The Journal of Systems and Software 135 (2018) 37–54

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On the value of a prioritization scheme for resolving Self-admitted technical debt

Solomon Mensah[a,*], Jacky Keung[a], Jeffery Svajlenko[b], Kwabena Ebo Bennin[a], Qing Mi[a]

[a] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[b] Department of Computer Science, University of Saskatchewan, Saskatoon, Canada

ARTICLE INFO

Article history:
Received 12 February 2017
Revised 16 August 2017
Accepted 25 September 2017
Available online 28 September 2017

Keywords:
Self-admitted technical debt
Prioritization scheme
Textual indicators
Source code comment
Open source projects

ABSTRACT

Programmers tend to leave incomplete, temporary workarounds and buggy codes that require rework in software development and such pitfall is referred to as Self-admitted Technical Debt (SATD). Previous studies have shown that SATD negatively affects software project and incurs high maintenance overheads. In this study, we introduce a prioritization scheme comprising mainly of identification, examination and rework effort estimation of prioritized tasks in order to make a final decision prior to software release. Using the proposed prioritization scheme, we perform an exploratory analysis on four open source projects to investigate how SATD can be minimized. Four prominent causes of SATD are identified, namely code smells (23.2%), complicated and complex tasks (22.0%), inadequate code testing (21.2%) and unexpected code performance (17.4%). Results show that, among all the types of SATD, design debts on average are highly prone to software bugs across the four projects analysed. Our findings show that a rework effort of approximately 10 to 25 commented LOC per SATD source file is needed to address the highly prioritized SATD (*vital few*) tasks. The proposed prioritization scheme is a novel technique that will aid in decision making prior to software release in an attempt to minimize high maintenance overheads.

1. Introduction

The concept of technical debt, first introduced by Cunningham (1993), refers to the debt incurred through the speeding up of software project development which results in a number of deficiencies ending up in high maintenance overheads. These deficient and imperfect traits in the software development prior to project release will have to be paid in the near future as an uncontrolled maintenance cost. The technical debt metaphor, a major issue in software development and maintenance, negatively affects software quality and has recently been a focus for several research studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Ramasubbu, 2013, Fernandez-Sanchez et al., 2015, Kruchten et al., 2012, Maldonado and Shihab, 2015, Wehaibi et al., 2016, Mensah et al., 2016).

With the increasing pressure of expediting software products for users, project managers obliged to meet stipulated deadlines and short-term business benefits, tend to impose pressure on their programmers. As a result, these programmers commit incomplete codes, buggy codes and temporary fixes that produce errors and require rework so as to meet the need of customers who demand quality and robust applications. These errors are assumed as intentional mistakes by the software development team. Potdar and Shihab (2014) describe this phenomenon of intentional weak software development process resulting in series of long-term overheads in the maintenance phase as Self-admitted Technical Debt (SATD). Thus, SATD refers to the incomplete or temporary fixes which are intentionally committed by developers and admitted as mistakes during software development. The SATD metaphor is gradually becoming a research focus (Potdar and Shihab, 2014, Bavota and Russo, 2016, Maldonado and Shihab, 2015, Wehaibi et al., 2016) whereby researchers aim at finding solutions for combating or minimizing the developers' misconducts and shortcuts of producing less quality applications. This misconduct in development is sometimes based on decisions that prioritize functionality over quality (Fernandez-Sanchez et al., 2015) and goes a long way to negatively affect software maintenance (Zazworka et al., 2011).

A plethora of prior studies Zimmermann et al. (2008) and Fluri et al. (2007) conducted in recent years focused on software quality issues in relation to software bugs introduced by developers and assumed such bugs as mistakes. However, very little is known about the proportion of different kinds of SATD in a software project. Knowledge of the proportion could help software practitioners prioritize their scarce testing resources. A potential way of determining the proportions is by examining the source

* Corresponding author.
E-mail addresses: smensah2-c@my.cityu.edu.hk (S. Mensah), Jacky.Keung@cityu.edu.hk (J. Keung), jeff.svajlenko@usask.ca (J. Svajlenko), kebennin2-c@my.cityu.edu.hk (K.E. Bennin), Qing.Mi@my.cityu.edu.hk (Q. Mi).

https://doi.org/10.1016/j.jss.2017.09.026
0164-1212/

code comments of the software project. Software comments (auxiliary segments which are not compiled) play a key role in understanding the source codes in relation to software development and maintenance. Inspired by previous works by Potdar and Shihab (2014) and Fluri et al. (2007), we perform an exploratory study on source code comments of open source projects whereby SATD tasks are prioritized to know the extent of its negative effects on software quality. According to Devroey et al. (2013), prioritization can be used to sort items and optimize coverage criteria or assign weights to features. Prioritization of technical debt items is an important activity in the software engineering domain since it balances short-term product release and long-term effects associated with the software release (Martini et al., 2014). According to Li et al. (2015), there is the need for more studies to be conducted on how to prioritize technical debt task list in order to maximize profit.

In the attempt to deal with the prioritization of the SATD metaphor, we introduce a 6-step prioritization scheme to provide a framework for minimizing SATD based on identified textual indicators by Potdar and Shihab (2014). We construct a text mining algorithm to mine indicative source code comments of SATD from four large open source software projects. Thus, we make use of extracted source code comments and empirically investigate the causes of SATD and estimate the rework effort involved in addressing them. We empirically validate the proposed prioritization approach on four open source projects using Recall and Precision accuracy measures. We further perform statistical tests using the Welch's *t-test* and Cliff's δ effect size as recommended by Kitchenham et al. (2016) to confirm the statistical significance of our results.

Results from this study show that about 30–38% of the identified SATD tasks were major tasks which on average developers face difficulty in addressing prior to software release. This resulted in a rework effort estimation of approximately 10–25 commented LOC per SATD source file needed to address highly prioritized SATD or *vital few* tasks. Finally, seven distinct causes of SATD were found with code smells, complex tasks, inadequate code testing and unexpected code performance being the most dominant causes. The prioritization scheme can form a decision-making baseline prior to software release to assist software engineers in minimizing SATD and its related impacts during software development.

Main contributions of this study include:

- a prioritization scheme for addressing SATD tasks.
- a text mining algorithm for SATD detection and prioritization.
- a rework effort estimation of prioritized SATD tasks.

The remaining sections of the paper are organized as follows. Section 2 presents reviews of related work. Section 3 presents details of the 6-step prioritization scheme as well as the methodological procedure employed in conducting the study. Section 4 discusses the results from the empirical analysis of the study. Section 5 presents the threats to validity and Section 6 gives the conclusions and future direction of the study.

2. Related work

Several works have been done in the field of technical debt in general (Ramasubbu, 2013, Li et al., 2015, Farias et al., 2015, Kruchten et al., 2012, Martini et al., 2014). Farias et al. (2015) performed an exploratory analysis on two large open source projects to identify technical debts through source code comment analysis. They proposed a model, namely CVM-TD which provides a vocabulary that can be used to detect technical debts. Results from their study indicate that developers use dimensions considered by CVM-TD model to assist them in writing source code comments during development. They concluded from their study that technical debts can be identified from source code comment analysis. Ramasubbu (2013) developed and validated the theory of accumulation of technical debt that measures the impact of tradeoffs between customer satisfaction and software quality. Their study was performed using a commercial software product at the various stages of the product's lifecycle. Their proposed theory provides relevant cost estimation and benefits of technical debt for the adoption of a commercial software product.

In relation to our study, we focused on technical debts which are self-admitted by the development team. The concept of SATD was introduced by Potdar and Shihab (2014) who performed an exploratory study on four large open source projects to study the amount, reason and likelihood of removing SATD after its introduction during software development. In their study, they manually examined the open source code comments and found 62 textual indicators for identifying SATD. Out of the 2.4–31% of source files that contained SATD, most of them were introduced by more experienced developers as compared to less experienced developers. They also found that time and code complexities do not correlate with the amount of SATD. Lastly, they found that even though these SATD are meant to be eliminated after its introduction, close to 30% were not addressed or removed from the software projects analysed. A follow-up study by Maldonado and Shihab (2015) identified and quantified SATD from at least 33,000 source code comments into five main types, namely requirement debt, design debt, test debt, defect debt and documentation debt. They found that design debts were the most common type of debt forming about 42–84% of the detected SATD comments. Another study by Wehaibi et al. (2016) examined the impact of SATD on software quality. They found that changes made on SATD tasks induced less defects in future as compared to non-SATD tasks and that these changes are more complex to be made. Bavota and Russo (2016) recently conducted a *differentiated replication* (Wohlin et al., 2012) study of Potdar and Shihab's work to investigate the diffusion rate of SATD in software development. In their study, they (Bavota and Russo, 2016) mined over 2 billion source code comments of 159 Java open source projects and identified an average of 51 SATD instances per each project. In their manual categorization of SATD inspired by the Grounded theory principles (Corbin and Strauss, 1990), they found that 30% of the SATD instances was code debt, 20% represented requirement and defect debts, and 13% represented design debt. They also found that the number of SATD instances increases with the change history, and in most cases (63%) the same developers who introduced the debt were the same fixing the debt. Another recent study by Mensah et al. (2016) estimated the rework effort needed to address the SATD tasks in open source projects. Result from the study shows that, a rework effort of approximately 13–32 commented LOC is required to address the SATD tasks in each source file of the software projects analysed.

Even though this study makes use of source code comment analysis as done in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Wehaibi et al., 2016, Maldonado and Shihab, 2015, Farias et al., 2015, Mensah et al., 2016), it is unique since we contribute with a 6-step prioritization scheme and a text mining algorithm for prioritizing SATD tasks (or *vital few* tasks). Thus, the text mining approach does not only detect SATD tasks as demonstrated in a previous study (Farias et al., 2015) but also prioritize the identified debts based on the prioritization scheme and estimates the rework effort needed with the sole aim of assisting in efficient decision making prior to software release. Again, whilst manual exploration of SATD was performed in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Maldonado and Shihab, 2015, Wehaibi et al., 2016), we automate the process by introducing a text mining algorithm for SATD analysis. It should be noted that, this study makes use of an effort estimation metric in our recent work (Mensah et al., 2016) to estimate the rework ef-

fort needed to address the prioritized SATD tasks (*vital few*). Even though the same projects were used, this study differs from our previous study (Mensah et al., 2016) since we focused on estimating the rework effort needed to address the prioritized SATD tasks.

