An NLP approach for Identification and Detection of Self-admitted Technical Debt: A Review of existing Techniques

Adelaide Anim-Annor
University of Ghana
Department of Computer Science
Legon, Ghana

Fredrick Boafo
Lancaster University Ghana
Department of Computer science
Tantra Hill, Ghana

Solomon Mensah
University of Ghana
Department of Computer Science
Legon, Ghana

ABSTRACT
Programmers tend to leave deficient, non-permanent bypass and demented codes that require rework in software development and such phenomenon is referred to as Self-admitted Technical Debt (SATD). Previous studies have shown that SATD is common in released software artefacts and is mostly found in source code comments. SATD negatively affects software project development and incurs high maintenance overheads. In this study, the authors seek to identify plausible approaches utilized by researchers to identify and detect SATD prone tasks in software artefacts prior to release to the market or clients. Accordingly, a literature review is carried out to perform this investigative study. Two popular approaches were found for identifying and detecting SATD prone tasks from a pool of SATD related research papers, namely manual and text mining approach.

Keywords
Self-admitted Technical Debt; Textual indicators; Source code comment; Lines of Code; Text Mining

1. INTRODUCTION
Self-admitted technical debt refers to the temporary workarounds, temporary fixes, buggy codes, shortcuts left in codes, and codes that require rework which are intentionally left by developers prior to software release [1][2]. Due to expediting pressures from clients, developers normally commit codes that are not quite right to clients. Thus, various patches and shortcuts are used to make the software product ready for release to clients. After a period of usage of the released software products, the clients normally face a series of issues with respect to component malfunctioning, unexpected flaws in components, inactive buttons, issues with logins, etc. These are reported back to the developers for them to rework. It should be noted that these unexpected issues or challenges faced are a result of the developers using shortcuts and temporary fixes to get the software product ready for release to clients. This phenomenon is referred to as self-admitted technical debt [1][2][3]. According to Mensah et al. [1], these deficient and imperfect traits in software development prior to project release will have to be paid in the near future as an uncontrolled maintenance cost.

Cunningham [4] describes the presence of not quite right code or code smell during development as technical debt. Potdar and Shihab [2] coined the term self-admitted technical debt (SATD) from Cunningham's technical debt metaphor because SATD is intentional on the side of the development team. Thus, the developers know that the shortcuts and temporary fixes they are using are not quite right but due to pressure from clients and project managers, there is no option than to find themselves introducing SATD. Aside the expediting pressure from clients and project managers, the level of experience counts in the introduction of SATD. Thus, if a developer has less experience in programming, the person may not be able to fully address all the challenges faced during the software development and hence, might have no choice than to patch faulty codes with temporary fixes. This was found to be true based on an exploratory study by Potdar and Shihab [2] where the authors found out that developers with more experience tend to introduce SATD during coding as compared to those with less experience.

Self-admitted technical debt emanates from software development and maintenance, with a negative effect on software and has lately been a focus for more research studies [5][1][2][6]. SATD is becoming a research focus. Researchers aim to find results for reducing developers' errors and avoid producing less quality software. This misconduct in development is sometimes based on resolutions that prioritize functionality over quality [7].

The challenging question that arises among project managers prior to the release of software products, should managers meet short-term business objectives and release the product as soon as possible or take time and fix the codes before release. From either point of view, a loss can be incurred. Even though not all bugs can be fixed before deployment of the software product, there is the need for the majority of these bugs to be resolved to lessen the issues of clients reporting back problems about the software product usage. This study seeks to examine previous studies that have attempted to classify the identified technical debts that were considered self-admitted into various classes. For example, a study by Maldonado and Shihab [8], introduced a classification scheme for SATD. These five classes are Requirement debt, Design debt, Testing debt, Defect debt and Documentation debt. The study will also consider other classification schemes introduced by other researchers in the Literature review (LR). Several writers have proposed approaches to spontaneously identify SATD comments [11]. Thus, in this study, we perform an in-depth analysis of existing works to examine the various approaches introduced in the literature to identify and detect SATD prone tasks during software development.

