36	0	38
	Documentation	0	0	0	0	10	10
	Total	54	104	55	37	10	260

3.2. Features engineering

The second phase of our approach is to extract useful information from the comments and commits and represent them in a form suitable for machine learning. In our approach, we used different NLP methods for transforming the words into numeric features. Some methods depend on syntax that defines the grammatical structures or the set of rules defining a language, such as Bag-of-words and TF-IDF. Other methods focus on the semantics of words, which takes care of the meaning, and how to combine the words together to make a meaningful sentence with consideration of the syntactical rules. Those methods are also called word embedding methods. In our experiments, we used five word-embedding pre-trained models; general Word2Vec, software engineering Word2Vec, FastText, BERT, and GloVe. With these pre-trained models, each word is represented by a high-dimensional vector. Additionally, we used the universal sentence encoder (USE) model, in which the whole sentence is represented by one high-dimensional vector.

3.2.1. Term frequency-inverse document frequency

TF-IDF takes into consideration the weights of words in documents. We generally compute a weight for each word, which signifies the importance of the word in the document and corpus. The most frequent words across all documents don't carry discriminating information, hence, multiplied by low weights. On the other hand, words with low frequency get high weight. In our experiments, we used the TF-IDF representation

method since it is an enhanced and weighted BoW representation. After the comments and commits pass through the pre-processing pipeline, the resulted tokens are converted into numeric features representing the frequencies of the unique words in the given text comment or commit. The number of unique words in the M-dataset is 6,327 out of 44,895 words. For A-dataset, the dictionary size is 3,948 unique words out of 20,929 words. This means that for each comment, the feature vector size is equal to the length of the dictionary.

3.2.2. Word embedding methods

Although the syntactic-based feature representation methods are effective for extracting features from comments, there is a loss of some important information such as semantics, structure, and context around nearby words in each comment. This motivates us to explore state-of-the-art models to capture more useful information than is embedded in the representation of words. In our experiments, we used the following pre-trained models since they are the most common and most successfully used word embedding pre-trained models:

- Word2Vec is the deep learning Google model to train word embedding or vector representation of words. In our approach, we used three models belonging to the Word2Vec family. The first one is the Word2Vec model [32] trained on part of the Google News dataset (about 100 billion words). The model contains 300-dimensional vectors for 3 million words and phrases. The second one is a software engineering-specific model [33]. It is a word2vec model trained over 15GB of textual data from Stack Overflow posts, with over 6 billion words used for training tasks, and the output pre-trained. The output of the model is 200-dimensional vectors for 1,787,145 keywords.
- GloVe is an extended Word2Vec model. The gloVe functions similarly to the Word2Vec. Word2Vec is a “predictive” model that predicts the context given a word and learns its vectors to enhance its predictive ability. GloVe [34] is a count-based model. It learns by building a co-occurrence matrix (words X context) that essentially counts the number of times the word appears in the context, in order to reduce the dimensionality of the co-occurrence matrix. We used the public domain model “glove.840B.300d”, which includes 840B tokens, 2.2M vocabulary, and 300-dimensional vectors, and the model size is 2.03 GB.
- FastText is the state-of-the-art word embedding approach that works at the character level. FastText introduced based on two studies [35], [36] which is essentially an extension of the Word2vec model, but each word is broken down into character n-grams. As a result, a word’s vector is made up of the number of this character’s n-grams. For example, the vector of the word “method” is a sum of the vectors of the n-grams characters: “me”, “met”, “meth”, “metho”, “method”, “eth”, “etho”, “etho”, “hod”, “hod”, “od”. The “crawl-300d-2M.vec” will be used: 2 million word vectors trained on Common Crawl (600B tokens) with a 300-dimensional vector [37].
- BERT stands for bidirectional encoder representations from transformers. BERT [38] is introduced in two variants, such as BERT-BASE and BERT-LARGE. The BERT-BASE has a number of transformer blocks of 12, hidden layer size of 768, attention heads of 12, and total parameters of 110M. The BERT-LARGE has a number of transformer blocks of 24, a hidden layer size of 1024, attention heads of 16, and total parameters of 340M. We followed the recommendation in [39] for tuning BERT parameters. The BERT-BASE model is used. It has a number of transformer blocks (12), a hidden layer size of 768, and attention heads (12). The TensorFlow hub was used to load the BERT pre-trained model.
- USE is a family of pre-trained sentence encoders introduced by Google. We used this method for embedding sentences for classic machine learning, instead of using an average of word embedding calculated for each sentence. The universal sentence encoder model is trained on huge data and supports more than 16 languages. The output of this model is 512-dimensional vectors for each sentence [40].