3. Empirical study design

The main goal of this study is to introduce a 6-step prioritization scheme comprising mainly of identification, examination and rework effort estimation of prioritized SATD tasks. In this study, we mine and analyse SATD tasks in extracted software projects with the intention of not only examining the *diffusion* (Bavota and Russo, 2016) of SATD but also prioritizing the *buggy prone* SATD tasks that require rework. We make use of manually extracted textual indicators identified by Potdar and Shihab (2014) to build a text mining algorithm with the aim of automating the exploratory analysis of SATD. To confirm the manual exploratory process by Potdar and Shihab (2014) with the text mining algorithm, we make use of the same open source projects and manually verify the SATD textual indicators identified. This is achieved by making use of a statistical stratification method (De los Santos et al., 2007) whereby each corpus of source code comments is equally partitioned into ***q*** strata. For consistency, we randomly choose a ***q*** value of four for each corpus. We examine each stratum per project and extract source code comments forming instances of SATD. We present a sample of the source code comments forming instances of SATD below.

Examples of SATD comments
* Don't wait around; just abandon it *
* Leave it for next release *
* Do nothing and bail out *
* Strictly speaking, this is a design error *
* DESIGN ERROR: a mix of repositories *
* TODO: this isn't quite right but it's ok for now *

After assessment and confirmation of the SATD textual indicators based on a manual exploratory process, we incorporate the vocabulary of key textual indicators into the text mining algorithm to facilitate the 6-step prioritization scheme for addressing SATD.

3.1. Prioritization scheme for SATD: a new approach

Prioritization of SATD refers to the identification of a subset of *buggy prone* SATD tasks from a pool of task list based on predefined rules to ascertain which SATD tasks are of key importance to be repaid first prior to software release or repaid later in future release. We define such *buggy-prone* SATD tasks as *vital few* tasks. Prioritization can be used to sort items and optimize coverage criteria or assign weights to features. Prioritization plays an important role in software development since it assigns priorities to a pool of tasks to assist developers to know which tasks need to be addressed most (Devroey et al., 2013). We climax the proposed 6-step prioritization scheme with a rework effort estimation to help identify the level of effort that will be expended when repaying for the prioritized debts. The 6-step prioritization scheme is listed below:

A 6-step prioritization scheme for SATD
Step 1: Identify SATD task list
Step 2: Gap analysis between task complexity and significance
Step 3: Distinguish between the *expected* and the *expedited* tasks
Step 4: Separation of the *vital few* tasks from the *trivial many* tasks
Step 5: Identify plausible causes of key SATD tasks
Step 6: Break-point for release decision

Following a manual exploratory process by Potdar and Shihab (2014), we further extract textual indicators for *major, minor, expected, expedited, vital few* and *trivial many tasks* by reading through over 600,000 source code comments of the studied projects. We then develop a text mining algorithm to facilitate mining of the SATD tasks. The text mining algorithm is described in Algorithm 1 as well as the performance evaluation in Section 3.8. A general overview of the prioritization scheme is depicted in Fig. 1 with an elaboration of each of the 6 steps provided. Thus, from the general overview of the prioritization scheme, the extracted SATD task list is partitioned into *major* and *minor* tasks (step 1 of prioritization scheme). *Expected* tasks from the developers are further extracted from the *major* and *minor* tasks based on a dictionary of textual indicators. Software developers sometimes use shortcuts or time savers to meet hard deadlines from project managers and customers. Such time savers (or *expedited* tasks) are further differentiated from the *expected* tasks (step 3). In order to know the complexity of *major* tasks addressed by developers, we further investigate the gap analysis

Algorithm 1 Source code comment text mining.

Notations:
P: remove punctuations function
SW: remove stop words function
StF: stemming function
Q: total number of commented tasks per project
D: total number of SATD commented tasks per project
SD: documents void of stop words
SATD: List of SATD comments (document)
Class: SATD textual indicators (query)
CoS: Cosine similarity between document and query
Top_k: Top *k* documents with a high lexical similarity with query
Conf_Mat: Confusion Matrix
Rec: Recall measure
Prec: Precision measure
tfidf: term frequency inverse document frequency

Input:
SCC: Source code comments from developers and contributors
StD_category: dictionary of SATD categorization of textual indicators
punct[]: array of punctuation characters
StW[]: array of stopwords
StD: array of SATD indicators

Output:
RW_{VT}: Rework effort estimation for *vital few* and *trivial many* SATD tasks

Procedure

```
// Remove Punctuation Characters
P ← remove_punct (SCC, punct[])
// Remove Stop Words
SW ← remove_stop.words(P, StW[])
// Stemming of words
SD ← stem.document(SW, StF)
//Extract SATD comments from corpus
for i, i=1,...,Q do
    SATD[i] ← extract_satd(SD[i], StD)
end for
// Categorization of SATD Tasks
for l, l=1,...,D do
    class[l] ← categorize(StD_category[l])
end for
// Compute term weights for SATD list using tfidf
for j, j=1,...,D do
// Compute number of terms(t) per each SATD comment
    tf_tot[j] ← compute(SATD[j], length)
        tf[j] ← count(t terms) / tf_tot[j]
            idf[j]← log(D / sum(grepl(SATD[j], ignore.case)))
                tfidf[j] ← tf[j] * idf[j]
end for
// Compute cosine similarity between tfidf for document (tfidf_d) and query (tfidf_q)
CoS ← cos(tfidf_d, tfidf_q)
// Rank in descending order and select the top k documents
Top_k ← select_k(rank(CoS))
// Performance Evaluation
        Compute Conf_Mat
            Output Rec and Prec
// Compute Rework Effort for Top_k vital few and trivial many tasks
    RW_VT ← compute(Top_k)
        Output RW_VT
```

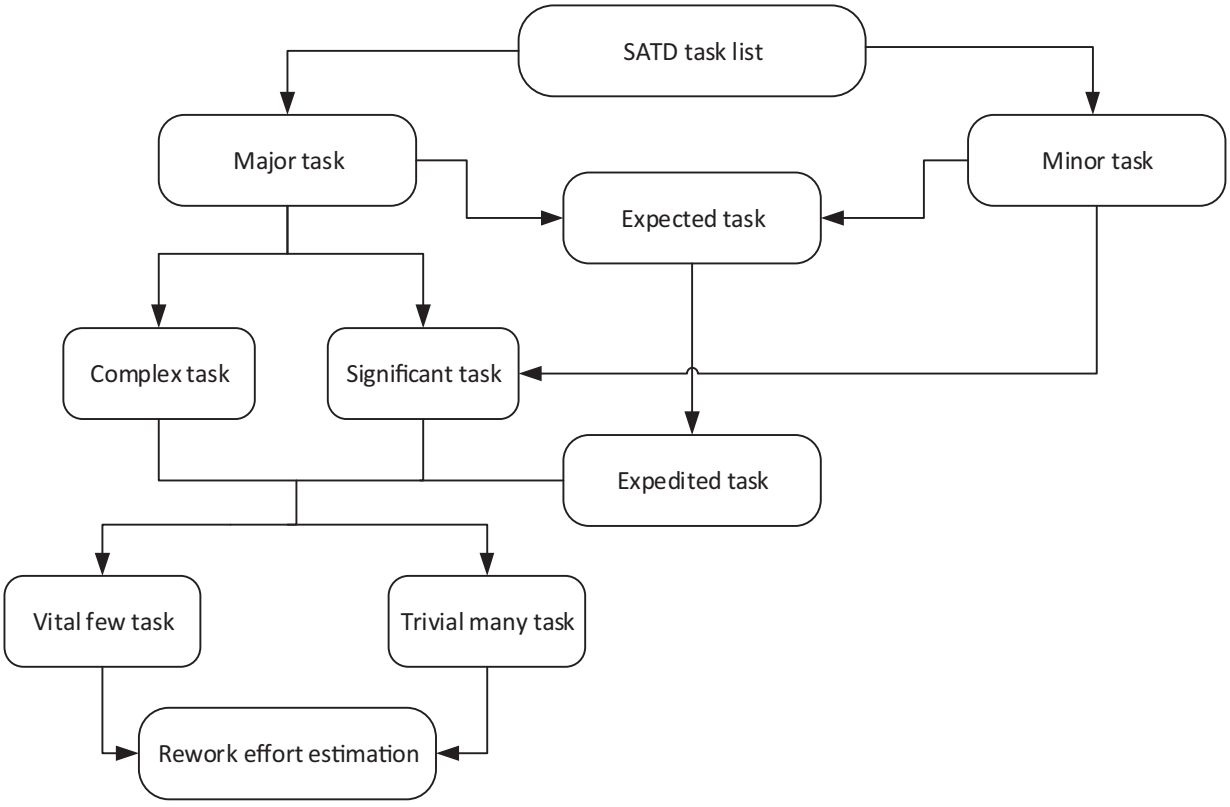


Fig. 1. A general overview of the prioritization scheme for SATD.

between *complex* and *significant* tasks as elaborated instep 2 of the prioritization scheme. A resulting subset of buggy prone (or *vital few*) and less buggy prone (or *trivial many*) tasks are obtained as the prioritized and less prioritized tasks respectively (step 4). We identify plausible causes of detected buggy prone tasks (step 6) and lastly estimate the rework effort needed to address both the *vital few* and *trivial many* tasks (step 6). These causes have to be given much attention during software development before reaching a break-point for time-to-market release. We describe the 6-step prioritization scheme as follows:

Step 1: Identify SATD task list

In order to prioritize SATD tasks, there is the need to first investigate the various SATD tasks that are mostly addressed by developers. In the attempt to tackle this issue, we make use of a vocabulary of extracted textual indicators (Potdar and Shihab, 2014) to identify all comments which are instances of SATD. This is made possible by constructing a text mining algorithm (Algorithm 1) and using the vocabulary of textual indicators to mine a corpus of extracted source code comments to identify instances of SATD tasks. The subset of extracted source code comments prone to SATD forms the SATD task list. A sample of these textual indicators are *bail out, give up, FIXME, temporary solution, kludge, workaround* and the remaining textual indicators are made publicly available.[1] After the SATD task list identification, we separate the *major tasks* from the *minor tasks*. Thus, a subset of textual indicators with respect to urgency, seriousness and significance is used to identify the *major tasks* whilst the remaining tasks form the *minor tasks*. Examples of textual indicators forming instances of *major* task identification are *design error, remove me before production, fix me before production, system malfunctioning, still bugs, kludge, test failure, untested, problematic, add it now* and *hack for testing*. Examples of textual indicators forming instances of *minor* task identification are *skip me, ignore, leave it, in future release, workaround, don't worry, check runtime, rename, poorly defined, does not match, eliminate, implement this*, and *remove me*.