2. RELATED WORKS ON SELF-ADMITTED TECHNICAL DEBT
The term Self-admitted technical debt (SATD) was first coined by Potdar and Shihab [2] in their exploratory study using manual inspection to detect SATD. The term SATD came up after Potdar and Shihab [2] had identified that technical debts were intentionally introduced by software developers to speed

up their work or meet deadlines. The authors used manual inspection techniques on four large open-source projects to detect SATD, namely Eclipse, Chromium OS, ArgoUML and Apache HTTP Server. One of the results from their study stated that 62 different commented patterns were indicated as SATD tasks after manually reading through 101,762 code comments. The authors identified that 2.4% - 31.0% of open-source files had SATD. Experienced developers were likely to introduce SATD and it was introduced throughout the development stages. The authors added that release pressure was not a major reason for SATD because less than 15% of SATD was introduced within a month of the latest release.

One of the most recent detection techniques was based on natural language processing (NLP) proposed by Maldonado et al. [9]. This was an automatic approach to identifying the types of SATD. In their work, the authors used a maximum entropy classifier based on NLP to identify the most common types of SATD which were SATD on design and SATD on requirement.

Another approach was introduced by Huang et al. [6] using text mining to automatically detect SATD in source code comments. In this approach, a selection of useful features for classifier training was used in addition to sub-classifiers from different source projects to build a composite classifier to predict accurately. At the end of the study, their approach improved the F1-score by 24.92% on average for each project without the classifier's vote and provided excellent performance while reducing the amount of data needed to train. The text mining approach used eight open-source codes that contained 212,413 comments to detect technical debt. It outperformed the approach introduced by Potdar and Shihab [2] in terms of F1- score. When compared to the manual approach, it improved the F1- score by 58.4% and the natural language processing used by Maldonado et al.[9] was 27.95%.

This section of the study is about the types of SATD. In recent studies by Maldonado and Shihab[8], the authors quantified the different types of SATD as design debt, defect debt, documentation debt, requirement debt and test debt. The authors used five well-commented open-source projects namely Apache Ant, Apache Jmeter, ArgoUML, Columba and JFreeChart. The authors examined more than 166,000 comments which after filtering, resulted in a dataset of 33,093. The authors manually analysed and classified them into the five different types of SATD. The study also stated the most common type of SATD was the design debt which is 42% to 84% in all the classified comments. Maldonado and Shihab [8] explained the types of SATD.

Self-admitted design debt is a comment that shows a problem with the design of the code and the developer sometimes states what needs to be done to improve the design of the code. It can be comments about misplaced codes, long methods, and workarounds. Examples of such comments include “this method is too complex; I hate this so much even before I start writing it”. Self-admitted defect debt states which part of the code is defective. An example of defect debt comments is “bug in the above method, the output stream version of this doesn’t work”. Self-admitted documentation debt is when there is no proper documentation supporting that part of the program. Examples include “this function needs documentation, document the reason for this”. Self-admitted requirement debt expresses incompleteness of method and class. Examples include “no methods yet for getClassname, no method for newInstance using a reserve-classloader”. Self-admitted test debt states the need for improvement of the current test. Examples include “need a lot more test, enable some proper test”. The study of Maldonado and Shihab[8] showed that the percentage of the most common type of SATD was design debt with 42% to 84% across the projects. The second most common type is requirement debt which is 5% to 45%. The rest of the types have low frequency with 10%.

Another study by Bavota and Russo [3] mined over 2 billion source code comments of 159 Java open-source projects and identified 51 SATD instances per project. In their manual categorization of SATD, the authors adopted an open coding process inspired by the Grounded theory principles [10]. The authors identified the types of SATD which were code debt, design debt, documentation debt, defect debt, test debt and requirement debt. The authors highlighted that the most frequent type was code debt with 30% of the open-source projects, followed by defect and requirement debt with 20% each and design debt with 13%. The authors also found out that in most cases 63% of the developers who introduced the debt were the same fixing the debt. The phase in the software development lifecycle which had the highest technical debt was design with 39.34% followed by test with 23.70% and project convention with 15.64% Silva et al. [11].