3.3. Machine learning classifiers

The last phase in our approach is classifying the comments into one of the considered five categories (requirement, design, defect, test, and documentation). Various machine learning techniques are investigated for this task including classical techniques; support vector machines classifier, NB, RF, and CNN which is successfully used for similar tasks. For all experiments that used the classical ML algorithms, the default setting parameters used, as the Scikit-learn library provided [41].

In most of the reviewed studies, the SVM was used as a binary classifier for the SATD identification task [42]. In our study, we use the SVM as a multi-classifier with the features explained in the earlier sections. We used the Scikit-Learn [43] Python library, which includes an SVM implementation. The NB machine

learning algorithm is one of the most famous and successfully applied supervised machine learning classifiers in NLP applications [6], [42].

A random forest is a meta estimator that uses averaging to increase predictive precision and control over-fitting by fitting a range of decision tree classifiers on different sub-samples of the dataset. We use the RF with default parameters as provided by Scikit-Learn [44]. The convolution neural network approach is used by more complex and modern forms of artificial neural network (ANN). We used the CNN model for the multi-classification task, where the CNN model takes an input comment and predicts the type of SATD (requirement, design, defect, test, documentation). The architecture of our CNN includes an input layer, a convolutional layer, a pooling layer, a fully connected layer, and finally an output layer. The implementation of the CNN model and the hyperparameters are studied and described in detail in the next sections. For all CNN experiments, we used the Keras TensorFlow library, which is an open source neural-network library written in Python [45]. The embedding dimension parameter was fixed according to the pre-trained model used. In our experiments, most of the pre-trained models produce 300 dimensions, except the software engineering and the universal sentence encoder pre-trained models, which are 200 and 512 dimensions, respectively. The number of target classes was set to five, which is equal to the number of SATD types considered in this study. We used two architectural neural networks for CNN; simple CNN (single hidden layer), and complex CNN (multiple hidden layers). The difference between the two networks is the number of layers. In the single-layer CNN, we used one layer for both convolutional and pooling layers, whereas, for the complex CNN, three layers are used. The reset of parameters was fixed as: Number of filters: 128, number of classes: 5, stride: 1, filter size for layers in order: [2,3,4], dropout: 0.2, batch size: 32, number of max epochs: 20. To avoid the over-fitting problem in the training and to get the optimal number of epochs. The model stops training when a monitored metric has stopped improving. We used callbacks early stopping parameter in the fit model, the callback will stop the training when there is no improvement in the validation loss for five consecutive epochs. The activation functions: “ReLU” and “softmax”, optimizer: “adam”. Since we are not training depending on our own embedding, and the pre-trained model used, the trainable parameter is set to false and our own embedding matrix is passed in the weights parameter. The following Figure 2 summarizes the three main processes of system design.