Step 2: Gap analysis between task complexity and significance

After identification of developers' SATD task list from step 1, there is the need to find the gap between task complexity and significance of the tasks addressed by developers who are sometimes constrained with hard deadlines. *Task complexity* refers to the nature of a given task which is difficult and sophisticated to accomplish (Nakatsu et al., 2014) whilst *task significance* refers to the essential relevance of a given task for effective functioning of a system. We examine all complex tasks and those which are very essential for system functioning prior to software release and classify them as task significance. The optimal goal is to examine the discrepancies between the complexity of tasks addressed by developers and how important these tasks affect the project life cycle if left unresolved. We examine the source code comments from the SATD task list (step 1) with the text mining algorithm and extract SATD tasks that are complex to address by developers. Examples of such tasks are "*Bail out if we cannot get the context*" and "*Serious security issue. Bail out rather than risk it!*". If such complex tasks are left unresolved, it increases the rate of system malfunctioning which affects the quality assurance of the software system (Kamei and Shihab, 2016). A sample of textual indicators for task complexity are *sophisticated, bail out, give up, this is killing, serious hack, workaround bug, difficult, cannot fix* and *nasty hack* whilst sample of textual indicators for task significance are *fixme before production, finishme before production, system malfunctioning, test failure, hack for testing* and *rework needed before production*. In our exploratory analysis, we realized that developers sometimes overlook these complex and significant tasks before deployment which can incur uncontrollable overheads during software maintenance. We also found that all *complex tasks* are a subset of *major*

[1] http://tinyurl.com/SATD-textual-indicators.

tasks whilst *significant tasks* can be a subset of *major* or *minor tasks* depending on how important those tasks need to be addressed before software release (Fig. 1).

Step 3: Distinguish between "expected" and "expedited" tasks

In this step, we mine the corpus of developers' source code comments and extract textual indicators from the SATD task list forming instances of *expected tasks.* We describe *expected tasks* as the actual or anticipated tasks required to be addressed by developers. Even though a *major* or *minor* task can be considered as an *expected task* (Fig. 1), the main issue here is whether all *expected tasks* are addressed appropriately or not. As a result of the hard deadline set for developers as well as meeting their customers' needs, they engage in using series of shortcuts to complete the tasks. Even though such shortcut codes (normally bug free) are workable, it does not always guarantee that the logic is correct. This is because, most of these shortcuts or timesavers used by developers sometimes lead to logical failures. We describe such shortcuts or timesavers adopted by developers as *expedited tasks.* This step seeks to differentiate *expedited tasks* from *expected tasks* in the SATD task list. The differentiation can help determine the rate of *expedited tasks* left in the studied projects prior to software release into the market. Since these *expedited tasks* comprising of series of shortcuts and timesavers do not always guarantee a quality software, we extract textual indicators from the SATD task list forming instances of *expedited tasks.* A sample of textual indicators of *expedited* tasks are *workaround, nuke it, temporary hack, really ugly, silly, yuck* and *nasty error.* On the other hand, we extract textual indicators for the actual development tasks (*expected tasks*) which developers normally use shortcuts or timesavers to complete. A sample of textual indicators forming instances of *expected tasks* are *fixme, todo, finishme, must fix it, we ought to fix, add it, hack for, detect, generate, rework* and *rewrite.*

Step 4: Separation of "vital few" tasks from "trivial many" tasks

The Pareto principle states that about 80% of a given problem in a system is as a result of about 20% of the causes (Fenton and Ohlsson, 2000). Thus, majority of buggy prone tasks (80%) within a software project is caused by few unaddressed tasks (20%) during the development process. This handful of unaddressed tasks are the variant of SATD that requires rework and overlooked by developers prior to software release. We describe the few unaddressed SATD tasks which are highly prone to software bugs as the *vital few* tasks. The *vital few* tasks result in a buggy and malfunctioning system if not resolved prior to software release (Wehaibi et al., 2016). On the contrary, the less buggy SATD tasks are regarded as the *trivial many* tasks. Similar to previous studies by Potdar and Shihab (2014), we perform a manual exploration on over 600,000 code comments to extract textual indicators for *vital few* and *trivial many* tasks. These textual indicators are incorporated in the text mining algorithm (Algorithm 1 in Section 3.7) to separate the *vital few* tasks from the *trivial many* tasks for each project source file. A sample of *vital few* textual indicators are *remove me before production, bail out cannot do this, we hang our heads in shame, still bugs in the system, give up looking for the bug, cannot introspect, design error, nasty hack, too much bugs, really dirty hack, large hack here, complex to hack, complex to debug, this bug is killing,* and *cannot introspect*. A sample of *trivial many* textual indicators are *remove when bug is resolved, test commented out until bug is fixed, temporary workaround for bug, temporary hack, fixme fails here, fixme this seems wrong, find a better way to fix bug* and *ugly trick for bug.*

Step 5: Identify plausible causes of key SATD tasks

After separation of the *vital few* tasks from the *trivial many* tasks in step 4, we identify the plausible causes associated with the buggy prone SATD tasks based on manual exploratory process. We examine the source code comments forming instances of buggy prone SATD tasks with the text mining algorithm and identify the plausible causes for such tasks left unaddressed by developers during the development process. These plausible causes can help in managerial decisions for reducing the *diffusing rate* (Bavota and Russo, 2016) of SATD for future release of the software projects. Again, since unaddressed SATD needs to be paid back as a debt in the next software release, identification of plausible causes of key SATD tasks need to be given much attention in order to minimize SATD during software development.

Step 6: break-point for release decision

Based on the plausible causes of buggy prone SATD tasks identified in step 5 and the rework effort (see Section 3.2) involved in addressing them, decisions can be made on the time-to-market release for the software project. Here, we implement a rework effort estimation metric within the text mining algorithm to estimate the rework effort involved in addressing such extracted and unresolved buggy prone tasks within the software. Thus, based on the rework effort estimate for the extracted and unresolved buggy prone tasks, the necessary managerial decisions can be made in reaching the break-point or time-to-market release of the developed software. Here, buggy prone SATD tasks are of major concern since they are highly problematic and can result in a buggy system in future release or during future software maintenance. Following a similar approach in our previous study (Mensah et al., 2016) in estimating the rework effort for SATD, we estimate the rework effort for prioritized SATD tasks, namely the *vital few* tasks in the next section. Similarly, the rework effort estimation metric is also used to estimate the rework effort needed for *trivial many* tasks for managerial decision making. Thus, we further estimate the rework effort need to address all *trivial many* tasks.

3.2. Rework effort estimation of vital few and trivial many SATD tasks

In the quest for investigating the extent of rework effort in relation to commented lines of code (LOC) forming instances of *vital few* and *trivial many* tasks, we formulate a rework effort estimation metric based on a study by Zhao et al. (2015) and Mensah et al. (2016). The computational metric for estimating the Rework Effort of *vital few* and *trivial many* tasks (RW_{VT}) is defined in (1).

$$RW_{VT} = \frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{k} LOC(f_j)\right)_i}{\sum_{i=1}^{m} f_i} \quad (1)$$

where

n = *total number of source files per a software system*
m = *total number of source files forming instances of vital few or trivial many tasks per system*
k = *total number of LOC forming instances of vital few or trivial many tasks per a source file*
f_i = *an* ith *source file*
$LOC(f_j)$ = *number of LOC forming instances of vital few or trivial many tasks in an* f_j *source file*

For example, given a software project containing n total number of source files. If there exist m source files which is a subset of n (i.e., $m \subseteq n$) and forming instances of *vital few* tasks. Assume after subjecting each ith source file, f_i to the rework effort estimation metric results in k commented lines of code (LOC) forming instances of *vital few* tasks. Then, the rework effort for the *vital few* tasks (RW_v) can be computed as the ratio of the total lines of code contained in all the n source files to the m source files forming instances of *vital few* tasks. Thus, the rework effort

estimated for addressing the *vital few* tasks for each source file on average is RW_v. The RW_v is computed in commented LOC per each *vital few* source file. Similarly, the rework effort for *trivial many* tasks (RW_T) can be computed in commented LOC per each *trivial many* source file. Thus, the estimation metric provides an estimate in lines of code per SATD source file that requires rework with respect to *vital few* or *trivial many tasks*.

The rework effort estimation metric will assist developers in the identification and rectification of overlooked *vital few* or *trivial many* SATD tasks and hence minimize its related negative impacts during software development process.