This section is about the removal of SATD after it has been identified. Recent studies by Maldonado et al. [9] stated that after the analysis of five case projects, 40.5% to 90.6% of identified SATD were removed. The study showed that on average, 54.4% of SATD tasks were self-removed meaning that developers who introduced it were the same people who did the removal and on the median, 61.0% were self-removed. This study further showed that the amount of time SATD remains in a project before removal ranges between 18.2 to 172.8 days on median and 82 to 613.2 days on average. Developers mostly remove SATD when they are fixing bugs or adding new features. Zampetti et al. [12] proposed an in-depth quantitative and qualitative empirical study on the removal of SATD which was built on the previous work of Maldonado et al [9]. The authors' study indicated that 25% to 60% of SATD comments were removed due to full class or method removals. Moreover, 33% to 63% of SATD was removed due to a change in method and 8% of SATD removal was documented in commit messages. Their paper further stated that 55% of SATD comments were removed while improving features. Bavota and Russo[3] showed that 57% of SATD were removed during the change history of software and 63% of SATD were self-removed. Potdar and Shihab [2] showed that 26.25% to 63.45% of SATD were removed either in the next immediate release or over multiple releases. Even though SATD is not an optimal solution, organizations have embraced technical debt into project plans knowing vividly the risk to software quality and maintenance (Krutchen et al. [13]). Properly managed SATD can add value to the software but poorly managed SATD can affect the software [13].

The last section is the repayment and prioritization of SATD. Mensah et al. [1] introduced a six(6) step prioritization scheme that aimed to inspect SATD in classifying them by how urgently they needed to be addressed and estimated the rework effort they require. The authors implemented a text mining algorithm on four open-source projects that were AgroUML, Chromium OS, Eclipse, and Apache HTTP Server and presented an empirical finding showing that 31% to 39% of SATD comments were major tasks and 58% to 69% were minor. The major task was difficult for developers to resolve.

There has been much research into the area of detection, comprehension, repayment and management techniques. Although SATD is not an optimal solution, it must be considered in the preparation of a balance sheet. The aftermath

of delaying development activity would be cost-effective so that it would aid software project managers to have a fair understanding of the financial status of the software organization. SATD should be considered a financial obligation. According to Akbarinasaji and Bener [14], technical debt listed as a liability gives a realistic insight into the liability of the corporation, provides a better overview of the business's finances and aids in getting more accurate financial information for the income statement. It helps in good decision-making whether the organization should improve profits or go into investment because liabilities and their values are visible on the balance sheet. It will also make managers aware of their debt and how it increases or decreases. Relegating technical debt as a liability might cause misinterpretation of the financial ratio. Akbarinasaji and Bener [14], further stated that companies that neglect technical debt would affect the total liability of the organization and deceive the gauge of earning power and financial condition. To append technical debt as a liability, all technical debt associated with the company's product should be extracted by specialists and quantified. Since technical debt is abstract, there is no consensus on how to measure its monetary value, but a balance sheet includes intangible assets such as goodwill. An approach to quantifying the intangible has been developed by accountants. Thus, technical debt can be fixed under intangible assets. This depicts the essence of appending technical debt in the balance sheet.

The review task seeks to identify and detect SATD prone tasks based on existing works [1][3][2][15][16]. The authors first perform an investigative study to an exploratory study by Potdar and Shihab [2] who extracted a pool of 62 textual indicators from four datasets, namely Eclipse, Apache, Chromium and ArgoUML. These textual indicators were considered as a baseline for the identification of SATD prone tasks in studies by Mensah et al. [1], and Bavota and Russo [3]. Thus, after the exploratory study analysis [2], the authors moved on to a study by Maldonado et al. [8] who identified five levels or classes of SATD prone tasks based on the textual indicators exploratory study [2]. This was followed by a study by Mensah et al. [1] who introduced an NLP approach based on a 5-step prioritization scheme for prioritizing SATD prone tasks and estimating the rework effort for such SATD tasks. It should be noted that Shihab and Potdar [2] were the first to introduce the SATD concept in 2014.

3. METHODOLOGY

The literature review (LR) approach has been adopted for the identification and detection of SATD in the reviewed papers. According to Kitchenham et al. [17], SLR is the systematic mode of reporting the outcomes extracted from the literature. SLR method offers a way of classifying, exploring and examining the present studies connected to any questions of interest and research areas. Kitchenham [18], classifies SLR into three main phases, i.e. planning the review, conducting the review and reporting the review. The SLR protocol is the outcome of the planning phase.

3.1 Research Questions (RQ)

RQ1: What are the methods used to identify and detect SATD prone tasks?