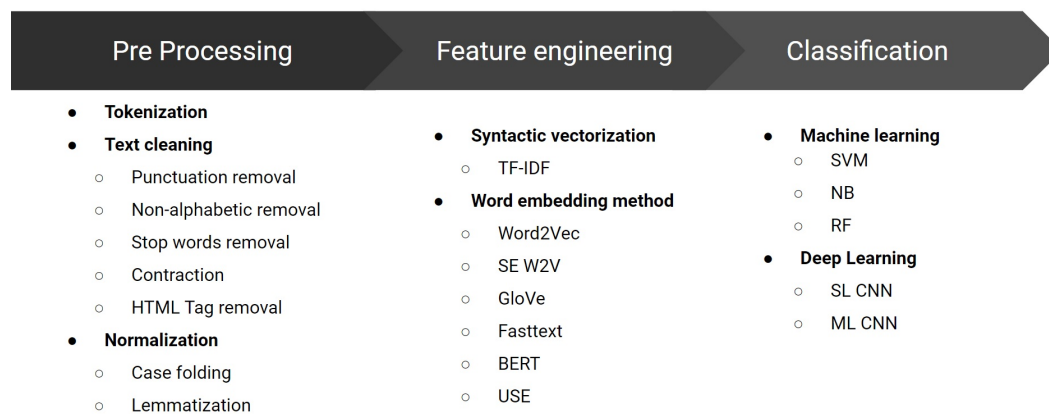


Figure 2. The three main processes in system design

4. RESULT AND DISCUSSION

This section presents all the conducted experiments and their results, with a discussion. In all of the presented experiments, the accuracy and weighted average of precision, recall, and F1-score metrics are used for the system performance. Each experiment is repeated 10 times, and the average performance is reported.

4.1. Experimental setup

To conduct our experiments, we used Google Colab, which is a cloud service that supports GPU processors. Colab allows to write and execute Python in the browser. Table 5 shows the detailed specifications

of the processing capability that we used in all of our experiments. In all of our experiments, the two datasets (A-dataset and M-dataset) were split randomly into two subsets; 0.80 for training and 0.20 for testing.

4.2. Experiments on A-dataset

In this experiment set, each of the ML techniques described in section 3, is trained on the training data and evaluated on the testing data of the A-dataset. Five classifiers are used. Three of them (RF, SVM, and NB) are trained and evaluated with TF-IDF and USE features, and two classifiers (single-layer CNN (SLCNN) and multiple-layer CNN (MLCNN)) are trained and evaluated using word embedding vectors extracted using the five pre-trained models and the TF-IDF. The pre-trained models include Word2Vec, SE-W2v, GloVe, FastText, and BERT. The results of these experiments are shown in Table 6.

Table 5. Environmental setup

Type	Specification
CPU model	Intel(R) Xeon(R) CPU @ 2.20 GHz
Cache size	56320 KB
Ram	13.3 GB
Disk	69 GB
GPU	Tesla T4
OS	Ubuntu 18.04.5 LTS

Table 6. Performance of all classifiers using A-dataset

NB	Precision	Recall	F1-score	Accuracy
TF-IDF	0.703	0.683	0.654	0.683
USE	0.746	0.723	0.726	0.723
RF				
TF-IDF	0.836	0.822	0.810	0.822
USE	0.801	0.779	0.771	0.779
SVM				
TF-IDF	0.826	0.812	0.816	0.812
USE	0.803	0.792	0.795	0.792
SL-CNN				
TF-IDF	0.778	0.772	0.771	0.772
W2V	0.817	0.815	0.811	0.815
SE-W2V	0.793	0.785	0.786	0.785
GloVe	0.826	0.822	0.819	0.822
FastText	0.835	0.828	0.827	0.828
BERT	0.841	0.832	0.834	0.832
ML-CNN				
TF-IDF	0.789	0.789	0.792	0.790
W2V	0.809	0.815	0.807	0.815
SE-W2V	0.813	0.822	0.814	0.822
GloVe	0.824	0.815	0.816	0.815
Fasttext	0.825	0.828	0.825	0.828
BERT	0.843	0.838	0.839	0.838

As shown in Table 6, the RF outperforms the NB and SVM classifiers with an accuracy of 0.822, using the TF-IDF features. The TF-IDF vectorization method outperforms the USE features with the first three classifiers (RF, SVM, and NB). The SVM works better than RF in terms of F1-score, whereas, the RF works better in terms of precision and recall. Moreover, the accuracy of the 10 runs of the SVM classifier ranges from 0.792 to 0.812, whereas, RF ranges from 0.802 to 0.822. In terms of F1-score, the RF and SVM systems with the TF-IDF outperformed the baseline binary classifier system described in [6], by 0.105 and 0.113, respectively. When comparing with only requirement and design classes (baseline classes), the RF with the TF-IDF outperforms the baseline by 0.144 in terms of the F1-score average.

For the CNN-based classifiers, the five pre-trained models and TF-IDF are used with A-dataset to train and test SLCNN and MLCNN models. The results show that the CNN system with the Bert pre-trained model outperforms the TF-IDF and the other pre-trained models. By comparing the results of the SLCNN and MLCNN systems, the MLCNN outperforms the SLCNN in two models SE-W2V and BERT. For the W2V and

FastText models, the accuracy and recall are equal, but the F1-score and precision in SLCNN is better than MLCNN. The Golve with SLCNN outperforms the MLCNN.