3.3. Research questions

The study aims at addressing the following five research questions (*RQ*):

- ***RQ1:*** *How can developers' major tasks be distinguished from minor tasks in open source projects based on extracted SATD task list?*
  Motivation: In order to meet short term deadlines, satisfy the needs of customers and ensure high quality delivery of software projects, developers are in a rush to complete assigned tasks. Due to the hasten pressure imposed on developers, they are normally faced with a pool of SATD task list to address which by the hard deadline, not all can be addressed before software release. This pool of task list comprises of *major tasks* (significant, important and urgent tasks) and *minor tasks* (less important or serious tasks). Differentiating the SATD task list into major and *minor tasks* can assist project managers and developers to make effective decisions on which tasks to assign high priorities. This will also assist in knowing which tasks are complex or less difficult to address prior to software release deadline and those to leverage to the maintenance phase.
  Approach: This *RQ* is complemented by a manual extraction process of *major* and *minor* textual indicators from the developers' source code comments. Thus, we first read over 600,000 source code comments from over 20,000 source files in the four open source projects to perform the extraction process. These textual indicators form the *bag-of-words* (Zhang et al., 2010) that was used to mine each of the open source projects. Thus, the *bag-of-words* and the source code comments are used as input and the manually extracted *major* and *minor* task list form the target output for the text mining process (Algorithm 1 in Section 3.7). We evaluate the text mining process for the *major* and *minor* tasks from the studied projects by constructing a confusion matrix to know the precision and recall measures. It should be noted that the actual or target variable is the manually extracted *major* and *minor* tasks whilst the output variable is the *major* and *minor* tasks as reported by the text mining algorithm. We further investigate the existence of statistical significant difference between the manual and text mining process using the Welch's *t*-test and Cliff's δ effect size. Further details are provided in Section 3.8.
- ***RQ2:*** *What is the difference between "expedited" and "expected" tasks in open source projects?*
  Motivation: After extracting the *major* and *minor* task list, there is the need to know the procedures and measures developers undertake to meet the project delivery demands. Developers obliged to meet hard deadlines tend to use series of shortcuts to leave in important tasks during software development. We define such development shortcuts which is a form of self-admitted technical debt as *expedited tasks*. On the other hand, if developers fail to use development shortcuts but follow the anticipated or actual development procedure to address assigned tasks, then such phenomenon is described as expected tasks. If such *expedited tasks* are identified and differentiated from *expected tasks*, it will help in minimizing the *diffusing rate* (Bavota and Russo, 2016) of SATD and result in high quality delivery of software projects.
  Approach: Following a similar process by *RQ1*, we extracted textual indicators forming instances of *expedited* and *expected tasks*. We then perform a text mining analysis with the extracted textual indicators and evaluated the prediction performance using the precision and recall measures from the constructed confusion matrix. Again, we use the Welch's *t-test* and Cliff's δ effect size to find the existence of statistical significant difference between the manual and text mining approach.
- ***RQ3:*** *How can buggy prone (or "vital few") tasks be prioritized from a pool of SATD tasks in open source projects?*
  Motivation: Software system's malfunctioning is as a result of faults, flaws, errors or failures regarded as software bugs or defects by Tang et al. (2015). According to the Pareto Principle, about 80% of a given system malfunctioning problem is caused by about 20% of unaddressed tasks (Fenton and Ohlsson, 2000). In this *RQ*, we argue that there exist a subset of SATD task list that contributes to the 20% unaddressed tasks prior to software release. We refer to these unaddressed tasks which are buggy prone instances of SATD as *vital few tasks*. On the other hand, SATD instances which are not buggy prone are regarded as *trivial many tasks*. The main focus is to provide a heuristic approach to prioritize the vital few tasks from a pool of SATD task list. It should be noted that since *trivial many tasks* do not contribute to majority (80%) of a system's malfunctioning, they were of less significance to this study. Nevertheless, we provide sample of textual indicators forming instances of *vital few* and *trivial many* SATD tasks in step 4 under Section 3.1.
  Approach: For this particular *RQ*, we follow the heuristics approach by Maldonado and Shihab (2015) to first extract the five classes of SATD, namely requirement debt, design debt, test debt, defect debt and documentation debt. We then read through the SATD task list to identify textual indicators forming instances of buggy prone or *vital few tasks*. Similarly, textual indicators were also identified for the *trivial many* (or less buggy prone) tasks. It should be noted that the heuristic approach followed in the extraction of textual indicators for *vital few* and *trivial many tasks* was the same utilized in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Maldonado and Shihab, 2015, Wehaibi et al., 2016). These textual indicators are used to perform a text mining analysis of the four studied projects to quantify the respective vital few and trivial many tasks in each system. We validate this approach by creating a confusion matrix for the recall and precision measures as well as the statistical significant difference tests as elaborated in *RQ1* and *RQ2*.
- ***RQ4:*** *What is the extent of rework effort required to resolve the buggy prone (or vital few) tasks in open source projects?*
  Motivation: After obtaining the *vital few tasks* contributing to majority of a software system's malfunctioning, there is the need to estimate the rework effort required to address them. Knowing the amount of rework effort required will enable in an effective task planning and scheduling in order to mitigate the expediting pressure from project managers and customers. We concentrate on obtaining the rework effort for *vital few tasks* since they are of paramount interest in negatively affecting software quality prior to software project delivery.
  Approach: Inspired by an effort estimation metric by Zhao et al. (2015), we develop a rework effort estimation metric following a similar approach in our previous study (Mensah et al., 2016) for the buggy prone tasks. This was done by mining the code comments for each source file of a project with the *vital few* textual indicators (Algorithm 1). The respective commented

Table 1
Description of open source projects.

Metric	Open source project			
	ArgoUML	Chromium	Eclipse	Apache
LOC	122,575	107,706	659,231	192,333
Comment	115,713	37,889	437,640	54,295
SATD comment	300	1364	447	591
Developers	53	1784	221	145
Code files	1846	14,328	6389	255
SATD files	77	574	152	179
Date	Dec, 2011	Nov, 2009	Jun, 2013	Jul, 2013
Version	0.34	30	4.3	2.4.6

lines of code (LOC) forming instances of the *vital few tasks* per each source file are obtained. The LOC prone to the *vital few tasks* are summed with respect to all the source code files and divided by the total number of source code files forming instances of the *vital few tasks*. The rework effort metric is computed in commented LOC.

- ***RQ5:*** *What are the plausible causes of SATD tasks from the source code comments analyzed?*
  Motivation: As reported in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Wehaibi et al., 2016, Mensah et al., 2016), SATD affects quality delivery of software projects. Exploring the plausible causes of SATD is of great importance since these causes can be rectified or prevented from affecting the quality development of a project. The key interest is to reduce SATD to the barest minimum with the intuition that, SATD can be minimized in future software release or development process.
  Approach: *RQ5* is complemented by exploring the source code comments and identifying which comments indicate SATD. Here, we read over 600,000 source code comments and identify key factors which form instances of SATD.

3.4. Data extraction

For the purpose of this study, we make use of four well-commented open source projects made available on *openhub.net* and first extracted by Potdar and Shihab (2014) for studying SATD. The four projects are ArgoUML, Chromium OS, Apache HTTP Server and Eclipse Platform project. For each project, we extract the source code comments using srcML (Collard et al., 2011) by converting the source codes into XML and storing the comments in separate text files. We first extract the source code comments forming instances of SATD from over 600,000 code comments using Potdar and Shihab's (2014) textual indicators. Secondly, we read through all the extracted SATD comments (or SATD task list) to further extract textual indicators forming instances of *major and minor tasks, expected and expedited tasks, complex and significant tasks,* and *vital few and trivial many tasks.* These textual indicators form a *bag-of-words* (Zhang et al., 2010) for constructing a text mining algorithm for performing an exploratory and empirical analysis of the SATD tasks in the four projects studied. We present a general overview of the data extraction process in Fig. 2. The description of the open source projects is presented in Table 1. In each project, we extract metrics such as the number of lines of code (LOC), number of source code comments, dates of software release, software release versions and contributors or developers for each open source project.

3.5. Data preprocessing methods

Preprocessing is an important phase in text mining and text classification. For an efficient regular expression matching or pattern recognition, we preprocess the extracted open source code comments based on the following as elaborated by Ye et al. (2014) and Kotsiantis et al. (2006):

Data Cleaning: In the dataset cleaning process, we use the text mining algorithm (Section 3.7) to remove punctuation marks and special characters in the form of *∼!@,.-#$%^*][|* from the corpus of source code comments. Again, we filtered out noise in the form of blank lines and white spaces within strings from each project using the text mining algorithm. Typical examples of white spaces within strings are "*FIX ME*", "*TO DO*" and "*FINISH ME*" which most developers use in various comments compared to "*FIXME*", "*TODO*" and "*FINISHME*". This filter approach enables the pattern recognition process to obtain desired searched patterns during the text mining process. Source code comments which are not instances of SATD are removed by the text mining algorithm.

Stemming: This is a method of identifying the root or stem of a word (Fu et al., 2016). For example, the words *fix, fixing, fixes* and *fixed* can all be stemmed to the word *"fix"*. Similarly, the words *hack, hacks, hacking* and *hacked* can all be stemmed to the word *"hack"*. We incorporate the Porter stemming method (Ye et al., 2014) as implemented in the NLTK package[2] into the text mining algorithm to obtain the root of words in the source code comments. Stemming aims at eliminating various suffixes, maintaining accurately matching stems, maintaining memory space and latency (Kotsiantis et al., 2006). When words are stemmed to a particular root, it positively improves the recall performance of the text mining algorithm (Ye et al., 2014).