Motivation: There is a need to identify and detect SATD prone tasks to assist software engineers in implementing various approaches to detecting SATD prone tasks. The SATD metaphor has been given attention by most researchers, hence assisting with effective tools and methods will provide the software engineering team to ameliorate the SATD prone tasks in software development.

Approach: This RQ is complemented by the various tools and techniques that have been used by researchers to identify and detect SATD. We read 19 SATD and TD papers to identify the methods of detection that have been used over the years. These papers were focused on the detection, comprehension and repayment of SATD. Since this paper is on detection, the focus of the 19 papers was drawn to the methods used. After several analyses of each detection paper, we identified the two main types of detection methods that have been adopted. Some papers used both manual and text mining as a detection technique. The manual approach is the first method used in SATD detection [2]. This technique reads all the source code comments and LOC manually. The text mining was the second novel approach which automated the process of manual inspection [7]. Text mining was to improve the results at a faster rate [1]. Other papers used both approaches in their methodology.

RQ2: Where are SATD prone tasks identified in software artefacts?

Motivation: This RQ seeks to identify plausible software artefacts or projects that researchers have been using to detect SATD prone tasks. Hence, a review of the literature will assist practitioners and researchers in knowing which artefacts to use for identifying SATD prone tasks.

Approach: An extensive LR is conducted to identify software artefacts prone to SATD tasks from literature. We read through 19 SATD papers and all these papers used open-source projects and SATD prone tasks were mainly identified from source code comments or LOCs. The source code comments are remarks the developers write at the end of the code. Most papers that were reviewed used the comments to identify SATD prone tasks.

RQ3: Which tools are used for data (source code comments or LOCs) extraction in SATD?

Motivation: This RQ seeks to identify extraction tools that researchers have been using to extract source code comments or LOCs in SATD. Therefore, a review of the literature will aid practitioners and researchers in knowing which extraction tools to use.

Approach: An extensive SLR is conducted to identify all extraction tools. The SATD papers considered in this work used a variety of extraction tools. These extraction tools are used to sunder out source code comments which are SATD instances from source files or projects. The extraction tools used were srcML, Jdeodorant, NLP, WEKA, Python-based tool, Eclipse plugin and Java-based tool. Most papers considered two of the tools mentioned together to extract source code comments to make their analysis.

RQ4: Which open-source projects are used in the detection and identification of SATD?

Motivation: This RQ strives after the open-source projects that have been used by researchers. Thus, a review of the literature will enlighten practitioners and researchers to know the projects to use.

Approach: An extensive SLR was conducted on all the reviewed papers. The source code comments were identified from the open-source projects. There were 20 open-source projects used in the papers reviewed. Most papers used more than one project.

3.2 Datasets

Four well-commented open-source projects were made available at openhub.net and extracted by Potdar and Shihab [1] for studying SATD. The four projects are ArgoUML,

Chromium OS, Apache HTTP Server and Eclipse Platform project. The description of the open-source projects is presented in Table 1. In each project, the authors used metrics such as the total number of Lines of Codes (LOC), number of commented lines, dates of software release, estimated effort in person-months and contributors or developers for each sampled open-source project.

Table 1. Description of Sampled Projects

Metric	Sampled Open-Source Projects			
	ArgoUML	Chromium	Eclipse	Apache
LOC	122575	107706	659231	192333
Comment	115713	37889	437640	54295
Date	Dec 2011	Nov 2009	Jun,2013	Jul 2013
Effort	3024	47856	8760	6000
Developers	53	1784	221	145
Version	0.34	30	4.3	2.4.6

From the text mining analysis, the authors observed that these projects were developed using several programming languages with the dominant ones being Java, C++, C and XM. With respect to the ArgoUML project, a majority of 39.3% was written in Java, 31.9% in XML, 14.1% in HTML, 11.0% in XSL Transformation, 1.5% in CSS and less than 1% each in Modula-2, JavaScript, shell-script, DOS batch script, MetaFont, Pearl and SQL. In relation to the Chromium project, a majority of 89.0% was written in C, 4.8% in C++, 2.4% in Assembly, 2.2% in Python and the remaining less than 1% in XML, Make, shell script, HTML and Java. For the Eclipse project, a majority of 80.2% was written in Java, 10.5% in XML, 5.5% in HTML, 2.7% in C and the remaining in C++, CSS, MetaFont and JavaScript. Lastly, a majority of 55.5% of Apache HTTP Server was written in XML32.4% in C, 8.2% in Forth, 1.1% in XSL Transformation and the remaining in HTML, Autoconf, shell script, CSS, JavaScript, C++, IDL/PV-WAVE/GDL, Pearl, AWK, CMake and Make.