4.3. Experiments on the M-dataset

In this experiment set, the five described ML classifiers are trained and evaluated using the M-dataset. All the configurations used in these experiments are kept the same as used in the previous experiments conducted on the A-dataset. The main results of these experiments are shown in Table 7. As it is clear from the presented results, the RF classifier outperforms NB and SVM classifiers. The two CNN-based systems take the same parameters as mentioned in the experimental setup section 4.1. Each system is trained on the training data and evaluated on the testing data of the M-dataset. The five pre-trained models (W2V, SEW2V, GloVe, FastText, and BERT), and TF-IDF are all used for feature extraction in the CNN experiments.

Table 7. Performance of all classifiers using M-dataset

	Precision	Recall	F1-score	Accuracy
NB				
TF-IDF	0.670	0.696	0.590	0.696
USE	0.664	0.639	0.646	0.639
RF				
TF-IDF	0.826	0.820	0.801	0.820
USE	0.838	0.807	0.778	0.807
SVM				
TF-IDF	0.774	0.783	0.775	0.783
USE	0.737	0.753	0.728	0.753
SL-CNN				
TF-IDF	0.757	0.761	0.739	0.761
W2V	0.807	0.812	0.798	0.812
SE-W2V	0.783	0.791	0.783	0.791
GloVe	0.802	0.802	0.785	0.802
FastText	0.804	0.806	0.788	0.806
BERT	0.796	0.804	0.792	0.804
ML-CNN				
TF-IDF	0.721	0.748	0.707	0.748
W2V	0.779	0.793	0.776	0.793
SE-W2V	0.797	0.805	0.794	0.805
GloVe	0.792	0.799	0.784	0.799
FastText	0.783	0.791	0.783	0.791
BERT	0.804	0.809	0.795	0.809

Similar to the experiments conducted on the A-dataset, the TF-IDF outperforms the USE vectorization method. The RF system with the TF-IDF outperforms the baseline system described in [6] which uses the same dataset (M-dataset) by 0.055 of average F1-score for two types of SATD (requirement and design). For the five types of SATD, our approach achieves an average of F1-score 0.801, which outperforms the baseline by 0.092. The RF with the TF-IDF achieves the best accuracy with the M-dataset.

With the CNN classifiers, the single-layer CNN with the Word2Vec outperforms the TF-IDF and the other pre-trained models. Moreover, the accuracy of the five pre-trained models ranges from 0.791 to 0.812. By comparing our system which classifies SATD into five classes with the result of the binary classification study published in [46] that used CNN model and the M-dataset, our SLCNN system outperforms it in the five pre-trained models and the TF-IDF.

The results of MLCNN show that the BERT model outperforms the TF-IDF and the other pre-trained models. MLCNN with the M-dataset are improved in two models, BERT and SE-W2V. This result is the same result of MLCNN with the A-dataset. The accuracy of the other three models (W2V, Glove, FastText) is better with SLCNN for the two datasets.

These results indicate that the single-layer CNN performs better than the multi-layer CNN with three pre-trained models. This suggests that no need for many layers in the neural network for this task, possibly because the length of comments is considered short. The length of comments ranges from 1-500 words, mostly falling between 1 and 50 words, with an average of 11 words. These results of the experiments agree with the findings of the study [47] which found that for text with about 50 words, one convolution pooling layer is suitable, and for text with around 500 words, two convolution pooling layers can be used. For the BERT model,

the result is better than the MLCNN, because the architecture of BERT for word embedding is different from the other models. The main difference between the SE-W2V model and the other models is the dimension of the vector. The SE-W2V produces 200-dimensional vectors, whereas, the other models produce 300-dimensional vectors.

4.4. Statistical test

In all of the experiments presented in this paper, each experiment is repeated 10 times to estimate the variability of the results and to evaluate how close to each other. In addition, to increase the accuracy of the estimate, assuming no bias or systematic error is present. The same classifier runs with two datasets, and the best performance is recorded for each one. To conduct a suitable statistical test for our approach, we adopt the method produced by [48], which performs the statistical tests for multi-classifiers over multiple datasets, similar to our case.