Stop word Filtering: Here, variant of information retrieval stop words such as determiners, prepositions and conjunctions occurring frequently are removed since they contribute less to the text mining and classification process. To perform an effective text mining with the extracted textual indicators, there is the need to remove irrelevant stop words from the corpus of source code comments. Removal of irrelevant stop words reduces the size of search space as well as memory overheads and speeds up the text mining process since only important terms are focused on in the search process. Thus, given a query (textual indicator) and a document (source code comment) void of irrelevant stop words, we compute the vector space representations (Ye et al., 2014) for both the query and document. We then compute the textual similarity using the cosine similarity between the query and document to obtain an effective textual matching. We provide a variant of irrelevant stop words removed from the corpus of source code comments as follows: *about, above, among, behind, being, but, by, beside, came, certain, come, did, differ, during, different, each, either, even, ever, everything, everywhere, face, fact, further, general, group, he, her, him, herself, himself, kind, like, long, man, woman, myself, nobody, nor, number, old, others, perhaps, place, room, said, saw, she, someone, somebody, their, therefore, thoughts, under, very, want, went, well, whether, year, yet* and *yours.* On the other hand, relevant stop words such as *fix, me, is, to, do, finish, bail, out, catch, miss, cares, give, up, and, remove, more, enough, can, cannot, good, still, really, around, work* and *silly* occurring frequently in the textual indicators such as *fixme, finishme, todo, this is a hack, workaround, bail out, really silly, give up, cannot introspect, remove me before …* are not removed from the text mining process. The irrelevant stop words occurring frequently in the source code comments are searched and removed from all the comments following a similar approach by Can et al. (2008).

3.6. Vector space representation

In this study, we employ the vector space modeling approach Ye et al., 2014) and regard the textual indicators as queries and the

[2] http://www.nltk.org/api/nltk.stem.html.

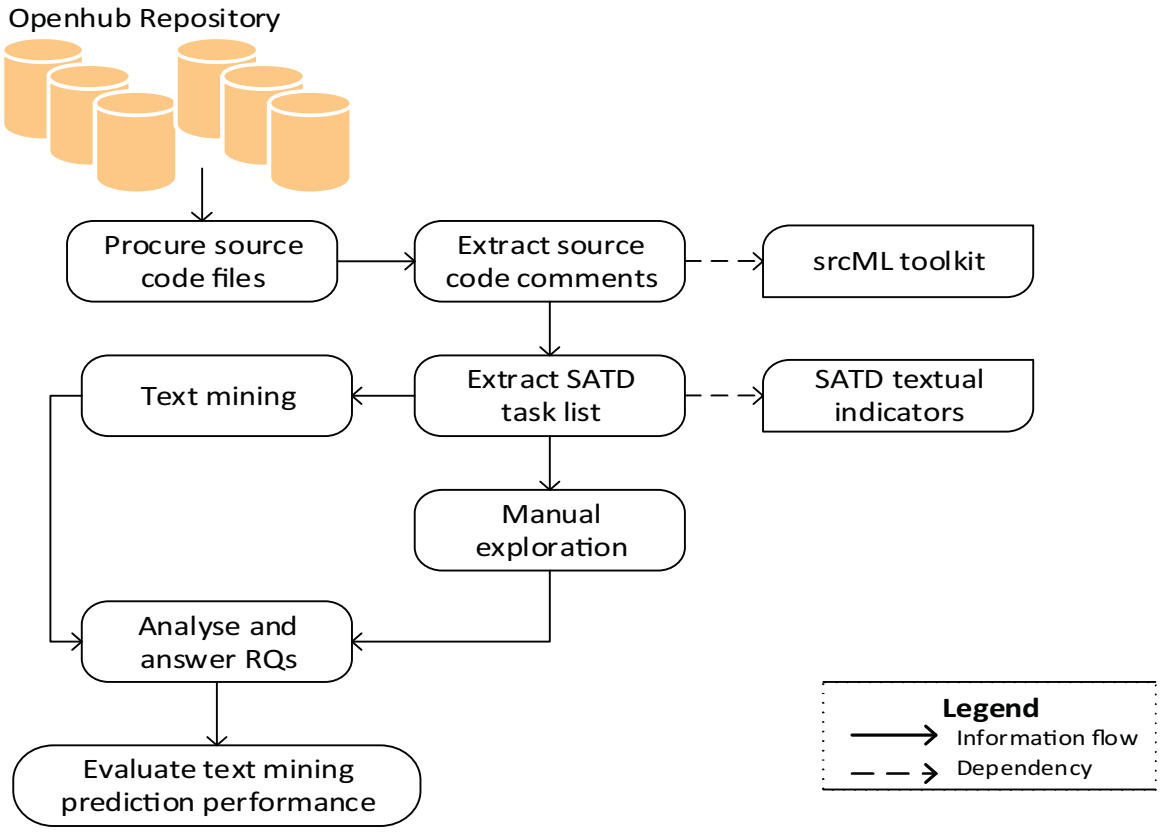


Fig. 2. Overview of the data extraction process.

SATD source code comments as text documents. In this approach, both the document and query are represented as term weight vectors. Thus, we assign term weights to the textual indicators (queries) and the SATD source code comments (documents) for a given project dataset based on the term frequency-inverse document frequency (*tf-idf*) (Manning et al., 2008) which is a well-known classical weighting function in text mining and information retrieval (Ye et al., 2014, Baena-García et al., 2011). This function assigns numeric weights to all terms in a corpus. The assigned value indicates the significance of a term in a document with respect to the entire corpus. *tf-idf* weight is composed of the product of the term frequency (*tf*) and the inverse document frequency (*idf*). We define these two functions in (2) and (3) as well as their product in (4) with respect to each project dataset.

$$tf(t,d) = \frac{f_{t,d}}{m_d} \forall d \in D \tag{2}$$

$$idf(t,D) = \log_e \frac{D}{N_t} \tag{3}$$

$$tfidf(t,d,D) = tf(t,d) \times idf(t,D) \tag{4}$$

where

- $f_{t,d}$ = frequency of term (t) in an SATD source code comment or textual indicator (d)
- m_d = number of terms in a given SATD source code comment or textual indicator
- D = total number of SATD source code comments or textual indicators
- N_t = number of SATD source code comments or textual indicators that contain a given term (t)

A simple example to illustrate the computation of weights for the *tf-idf* is given as follows. Consider a software project with a finite number of SATD source code comments considered in our case as documents. Assume a document contains 6 terms free from irrelevant stop words whereby the term *error* appears 2 times. The *tf* for *error* is then (2/6) = 0.33. Again, assume we have 10,000 documents and the term *error* appears 100 times, then the *idf(error)* is calculated as *log*(10,000/100) = 4.61. Thus, the *tf-idf* weight is the product of these two quantities: $0.33 \times 4.61 = 1.53$.

After obtaining the term weights for both the query and documents, we obtain the lexical similarity[3] between the query (textual indicator) and each of the documents (source code comments). The main goal is to find out which document(s) matches with the query. We make use of the standard cosine similarity function defined in (5) and considered in a previous study by Ye et al. (2014). The cosine similarity function (5) is the ratio of the inner dot product between two vectors to the product of the norms of the two vectors by their respective Euclidean distances. Let d denotes a document of SATD comment from a set of extracted SATD source code comments (D) from a source code file of a given software project whereby $d \in D$. Then, given a textual indicator regarded as a query (q) which is a subset of all textual indicators (Q), we first compute the vector of term weights for D and Q. Secondly, for every q compute the cosine similarity between q and each of the documents, $d \in D$. We rank the resulting cosine similarity scores for the searched query (q) and all the respective documents in non-increasing order and selects the scores with the highest similarity threshold values. For the best lexical similarity match, the cosine function should result in a value close to 1 and for the

[3] Lexical similarity refers to how two texts are similarly related to each other at the surface level.

worse case matching, the function should result in a value close to 0. In order to obtain the best lexical similarity threshold, we empirically investigate how large the cosine similarity function should be for an accurate matching of the query to the relevant documents. This is done by benchmarking the ranking scores obtained with respect to selected documents from the text mining process with documents selected from the manual exploratory process.

$$\cos(q, d) = \frac{q.d}{\|q\|\|d\|} \quad (5)$$

3.7. Text mining algorithm

In this study, we construct a text mining algorithm for mining source code comments of open source projects and extracting textual key terms forming instances of SATD. The text mining algorithm plays a significant role by assigning term weights to the source code comments and query of textual indicators for easy modeling and matching of searched patterns. In order to support the replication of this study, we describe the text mining approach used for the empirical analysis in this section. The text mining approach for source code comment analysis is depicted in Algorithm 1 and it is divided into seven phases as follows:

- *Phase I:* Preprocessing phase of the source code comments
- *Phase II:* Extraction of source code comments forming instances of SATD
- *Phase III:* Categorization of SATD tasks into required classes using textual indicators (query)
- *Phase IV:* Computation of term weights for SATD tasks (documents) and textual indicators
- *Phase V:* Computation of lexical similarity between query and documents
- *Phase VI:* Performance evaluation using precision and recall measures
- *Phase VII:* Rework effort estimation of *vital few* and *trivial many* SATD tasks

Provision of some notations of the various variable names used is made available (Algorithm 1). The algorithm constructed with regular expressions is supplied with the respective source code comments made by the various software developers or contributors. Prior to Phase I, we employ the *textscan* function to read the separated strings in each of the source code comments and their respective developers into two separate vectors. This function also contributed in reading commented strings with whitespaces.

In *Phase I*, punctuation and special characters such as {“ ”:\;!/.@[]-?#%^()’ ’} are eliminated from each of the source code comment using the *punct[]* function and result assigned to the variable *P* (line 1). Irrelevant stop words defined in Section 3.5 are removed in line 2 and the remaining result assigned to the *SW* variable. Stemming is performed on resulting documents to obtain the root of words. Result is assigned to the variable *SD* as shown in line 3.

In *Phase II*, SATD source code comments void of irrelevant stop words are extracted using an implemented *extract_satd* function containing an array of SATD textual indicators (Potdar and Shihab, 2014) in the first *for loop* from lines 4–6.

In *Phase III*, we make use of a dictionary of textual indicators, *StD_category* for the various categorization of SATD tasks (*minor* and *major, expected* and *expedited, complex* and *significant*). Thus, with the help of this dictionary of textual indicators, we are able to search and extract the various categorization of SATD tasks in lines 7–9. We separate the *vital few* and *trivial many* tasks using the respective dictionary of textual indicators as query (see Section 3.1).