3.3 Extraction of Self-Admitted Technical Debt Textual Indicators

A study by Potdar and Shihab [2] performed a manual exploratory study by reading through source code comments from four datasets, namely Eclipse, Apache, Chromium OS and ArgoUML to identify SATD textual indicators. This was to assist researchers in building a text mining algorithm or NLP approach so that the process of SATD extraction can be automated to increase the possibility of its use in practice. The source code comments from each dataset were examined by Potdar and Shihab [2], and the key indicators contributing to the various classes of SATD were extracted. This study presents a sample of the source code comments prone to SATD below.

* TODO: REMOVE ME BEFORE PRODUCTION (????) *

* Ready to revalidate, pretend we were never here *

* Not removed and not expired yet, we're done iterating *

* TODO: Add directive for tuning the update interval *

* Something is wrong here but the result is what we wanted

* Strictly speaking, this is a design error *

* DESIGN ERROR: a mix of repositories *

* Alternative: just load unresolved locks *

* Note: this shouldn't happen, but just be sure... *

* Hmm. this doesn't feel like the right place or thing to do *

* Open the thing lazily *

* If we've never heard of this object bail out *

* Give up because the parent is still not materialized *

* TODO: this isn't quite right but is ok for now *

* Yuck, this is awkward to use *

* Macro is ugly but makes the tests pretty *

* TODO: analyze why we have such a bad bail out here! *

* FINISHME: This is wrong. The constant value field should…*

* Detect dead code, nuke it, and calculate again for new change *

* FINISHME: This hack makes writing to … *

* Skip the stupid Microsoft UTF-8 Byte order marks *

Ignore any errors and continue with index0 if there is a problem

This list of textual indicators formed a dictionary of words which was used in a proposed text-mining approach by Mensah et al. and realized that most of the SATD comments with their respective indicators were similar across projects [1]. The authors observed that most of these indicators were common for most of the five classes and based on previous work [8]. Thus, present the five levels or classes of categorizing the SATD prone tasks as follows:

Class of SATD	**Class**
Requirement debt	1
Design debt	2
Test debt	3
Defect debt	4
Documentation debt	5

The explanation with examples of the classes of SATD is elaborated in the study by Maldonado and Shihab [8].

Text classification is the process of labelling a set of text documents into several classes or categories from a predefined set [19]. The features are the developers and source code comments which correspond to the terms or words in a text document. Dilara et al. [20] define these features as bag-of-words. With a pool of dictionary of indicators from these bag-of-words, previous studies [16][1][3] were able to perform the text classification analysis on SATD prone tasks.

3.4 Text Mining Technique

Mensah et al. [1] proposed a text-mining technique for mining source code comments of open-source projects and extracting key terms on SATD. This technique plays a significant role in transforming source code comments into numeric counts based on the assignment of term weights for easy modelling of the SATD source code comments. The text mining technique for commented source code projects is divided into four phases which are the pre-processing phase of the project datasets, extraction of code comments containing technical debts specifically SATD from the overall source code comment,

Computation of term weights for SATD, and computation of rework effort for each class of SATD.

Data Pre-processing Phase: Pre-processing is an important phase in text mining and text classification. For efficient regular expression matching, the authors per-processed the extracted open-source code comments based on the following as elaborated by Kotsiantis et al. [21] and Torunoglu et al. [20].

Data Cleaning: The authors used the proposed text mining approach to remove punctuation marks in the form of ~! @, - #$%^*] [|\ from the corpus of source code comments. Again, the authors filtered out noise in the form of blank lines and white spaces within strings from each project dataset. Typical examples of white spaces within strings were "FIX ME" and "TO DO" which it was realized that most developers used in various comments compared to "FIXME" and "TODO". This was addressed by the proposed text mining technique (using the grepl function). This filter approach enabled the pattern recognition process to obtain desired searched patterns during the text mining.