This method compares the obtained results using Friedman's non-parametric test. It ranks the classifiers for each dataset separately, then the Friedman test compares the average ranks of classifiers over all datasets that are shown in Table 8. If Friedman's test finds statistically significant at $p < 0.05$, then the null hypothesis is rejected, we can proceed with a post-hoc test. The Nemenyi test is used to compare all classifiers to each other.

Table 8. Average ranking for all classifiers and language models

	TFIDF	USE	W2V	SW2V	GloVe	FastText	BERT
NB	0.46	0.33	-	-	-	-	-
RF	4.17	1.92	-	-	-	-	-
SVM	2.19	1.13	-	-	-	-	-
SCNN	1.19	-	3.77	2.38	3.11	3.97	4.11
MCNN	1.26	-	2.38	3.08	2.45	3.01	4.37

The result of the Friedman test for null-hypothesis: the distributions of all samples are equal was rejected with $P=4.69E-09$. Additionally, as the study in [48] recommends that Friedman chi-square is working better with N and k are big enough (as a rule of a thumb, $N > 10$ and $k > 5$) where N is the number of datasets, and K is the number of classifiers. We repeated the test with another non-parametric test that should be preferred over the parametric ones [48]. We conduct Nemenyi to compare the accuracy of algorithms for each other which showed there are statistically significant between the accuracy of algorithms.

5. CONCLUSION

In this study, we presented classical machine learning and convolutional neural network approaches to identify and classify SATD from source code comments and commits. Furthermore, we investigated the effectiveness of NLP feature engineering techniques for the SATD classification. Different NLP techniques were used in this study, including TF-IDF and word embedding vectorization methods to feed in different classifiers.

Two datasets were used in this study; the publicly available dataset used in previous studies (M-dataset), and a manually collected and annotated dataset for this study (A-dataset). A Kappa statistical test was applied to accept the label of each SATD comment to verify its authenticity. We achieved a level of agreement measured between the author and experts of +0.82 based on a sample including 0.17 of all technical debt types, which is considered almost perfect agreement.

The traditional and well-known TF-IDF NLP techniques and the state-of-the-art word embedding techniques: USE, Word2Vec, Glove, FastText, and BERT were used for representing comments into a numerical feature vector. We evaluated the proposed approach by comparing the accuracy of the classifiers using the two datasets. For classical machine learning, three types of classifiers are used (NB, RF, and SVM) with two types of word representation methods (TF-IDF and USE). The classical machine learning techniques worked better with the TF-IDF. This can be improved by comparing the best classifier accuracy (RF and TF-IDF) with (RF and USE). The results for the A-dataset showed that TF-IDF achieved an accuracy of 0.822, while USE achieved 0.771. For the M-dataset, TF-IDF achieved 0.820, and USE 0.807. For convolutional neural networks, we used the CNN classifier with five NLP word embedding methods and the TF-IDF. The CNN with BERT achieved the best accuracy for the A-dataset: 0.838. The Word2Vec is the best according to the M-dataset with

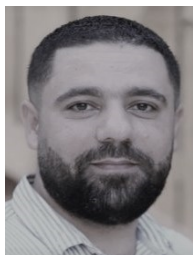
an accuracy of 0.812. In the future, we plan to increase the scale of our approach by adopting more projects that are developed in different programming languages. Additionally, in different domains, for example, mobile applications, commercial software, and medical and healthcare domains, more investigation into the other types of neural networks, deep learning architectures, and pre-trained models, and fine-tuning the parameters of models in order to improve the accuracy of classification SATD.