In *Phase IV*, we make use of *tf-idf* (Manning et al., 2008, Baena-García et al., 2011) in the third *for loop* statement from lines 10–15 to compute the term weights for the SATD task list. In line 11, the total number of terms per each document (source code comment) is computed and the term frequency (*tf*) computed in line 12 as the ratio of the number of searched and targeted *t* terms to the end result in line 11 (i.e., *tf_tot*). We compute the inverse document frequency (*idf*) in line 13 ignoring case sensitiveness of terms in the *grepl* function. The *grepl* function (found in the *CRAN* library of *R* Software) returns a logical vector containing searched SATD comments (documents). The *tf-idf* values are computed in line 14 for each SATD source code comment. Similar approach is followed to compute the *tf-idf* for the textual indicators (regarded as the query in this study).

In *Phase V*, we compute the cosine similarity between the searched query in the form of a vector of *tf-idf* with the documents also in the form of a vector of their respective *tf-idf* (see cosine similarity function (5) in Section 3.6). Thus, the cosine similarity function between the *tf-idf* of the documents (*tfidf_d*) and that of the query (*tfidf_q*) is computed and the result assigned to the variable *CoS* in line 16. We rank the resulting output (*CoS*) in descending order and select the top *k* values (close to 1) depicting the relevant documents with a high lexical similarity with the query (line 17). It should be noted that each of the documents are assigned with unique IDs for easy traceability of relevant documents.

In *Phase VI*, we evaluate the traced relevant documents from the text mining algorithm to the manually extracted relevant documents. Here, we make use of binary classification whereby for a corpus containing a number of documents, the relevant documents with respect to the searched query are coded with 1 and the irrelevant documents coded with 0. This is done for both the text mining process and the manual exploratory process (benchmark). We compute the confusion matrix to evaluate the text mining results and obtain the resulting *recall (Rec)* and *precision (Prec)* measures in lines 18 and 19. It should be noted that, the confusion matrix is constructed using the binary classification output from the manual exploratory process and that of the text mining process. We elaborate more on the *recall* and *precision* evaluation measures in Section 3.8.

In *Phase VII*, we compute the rework effort estimation for the *vital few* and *trivial many* SATD tasks (RW_{VT}) in line 20 and further explained in (1) of Section 3.2.

3.8. Experimental evaluation of the text mining algorithm

We describe in this sub section the necessary measures taken to evaluate the relative prediction performance of the text mining algorithm. For each corpus of extracted source code comments (documents) from a given software project, we equally partition the corpus into *q* strata or subsets. We randomly chose a *q* value of 4 for consistency across all four projects and also to tally with the stratification threshold for the manual exploratory process of the source code comments. In each iteration process for a given partition subset, all the textual indicators (query) are used for the text mining process. At the end of each iteration process using Algorithm 1, we compute the confusion matrix and evaluate the prediction performance using the precision and recall evaluation measures. When examining the prediction performance of the text mining algorithm, we can have four possible outcomes - *true positives, true negatives, false positives* and *false negatives*. For instance, in a classification task, if the text mining algorithm correctly classifies the relevant documents for a given query into the required class, then the number of relevant documents is referred to as the *true positives*. Again, when irrelevant documents are wrongly classified as relevant documents (or into the correct class) after the search process, then the number of such irrelevant documents are referred to as *false negatives*. In the same way, if relevant documents are wrongly classified as irrelevant docu-

Table 2
Confusion matrix for computing precision and recall.

	Sample size	Prediction outcome from text mining process	
		Predicted positives	*Predicted negatives*
Actual outcome from manual exploratory process	*Actual positives*	*True positive (TP)*	*False negative (FN)*
	Actual negatives	*False positive (FP)*	*True negative (TN)*

Table 3
Precision and recall measures for extracted SATD comments.

SATD Type (Maldonado and Shihab, 2015)	ArgoUML		Chromium		Eclipse		Apache	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Requirement	0.894	0.803	0.821	0.742	0.86	0.848	0.772	0.689
Design	0.961	0.935	0.852	0.853	0.788	0.808	0.787	0.762
Test	0.749	0.822	0.941	0.977	0.814	0.981	0.98	0.898
Defect	0.778	0.981	0.661	0.802	0.884	0.827	0.75	0.681
Documentation	0.739	0.548	0.802	0.861	0.79	0.889	0.905	0.845
Mean	**0.8242**	**0.8178**	**0.8154**	**0.8470**	**0.8272**	**0.8706**	**0.8388**	**0.7750**

ments after the search process, then the number of such relevant documents is referred to as *false positives*. Lastly, when irrelevant documents are correctly classified as irrelevant documents (or into the correct class) after the search process, then the number of such irrelevant documents is referred to as *true negatives*. We define the confusion matrix for computing the precision and recall measures in Table 2. As stated earlier, this study was initialized with a manual exploratory process whereby we read through over 600,000 source code comments and identified all SATD comments forming instances of the various categorization of SATD based on their respective textual indicators. Binary classification was used to classify source code comments forming instances of the textual SATD indicators as 1 and source code comments not forming instances as 0. Similarly, the text mining process resulted in a 1 for correct classification of a searchedquery and a 0 for wrong classification. This made it possible to compute the confusion matrix (Table 2) for the precision and recall measures for this study.

Precision defined in (6) is the probability that a retrieved document or targeted SATD task by the text mining process is relevant. That is, *precision* is defined as the ratio of the number of relevant documents (targeted SATD tasks) retrieved by the search process to the total number of misclassified relevant documents (false positives) and number of relevant documents (true positives) retrieved by that same search process. Alternatively, *recall* defined in (7) is the probability that a relevant document or targeted SATD task is retrieved in the search process. That is, *recall* is defined as the ratio of the number of relevant documents retrieved by the search process to the total number of existing or actual relevant documents.

$$precision = \frac{TP}{TP + FP} \tag{6}$$

$$recall = \frac{TP}{TP + FN} \tag{7}$$

As a result of the differences that might occur between the results from the text mining approach and the manual exploratory process, we further investigated the statistical and practical significance of the size of the difference between the two approaches. This was done using the Welch's *t-test* (Kitchenham, 2015) and the Cliff's delta effect size as recommended by Kitchenham et al. (2016). The Welch's *t-test* is a robust test statistic that is used to test the hypothesis that two groups of observations have equal means irrespective of they having unequal sample sizes and variances. The Welch's *t-test* statistic is the default *t-test* function in the *R* software toolkit (Team, 2016). We interpret the existence of statistical difference at 5% asymptotic significance level whereby *p-values* less than 0.05 depict statistical significant differences between the text mining and manual exploration results.

The rationale behind the Cliff's delta (δ) effect size is that, given two groups of observations without necessarily following the same distribution Kitchenham et al., 2016), what is the amount of overlap that exist between these two groups. Thus, whether a random observation from one group is greater than that of the other group or vice versa. This non-parametric effect size method was also recommended by Arcuri and Briand (2014) for empirical software engineering data analysis. Effect size computation is very important since it provides an appropriate and reliable measure for the practical usefulness of an experimental effect regardless of the *p-value* for the statistical significance for a test statistic (Kitchenham et al., 2016, Arcuri and Briand, 2014). The Cliff's δ effect size is defined in ((8) whereby each observation y_i from the text mining process is compared to each observation y_j in the manual process Cliff, 1993). As defined in ((8), we count the total number of $(y_i > y_j)$ and $(y_i < y_j)$, compute the difference and divide by product of the sample sizes for the text mining (n) and manual exploration (m) observational groups.

$$\delta = \frac{sum(y_i > y_j) - sum(y_i < y_j)}{nm} \tag{8}$$

We interpret the effect size or practical significance based on the magnitude thresholds by Kampenes et al. (2007) and Bavota and Russo (2016) as follows: negligible effect size ($\delta < 0.112$), small effect size ($0.112 \leq \delta < 0.330$), medium effect size ($0.330 \leq \delta < 0.427$) and large effect size ($\delta \geq 0.427$). The Cliff's δ effect size metric was considered in a recent study by Bavota and Russo (2016) to estimate the magnitude of measured differences between the developers introducing and fixing SATD. In order to facilitate the effective computation of statistical test and effect size, we use the term weights of the searched documents from the text mining approach and also computed the term weights for the manually extracted documents.

4. Results and discussion

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

4.1. RQ1: how can developers' major tasks be distinguished from minor tasks in open source projects based on extracted SATD task list?

In order to distinguish between developers' major and minor tasks, we first performed an exploratory analysis to extract

the SATD commented tasks with the text mining algorithm. We realized that out of the preprocessed source code comments in each of the analysed projects, 18.8% SATD commented tasks were found in ArgoUML, 29.3% found in Eclipse, 31.5% and 15.5% found in Chromium and Apache projects respectively (step 1 of prioritization scheme). Prior to the text mining process, a manual exploratory process following similar approaches in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Maldonado and Shihab, 2015) was done to extract all source code comments forming instances of SATD in each project. Examples of such SATD instances from the manual exploratory process are as follows:

Examples of SATD tasks in ArgoUML

* TODO: Bob says - This is a really nasty horrible hack *
* We hang our heads in shame. There are still bugs in ArgoUML *
* FIXME: fails here *
* TODO: This indicates a more fundamental problem *
* We can give up looking if we have hit both criteria *
* This test is a bit stupid *
* NOTE: This is temporary and will go away in a future release *

Examples of SATD Tasks in Chromium OS

* Really ugly, but there is no good way to avoid it *
* Bail out if we are before the onset of daylight savings time *
* FIXME: LOC is incorrect *
* FIXME: this code appears to be untested - and probably unused - and probably wrong *
* FIXME: Figure out how to throw a JavaScript error *
* This is a temporary hack till device manufacturers fix the problem *
* TODO: analyze why we have such a bad bail out here *

Examples of SATD Tasks in Eclipse

* If we have never heard of this object bail out *
* TODO: Hack to work around Bug *
* FIXME: this is killing at least SSE editors see bug *
* FIXME: No implementation. This should be a temporary solution *
* TODO: this isn't quite right but is ok for now until next updates *
* XXX: this is a hack and will not work with some components *
* Cannot introspect. Give up *

Examples of SATD Tasks in Apache

* NOTE: This should never happen, but just be sure it works before production *
* TODO: REMOVE ME BEFORE PRODUCTION (????) *
* DESIGN ERROR: a mix of repositories *
* don't wait around; just abandon it *
* ### strictly speaking, this is a design error; we should not have reached this point *
* Something is wrong here but the result is what we wanted *
* TODO: analyze why we have such a bad bail out here *

Similar to previous studies (Maldonado and Shihab, 2015), we also found that the most diffused SATD is design debt forming an average of 60% of the total SATD instances across all four projects analysed. Details of this is reported in our recent study (Mensah et al., 2016).