Data Reduction: The authors applied a stratified sampling method to divide the total instances in each corpus into k strata (partitions) and eliminated blank lines and comments that were not prone to SATD.

Stopword Filtering: Here, common words occurring frequently such as and, this, the, or, of, am, it, on, at were removed since they less contributed to the text mining and classification process. These words and other related words that frequently occurred in the source code comments were searched and removed from all the sampled projects based on a similar approach by Fazli et al. [22].

Term Weighting: Based on the proposed technique, the authors assigned term weights to the various SATD source code comments in all cases of the sampled OS projects. This enabled the authors to know the frequency at which the SATD indicators occurred in the source code comments. The assignment of term weight values was done based on term frequency-inverse document frequency (tf-idf) [23][24], which is a well-known ranking function in text mining and information retrieval [25]. This function assigns numeric weights to significant and frequently occurring terms in documents [25]. Tf-idf weight is composed of the product of the term frequency (tf) and the inverse document frequency (idf). The authors defined these two terms in (1) and (2) with respect to each project dataset.

$$tf(t,d) = \frac{f_{t,d}}{m_d} \quad \forall d \in D \qquad (1)$$

$$tdf(t,D) = \log_e \frac{D}{N_t} \qquad (2)$$

$$tftdf(t,d,D) = tf(t,d) \times idf(t,D) \qquad (3)$$

where

ft,d = frequency of term (t) in an SATD comment (d)

md = number of terms in a given SATD comment

D = total number of SATD comments

Nt = number of SATD comments with a given term (t)

A simple example to illustrate the computation of weights for the tf-idf is given as follows. Consider a sample project with a finite number of commented LOC considered in the authors' case as documents. Assuming a document contains 6 terms free from stopping words whereby the term error appears 2 times. The tf for error is then (2/6) = 0.333. Assuming there is 10,000 documents and the term error appears 100 times, then the idf(error) is calculated as log (10,000/100) = 4.605. Thus, the tfidf weight is the product of these two quantities: 0.333 × 4.605 = 1.533.

4. RESULTS AND DISCUSSION

This section discusses the findings obtained from this study based on the postulated four research questions (RQs).

RQ1: What are the methods used to identify and detect SATD prone tasks?

In the 19 reviewed SATD papers there was a realization that there were two popular methods used for SATD detection manual inspection and text mining. The manual inspection introduced by [2] manually inspected comments using grounded theory principles and the text mining approach introduced by [1], used this approach to extract SATD prone tasks from the studied projects. In the authors' findings, the most used method was text mining which was 26.1% and 5.3% used manual inspection whereas 10.5% used both methods. These two methods fall under two categories which are pattern-based and machine learning-based. The result concluded that 15.8% were pattern-based and 26.1% were machine learning-based. Table 2 shows the various approaches and its frequency and percentage.

RQ 2: Where are SATD prone tasks identified from in software artefacts?

From the 19 reviewed SATD papers the authors found that SATD prone tasks are identified from source code comments and Lines of codes (LOCs). The total frequency for LOCs and source code comments is 9 where LOCs had 0, source code comments had 6, source code comments and LOCs had 3 which implies some papers identified SATD prone tasks from both LOCs and source code comments. Thus, in the reviewed 19 SATD papers no SATD prone task was identified from LOCs. In the reviewed papers 66.67% identified SATD prone instances from source code comments and 33.33% were from both LOCs and source code comments as indicated in Figure 1.

Table 2. Approaches for SATD identification and detection

APPROACH	FREQUENCY =19	PERCENTAGE
Manual	1	5.263157895
Text mining	5	26.31578947
Manual & text mining	2	10.52631579
Pattern-based	3	15.78947368
Machine learning based	5	26.31578947

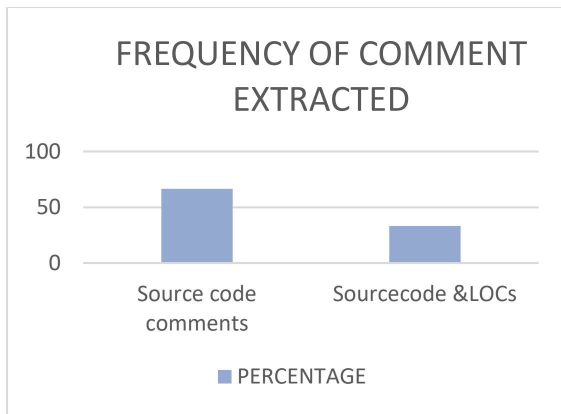

Fig 1: Frequency of comment extracted from source code and LOCs in percentage

RQ3: Which tools are used for data (source code comments or LOCs) extraction in SATD?