REFERENCES

- [1] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, 1992, pp. 29–30, doi: 10.1145/157709.157715.
- [2] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola, "Towards an ontology of terms on technical debt," in *Sixth International Workshop on Managing Technical Debt*, Sep. 2014, pp. 1–7, doi: 10.1109/MTD.2014.9.
- [3] E. Lim, N. Taksande, and C. Seaman, "A balancing act: what software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, 2012, doi: 10.1109/MS.2012.130.
- [4] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, Aug. 2012, pp. 91–100, doi: 10.1109/WICSA-ECSA.2012.17.
- [5] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 91–100, doi: 10.1109/ICSME.2014.31.
- [6] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, Feb. 2018, doi: 10.1007/s10664-017-9522-4.
- [7] G. Sierra, E. Shihab, and Y. Kamei, "A survey of self-admitted technical debt," *Journal of Systems and Software*, vol. 152, pp. 70–82, Jun. 2019, doi: 10.1016/j.jss.2019.02.056.
- [8] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2016, vol. 1, pp. 179–188, doi: 10.1109/SANER.2016.72.
- [9] M. A. de F. Farias, M. G. de M. Neto, A. B. da Silva, and R. O. Spinola, "A contextualized vocabulary model for identifying technical debt on code comments," in *IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct. 2015, pp. 25–32, doi: 10.1109/MTD.2015.7332621.
- [10] E. D. S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017, doi: 10.1109/TSE.2017.2654244.
- [11] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories*, May 2016, pp. 315–326, doi: 10.1145/2901739.2901742.
- [12] M. A. de F. Farias, M. G. de M. Neto, M. Kalinowski, and R. O. Spinola, "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary," *Information and Software Technology*, vol. 121, May 2020, doi: 10.1016/j.infsof.2020.106270.
- [13] M. A. de F. Farias, J. A. Santos, M. Kalinowski, M. Mendonça, and R. O. Spínola, "Investigating the identification of technical debt through code comment analysis," in *Enterprise Information Systems - 18th International Conference*, 2017, vol. 291, pp. 284–309, doi: 10.1007/978-3-319-62386-3_14.
- [14] S. Wattanakriengkrai *et al.*, "Automatic classifying self-admitted technical debt using N-gram IDF," in *26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 316–322, doi: 10.1109/APSEC48747.2019.00050.
- [15] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, "Identifying self-admitted technical debts with jitterbug: a two-step approach," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1676–1691, May 2022, doi: 10.1109/TSE.2020.3031401.
- [16] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, "Automating change-level self-admitted technical debt determination," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1211–1229, Dec. 2019, doi: 10.1109/TSE.2018.2831232.
- [17] R. Maipradit, C. Treude, H. Hata, and K. Matsumoto, "Wait for it: identifying 'on-hold' self-admitted technical debt," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3770–3798, 2020, doi: 10.1007/s10664-020-09854-3.
- [18] L. Rantala and M. Mäntylä, "Predicting technical debt from commit contents: reproduction and extension with automated feature selection," *Software Quality Journal*, vol. 28, no. 4, pp. 1551–1579, Dec. 2020, doi: 10.1007/s11219-020-09520-3.
- [19] "Natural language toolkit," *NLTK Project*, 2022. <https://www.nltk.org/> (accessed Jul. 21, 2022).
- [20] "Industrial-strength natural language processing in Python," *spaCy*. <https://spacy.io/> (accessed Jul. 21, 2022).
- [21] 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 *Proceedings - International Conference on Software Engineering*, May 2018, pp. 9–12, doi: 10.1145/3183440.3183478.
- [22] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 238–248, doi: 10.1109/ICSME.2017.8.
- [23] Y. Li, M. Soliman, and P. Avgeriou, "Identification and remediation of self-admitted technical debt in issue trackers," in *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 495–503, doi: 10.1109/SEAA51224.2020.00083.
- [24] A. Deng, "Mining technical debt in commit messages and commit linked issues," Ph.D. dissertation, Faculty of Engineering and Science, University of Groningen, 2020.
- [25] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: a systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, Feb. 2016, doi: 10.1016/j.infsof.2015.10.008.
- [26] E. D. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical Debt," in *IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct. 2015, pp. 9–15, doi: 10.1109/MTD.2015.7332619.