In order to validate the results from the text mining process with the manual exploratory process, we constructed a confusion matrix and reported the recall and precision measures. We report the recall and precision measures in Table 3 for each of theanalysed project with respect to the extracted five types of SATD (namely requirement debt, design debt, test debt, defect debt and documentation debt) as described in a study by Maldonado and Shihab (2015).

The mean Precision (P) and Recall (R) values of the confusion matrices created based on the text mining algorithm for the four open source projects are as follows: ArgoUML (P=*.824, R*=*0.818*), Chromium (P=*.815, R*=*0.847*), Eclipse (P=*.827, R*=*0.871*) and Apache (P=*.839, R*=*0.775*). On average, with the exception of recall (R) measure of the Apache project with a minimal value, the remaining projects had both recall and precision of not less than 80%. Thus, even though the text mining algorithm detected most of the SATD instances, there are still some instances which were falsely detected. Examples of SATD instances which were false positives and false negatives as reported by the text mining algorithm are as follows:

Table 4
Manual exploratory of SATD tasks within source code comment.

Project	Major		Minor	
	Frequency	Percent	Frequency	Percent
ArgoUML	21	33.3	42	66.7
Chromium	315	30.6	714	69.4
Eclipse	62	42.2	85	57.8
Apache	181	38.5	289	61.5

Examples of SATD instances which were false positives

* This should never have happened *
* Don't wait around; just abandon it *
* ### strictly speaking, this is a design error; we should not have reached this point *
* DESIGN ERROR: a mix of repositories *

Examples of SATD instances which were False Negatives

* TODO: this isn't quite right but is ok for next updates *
* FIXME: this code appears to be untested - and probably unused - and probably wrong *
* NOTE: This should never happen, but just be sure it works before production *
* FIXME: this code appears to be untested - and probably unused - and probably wrong *

From the Welch's *t-test* and Cliff's δ effect size between the manual and text mining process, we realized that there was no significant difference between the two processes with respect to each of the five types of SATD. Thus, at 5% asymptotic significance level, we found that even though there exist some variations between the manual and text mining process with respect to the types of SATD, the differences were not significant. This implies that, the text mining algorithm has a 95% certainty of accurately extracting the five types of SATD across the 4 projects analysed.

We present examples of extracted source code comments considered as *major* and *minor* tasks as follows:

Examples of extracted Major SATD tasks

* TODO: REMOVE ME BEFORE PRODUCTION (????) *
strictly speaking, this is a design error; we should not...
* Missing project window working set inconsistency *
* TODO: Hack to work around Bug *
* FIXME: windowRestoreState was a compile error *

Examples of extracted Minor SATD tasks

* Take care of the checkbox *
* Name is set to the empty string (yuck!) by default - fix it *
* FIXME: duplicated from the method with same name in ...*
* Remove duplicates and take care of ... *
* TODO: this isn't quite right but is ok for next updates *

Result in Table 4 presents the frequencies and respective percentages of the SATD tasks mined from the source code comments and categorized into *major* and *minor* tasks. We realized that most of the SATD considered as *major* and *minor* tasks were design debt and defect debt forming approximately 80% of the total SATD tasks extracted. The remaining SATD tasks were test debt, documentation debt and requirement debt forming approximately 20%. From our exploratory analysis, we realized that majority of SATD tasks assigned to developers to work on are minor tasks. Thus, approximately 62–70% of the SATD tasks were recorded in all four projects as *minor* tasks whilst approximately 30–38% of the SATD tasks were recorded as *major* tasks. Comparing results in Table 4, it can be seen that even though there were slight differences between the *major* and *minor* tasks in relation to the text mining and manual exploratory process, the differences were not significant. These insignificant differences were confirmed based on a robust Welch's

Table 5
Confusion matrix for extracted Major/Minor tasks.

Total number of major/minor instances for ArgoUML (n = 63)		Text mining Prediction		Precision (P) and Recall (R)
		Positive	Negative	
Manual exploration (actual)	Positive (*major*)	37	11	P = 86.0
	Negative (*minor*)	6	9	R = 77.1
Total number of major/minor instances for Chromium (n = 1029)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*major*)	352	104	P = 71.0
	Negative (*minor*)	144	429	R = 77.2
Total number of major/minor instances for Eclipse (n = 147)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*major*)	50	16	P = 80.6
	Negative (*minor*)	12	69	R = 75.8
Total number of major/minor instances for Apache (n = 470)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*major*)	180	68	P = 83.7
	Negative (*minor*)	35	187	R = 72.6

t-test statistic as recommended by Kitchenham et al. (2016) and Kitchenham (2015) with *p-values* of −.0869 and −.1128 for *major* and *minor* tasks respectively at a significance level of $\alpha = 0.05$. Thus, pairwise comparison using the robust Welch's *t-test* was made between the manual and text mining results with respect to the *major* and *minor* tasks respectively. The *p-values* more than .05 show that there are no significant differences between the manual and text mining process. We further estimated the magnitude of the differences between the manual and text mining results using the Cliff's δ effect size metric. The Cliff's δ effect size measures for the manual and text mining differences wrt *major* and *minor* tasks are $\delta = 0.1207$ and 0.2384 respectively. These δ values are very minimal showing small effect size and less significant differences between the manual and text mining exploration of the *major* and *minor* tasks respectively. In order to further validate the text mining algorithm in detecting the *major tasks* (regarded as positives) and *minor tasks* (regarded as negatives), we present the confusion matrix for the four analysed projects with respect to the *major* and *minor* extracted tasks in Table 5. We further provide the Precision (P) and Recall (R) values at the last column of Table 5. On average, a precision of 80.3% and recall of 75.7% were observed across the four analysed projects. Thus, from the classification analysis by the text mining algorithm, the average percentage of actual *major tasks* (true positives) retrieved from the corpus out of the total number of misclassifed *major tasks* (false positives) and the number of actual *major tasks* (true positives) is 80.3%. Similarly, with regards to recall, the average percentage of actual *major tasks* (true positives) retrieved from the corpus out of the total number of misclassifed *minor tasks* (false negatives) and the number of actual *major tasks* (true positives) is 75.7%.

In an attempt to address the gap analysis between task complexity and significance (step 2 of prioritization scheme), we examined the corpus of the source code comments and extracted SATD comments in relation to this phenomenon. We realized that most developers (38–60%) are faced with tasks (mostly *major* tasks) which they record as difficult and complex to address. We present examples of source code comments indicating complex tasks recorded by developers in the open source projects as follows:

Examples of extracted Complex SATD tasks
* Cannot introspect give up *
* We don't have anything configured, bail out *
* This is currently a nasty hack. We really cannot do this...*
* This test is a bit stupid. We cannot introspect *
* TODO: this is such a silly and nasty hack to do *

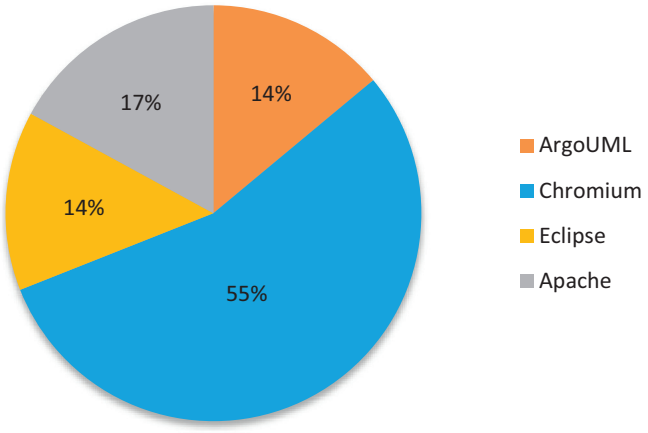


Fig. 3. Complex SATD tasks in open source projects.

The pie chart in Fig. 3 presents the distribution of the complex SATD tasks in the four open source projects. It can be seen that majority of these complex SATD tasks were found in Chromium OS project (55.0%). Apache has 17.0% of the complex SATD tasks and 14% found in ArgoUML and another 14% also found in Eclipse. We realized from the source code comments that about 4–12% of these complex tasks were left unresolved prior to release of the open source projects. Therefore, there is the need for these unresolved complex tasks to be given critical attention and addressed especially by more experienced developers (Potdar and Shihab, 2014) in the open source development community. It should be noted that based on the robust Welch's *t-test* statistic, there was no statistical significant difference between the manual exploration and text mining approach in the extraction of the complex SATD tasks. The statistical significant difference was confirmed by the Cliff's δ effect size.