There was a realization from the 19 reviewed papers that the extraction tools mostly used were srcML, Jdeodorant, NLP, WEKA, Python-based tool, Eclipse plugin and Java-based tool. The frequently used tool was srcML with 22.22% and the others all had 11.11% as indicated in Figure 2.

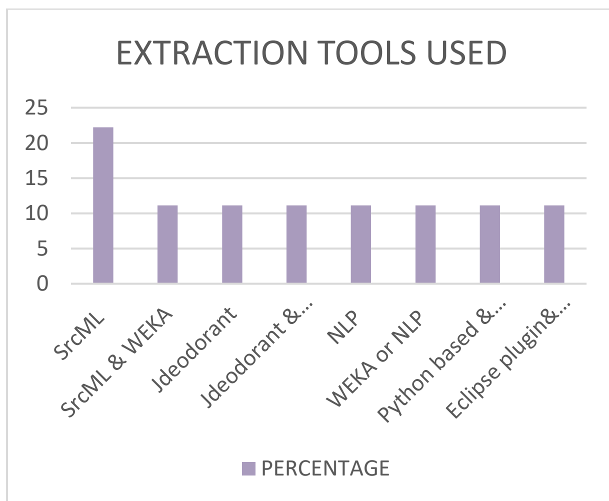

Fig 2: Percentages of extraction tools used

RQ4: Which open-source projects are used in the detection and identification of SATD?

The open-source projects used in the reviewed papers were Eclipse, Apache HTTP Server, Chromium OS, ArgoUML, Cassandra, Tomcat, EMF, Camel, Log4j, Hadoop, Jmeter, Ant, Gerrit, Jruby, Spark, Hibernate, JEdit, Columba, JFreeChart and Squirrel. The percentage of each open-source project used in SATD identification and detection is shown in Figure 3.

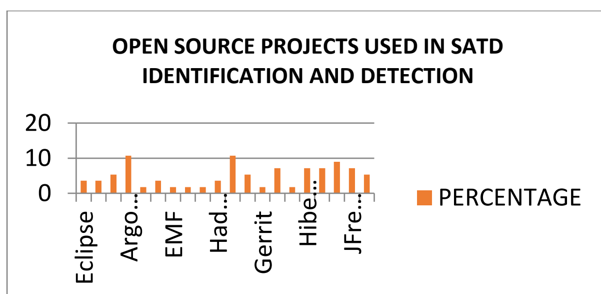

Fig 3: Open-source projects used in SATD identification and detection

5. CONCLUSION AND FUTURE REMARKS

Self-admitted Technical Debt (SATD) refers to the tendency of programmers to leave inadequate, temporary bypass, and insane codes that need to be rewritten in software development. Previous research has demonstrated that SATD is typically found in source code comments and is present in many released software artefacts. Software project development is negatively impacted by SATD, which also has high maintenance costs. In this study, the authors looked for conceivable methods that academics have used to identify and detect SATD-prone jobs in software artefacts before releasing them to customers or the market. As a result, a literature review is done in order to conduct this research. From a collection of SATD-related research papers, two popular methods for identifying and detecting SATD-prone tasks were discovered, namely natural language processing/ text mining and manual classification.

Future works could consider machine learning, deep learning or state-of-the-art techniques and approaches for the identification and detection of self-admitted technical debt.

6. ACKNOWLEDGMENTS

Our thanks to the experts who have contributed towards the development of this study.

7. REFERENCES

[1] 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, doi: 10.1016/j.jss.2017.09.026.

[2] A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” Proc. - 30th Int. Conf. Softw. Maint. Evol. ICSME 2014, pp. 91–100, 2014, doi: 10.1109/ICSME.2014.31.

[3] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” Proc. - 13th Work. Conf. Min. Softw. Repos. MSR 2016, pp. 315–326, 2016, doi: 10.1145/2901739.2901742.