- [27] A. Sabbah, "Self admitted technical debt," *GitHub*, 2021. <https://github.com/asabbah44/SATD> (accessed Jul. 07, 2021).
- [28] J. L. Fleiss, B. Levin, M. C. Paik, and others, "The measurement of interrater agreement," in *Statistical Methods for Rates and Proportions*, vol. 2, no. 212–236, 2003, pp. 598–626.
- [29] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, Apr. 1960, doi: 10.1177/001316446002000104.
- [30] J. L. Fleiss and J. Cohen, "The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability," *Educational and Psychological Measurement*, vol. 33, no. 3, pp. 613–619, Oct. 1973, doi: 10.1177/001316447303300309.
- [31] "Quantify interrater agreement with kappa," *GraphPad*. <https://www.graphpad.com/quickcalcs/kappa1/> (accessed Jul. 21, 2022).
- [32] "Google code archive - long-term storage for google code project hosting," *Google*. <https://code.google.com/archive/p/word2vec/> (accessed Jul. 21, 2022).
- [33] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *Proceedings of the 15th International Conference on Mining Software Repositories*, May 2018, pp. 38–41, doi: 10.1145/3196398.3196448.
- [34] J. Pennington, R. Socher, C. D. Manning, "GloVe: global vectors for word representation." [stanford.edu. https://nlp.stanford.edu/projects/glove/](https://nlp.stanford.edu/projects/glove/) (accessed Jul. 21, 2022).
- [35] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *15th Conference of the European Chapter of the Association for Computational Linguistics*, vol. 2, pp. 427–431, Jul. 2016, doi: 10.18653/v1/e17-2068.
- [36] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, Dec. 2017, doi: 10.1162/tacl.a.00051.
- [37] "English word vectors," *Facebook Open Source*. <https://fasttext.cc/docs/en/english-vectors.html> (accessed Jul. 21, 2022).
- [38] J. Devlin, "Google-research/BERT," *GitHub*. <https://github.com/google-research/bert> (accessed Jul. 07, 2022).
- [39] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language*, Oct. 2019, vol. 1, pp. 4171–4186, doi: 10.48550/arxiv.1810.04805.
- [40] "Universal-sentence-encoder-multilingual," *TensorFlow Hub*. <https://tfhub.dev/google/universal-sentence-encoder-multilingual/3> (accessed Jul. 07, 2022).
- [41] "Scikit-learn: machine learning in python — scikit-learn 1.1.1 documentation." [scikit-learn.org. https://scikit-learn.org/stable/](https://scikit-learn.org/stable/) (accessed Jul. 07, 2022).
- [42] J. Flisar and V. Podgorelec, "Identification of self-admitted technical debt using enhanced feature selection based on word embedding," *IEEE Access*, vol. 7, pp. 106475–106494, 2019, doi: 10.1109/ACCESS.2019.2933318.
- [43] "1.4. support vector machines." <https://scikit-learn.org/stable/modules/svm.html> (accessed Jul. 21, 2022).
- [44] "Sklearn.ensemble.RandomForestClassifier." <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html> (accessed Jul. 21, 2022).
- [45] "TensorFlow." <https://www.tensorflow.org/> (accessed Jul. 21, 2022).
- [46] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 3, pp. 1–45, Jul. 2019, doi: 10.1145/3324916.
- [47] P. Ce and B. Tie, "An analysis method for interpretability of CNN text classification model," *Future Internet*, vol. 12, no. 12, Dec. 2020, doi: 10.3390/fi12120228.
- [48] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

BIOGRAPHIES OF AUTHORS



Ahmed F. Sabbah    received a B.S. in computer science from an-Najah National University, Palestine in 2008, and a master's degree in software engineering from Birzeit University in 2021. From 2008 to 2011, he worked as a software developer in a strategic planning department. From 2011 to 2017, he worked as the head of the section for programming departments in the ministry of interior. From 2017 to 2019, he worked as an IT consultant for the Water Sector Regulation Council. Afterward, he returned to the ministry of interiors with a software engineering position. Ahmed Sabbah joined in 2022 as a Ph.D. student in computer science at Birzeit University. He was interested in some fields of research, such as machine learning, deep learning, NLP, mobile testing, and mobile malware. He can be contacted at email: asabbah44@gmail.com.



Abualsoud A. Hanani    is an associate professor at Birzeit University, Palestine. He obtained Ph.D. Degree in Computer Engineering from the University Birmingham (United Kingdom) in 2012. His researches are in the fields of speech and image processing, signal processing, AI for health and education, and AI for software requirement Engineering. He is affiliated with an IEEE member. In IEEE Access journal, IJMI journal, Computer Speech, and Language Journal, IEEE ICASSP conference, InterSpeech conference, and other scientific publications, he has served as an invited reviewer. Besides, he is also involved in different management committees at Birzeit University. He can be contacted at email: abualsoudh@gmail.com.