[4] W. Cunningham, “The WyCash Portfolio Management System,” 1992.

[5] S. Wehaibi, E. Shihab, and L. Guerrouj, “Examining the impact of self-admitted technical debt on software quality,” 2016 IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2016, vol. 1, pp. 179–188, 2016, doi: 10.1109/SANER.2016.72.

[6] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” Empir. Softw. Eng., vol. 23, no. 1, pp. 418–451, 2018, doi: 10.1007/s10664-017-9522-4.

[7] C. Fern and J. Garbajosa, “A Framework to Aid in Decision Making for Technical Debt Management,” pp. 69–76, 2015.

[8] E. D. S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical Debt,” 2015 IEEE 7th Int. Work. Manag. Tech. Debt, MTD 2015 - Proc., pp. 9–15, 2015, doi: 10.1109/MTD.2015.7332619.

[9] S. Maldonado, E. Shihab, and N. Tsantalis, “Using nutural language to Automatically Detect Self-Admitted Technical Debt,” vol. 43, no. 11, pp. 1044–1062, 2017.

[10] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," Qual. Sociol., vol. 13, no. 1, pp. 3–21, 1990, doi: 10.1007/BF00988593.

[11] M. C. O. Silva, M. T. Valente, and R. Terra, "Does technical debt lead to the rejection of pull requests?," SBSI 2016 - 12th Brazilian Symp. Inf. Syst. Inf. Syst. Cloud Comput. Era, Proc., no. ii, pp. 248–254, 2016, doi: 10.5753/sbsi.2016.5969.

[12] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal?: An in-depth perspective," Proc. - Int. Conf. Softw. Eng., pp. 526–536, 2018, doi: 10.1145/3196398.3196423.

[13] A. Martini and J. Bosch, "Towards Prioritizing Architecture Technical Debt : Information Needs of Architects and Product Owners," pp. 422–429, 2015, doi: 10.1109/SEAA.2015.78.

[14] S. Akbarinasaji and A. Bener, "Adjusting the Balance Sheet by Appending Technical Debt," Proc. - 2016 IEEE 8th Int. Work. Manag. Tech. Debt, MTD 2016, pp. 36–39, 2016, doi: 10.1109/MTD.2016.14.

[15] G. Sierra, A. Tahmid, E. Shihab, and N. Tsantalis, "Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences?," SANER 2019 - Proc. 2019 IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering, pp. 534–543, 2019, doi: 10.1109/SANER.2019.8667999.

[16] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, "Automating Change-level Self-admitted Technical Debt Determination," vol. 5589, no. c, pp. 1–18, 2018, doi: 10.1109/TSE.2018.2831232.

[17] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - A systematic literature review," Inf. Softw. Technol., vol. 51, no. 1, pp. 7–15, 2009, doi: 10.1016/j.infsof.2008.09.009.

[18] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in SE," pp. 1–44, 2007, doi: 10.1145/1134285.1134500.

[19] F. Sebastiani, "Machine Learning in Automated Text Categorization," ACM Comput. Surv., vol. 34, no. 1, pp. 1–47, 2002, doi: 10.1145/505282.505283.

[20] D. Toruno, E. Çak, M. C. Ganiz, S. Akyoku, and M. Z. Gürbüz, "Analysis of Preprocessing Methods on Classification of Turkish Texts," pp. 112–117, 2011.

[21] S. B. Kotsiantis, D. Kanellopoulos, and P. E. Pintelas, "Data Preprocessing for Supervised Leaning," vol. 1, no. 2, pp. 111–117, 2006.

[22] X. Liu, "Full-Text Citation Analysis : A New Method to Enhance," J. Am. Soc. Inf. Sci. Technol., vol. 64, no. July, pp. 1852–1863, 2013, doi: 10.1002/asi.

[23] G. Salton, "1988_Salton, G. and Buckley, C., 1988. Term-weighting approaches in automatic text retrieval._7896.pdf." 1988.

[24] H. Sch, "Introduction to Information Retrieval IIR 8 : Evaluation & Result Summaries Recap Unranked evaluation Ranked evaluation," Evaluation, 2008.

[25] M. Baena-garc, "TF-SIDF : Term Frequency , Sketched Inverse Document Frequency," pp. 1044–1049, 2011.