4.2. RQ2: what is the difference between "expedited" and "expected" tasks in open source projects?

As explained in step 3 of the prioritization scheme (Section 3.1), in meeting certain development objectives, most developers sometimes take delight in using series of shortcuts (*expedited tasks*) to obtain workable and executable codes. We mined the source code comments and present examples of *expedited tasks* as well as *expected tasks* as follows:

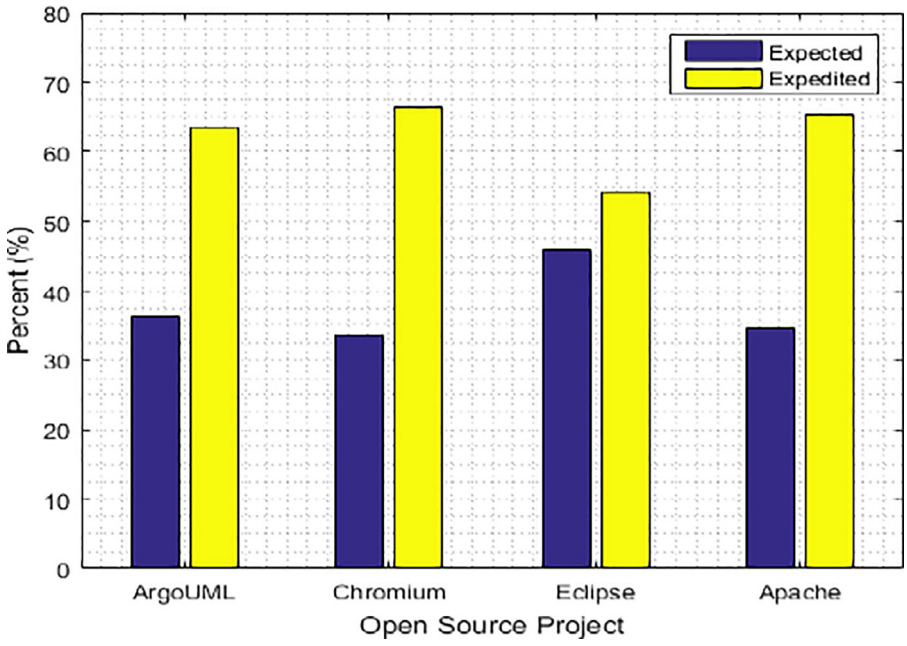


Fig. 4. Expected and expedited tasks in open source projects.

Examples of extracted Expedited SATD tasks
Yuck, this is awkward. All bcos we pass null in an overload
* Skip the stupid Microsoft UTF-8 Byte order marks *
* FIXME: pity this code isn't ...What is this needed for???? *
* Hmm, the simple match didn't work, probably ... *
* ... but definitely won't work on Windows *
* The workaround is to fake a query string match if ... *
Examples of extracted Expected SATD tasks
* Remove this hack in UmlFilePersister. It should be ... *
* Don't use this directly! Use instead ... *
* FIXME: Consider providing support for this. See 4,408,210 *
* Remove duplicates and take care of getOwner() *
* FIXME: this and find chase should be merged together ... *
* HACK!! This should be done using a separate renderer *

Fig. 4 illustrates the distribution of both the *expected* and *expedited tasks* mined from the source code comments of the four projects studied. Results indicate that majority of tasks undertaken by developers/contributors of the four open source software projects were e*xpedited tasks* with 63.6% in ArgoUML, 66.3% in Chromium, 54.2% in Eclipse and 65.5% in Apache. As a result of majority of these tasks being *expedited*, there is the need for the development team paying much attention to such tasks and using the appropriate expected approaches to rectify them in order to minimize logical failures andmalfunctioning of components. We further performed a statistical test using the Welch's *t-test* statistic as well as computing the Cliff's δ effect size between the manual and text mining methods for both the *expedited* and *expected* SATD tasks. Here, we found that, there was no statistical significant difference between the manual and text mining extraction processes.

Aside of the statistical test between the text mining and manual exploratory process, we provide evidence of the true positives, true negatives, false positives and false negatives of the text mining algorithm with respect to the extraction of the *expedited* and *expected tasks*. This is reported in the confusion matrix for the *expedited tasks* (regarded as postives) and *expected tasks* (regarded as negatives) in Table 6. We realized that, an average precision measure of 76.1% and average recall measure of 71.2% were recorded across the four analysed projects. Thus, from the classification analysis by the text mining algorithm, the average percentage of actual *expedited tasks* (true positives) retrieved from the corpus out of the total number of misclassifed *expedited tasks* (false positives) and the number of actual *expedited tasks* (true positives) is 76.1%. Similarly, with regards to recall, the average percentage of actual *expedited tasks* (true positives) retrieved from the corpus out of the total number of misclassifed *expected tasks* (false negatives) and the number of actual *expedited tasks* (true positives) is 71.2%.

4.3. RQ3: how can buggy prone (or "vital few") tasks be prioritized from a pool of SATD tasks in open source projects?

In the attempt to identify the plausible causes of SATD in the corpus of open source code comments, we first separated the *vital few* tasks (highly prone to bugs) from the *trivial many* tasks (less prone to bugs) in each open source project (Step 4 of prioritization scheme). We investigated the tasks within each project with respect to the five types of SATD (Maldonado and Shihab, 2015) and presented the distribution of buggy prone or defective SATD tasks corresponding to *vital few* tasks and less buggy prone tasks corresponding to *trivial many* tasks in Table 7. From the Pareto analysis, we realized that majority of the *vital few* tasks were more defective or buggy prone as compared to the *trivial many* tasks within each of the open source project. For example, in ArgoUML project, the buggy prone tasks (*vital few*) within the design and defect debts were 34.2% and 29.3% respectively (Table 7). Hence, design debts were highly prone to bugs within the ArgoUML project. In the Chromium project, we realized that design debts with a percentage of 47.8 were highly prone to software bugs. Similarly, in the Eclipse and Apache projects, design debts contributed to about 39% for *vital few* tasks. Thus, among all the five types of SATD tasks, design debts on average were highly prone to software bugs across the four open source projects analysed. In the text mining analysis differentiating between the *vital few* and *trivial many* tasks, we found some handful of overlaps that were recorded as both *vital few* and *trivial many* tasks. There were about a range of 5–33 instances of such overlaps in the four projects analysed. Results are presented in the last two columns of Table 7 and labeled as *overlap**.

We present the confusion matrix with their respective precision and recall measures for the extraction of *vital few* (positives) and *trivial many* (negatives) tasks in Table 8. We realized that an average precision of 81.05% and an average recall of 80.9%

Table 6
Confusion matrix for extracted Expedited/Expected tasks.

Total number of expedited/expected instances for ArgoUML (n = 33)		Text mining Prediction		Precision (*P*) and Recall (*R*)
		Positive	Negative	
Manual exploration (actual)	Positive (*expedited*)	12	7	*P* = *80.0*
	Negative (*expected*)	3	9	*R* = *63.2*
Total number of expedited/expected instances for Chromium (n = 51)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*expedited*)	17	4	*P* = *73.9*
	Negative (*expected*)	6	24	*R* = *81.0*
Total number of expedited/expected instances for Eclipse (n = 34)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*expedited*)	14	6	*P* = *82.4*
	Negative (*expected*)	3	11	*R* = *70.0*
Total number of expedited/expected instances for Apache (n = 57)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*expedited*)	19	8	*P* = *67.9*
	Negative (*expected*)	9	21	*R* = *70.4*

Table 7
Distribution of Buggy prone tasks within the types of SATD.

Types of SATD	Vital few tasks		Trivial many tasks		Overlap*	
ArgoUML Project	Frequency	Percent	Frequency	Percent	Frequency	Percent
Design	14	34.15	4	30.77	3	60.0
Defect	12	29.27	6	46.15	2	40.0
Test	8	19.51	2	15.38	0	0.0
Documentation	5	12.20	0	0.00	0	0.0
Requirement	2	4.88	1	7.69	0	0.0
Total	41	100	13	100	5	100
Chromium Project						
Design	184	47.67	59	29.80	15	45.45
Defect	97	25.13	62	31.31	6	18.18
Test	83	21.50	40	20.20	5	15.15
Documentation	8	2.07	27	13.64	2	6.06
Requirement	14	3.63	10	5.05	5	15.15
Total	386	100	198	100	33	100
Eclipse Project						
Design	29	39.19	18	36.73	2	22.22
Defect	14	18.92	9	18.37	4	44.44
Test	18	24.32	13	26.53	0	0.00
Documentation	4	5.41	4	8.16	1	11.11
Requirement	9	12.16	5	10.20	2	22.22
Total	74	100	49	100	9	100
Apache Project						
Design	92	39.83	29	27.88	3	14.29
Defect	91	39.39	42	40.38	6	28.57
Test	33	14.29	27	25.96	4	19.05
Documentation	12	5.19	2	1.92	3	14.29
Requirement	3	1.30	4	3.85	5	23.81
Total	231	100	104	100	21	100

**overlap between the vital few and trivial many tasks*

Table 8
Confusion matrix for extracted Vital few/Trivial many tasks.

Total number of vital few/trivial many instances for ArgoUML (n = 54)		Text mining Prediction		Precision (*P*) and Recall (*R*)
		Positive	Negative	
Manual exploration (actual)	Positive (*vital few*)	34	7	*P* = *87.2*
	Negative (*trivial many*)	5	8	*R* = *82.9*
Total number of vital few/trivial many instances for Chromium (n = 584)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*vital few*)	297	89	*P* = *74.1*
	Negative (*trivial many*)	104	94	*R* = *76.9*
Total number of vital few/trivial many instances for Eclipse (n = 123)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*vital few*)	61	13	*P* = *76.3*
	Negative (*trivial many*)	19	30	*R* = *82.4*
Total number of vital few/trivial many instances for Apache (n = 335)		**Text mining Prediction**		
		Positive	Negative	
Manual exploration (actual)	Positive (*vital few*)	188	43	*P* = *86.6*
	Negative (*trivial many*)	29	75	*R* = *81.4*

Table 9
Rework effort estimation of resolving *vital few* SATD.

SATD Class	Rework effort estimation			
	ArgoUML	Chromium	Eclipse	Apache
Requirement	0.5	1	3.1	2.6
Design	5.1	12.3	9.3	9.2
Test	3.1	3.5	4.8	7.3
Defect	1.2	2.9	2.6	4.1
Document	0	0.4	0.1	2
Total	9.9	20.1	19.9	25.2

Table 10
Rework effort estimation of resolving *trivial many* SATD.

SATD Class	Rework effort estimation			
	ArgoUML	Chromium	Eclipse	Apache
Requirement	1.9	3.1	2.4	1.3
Design	2.6	7.2	5.9	6.2
Test	2	4.2	3.7	7.5
Defect	1.1	2.3	3.4	5.2
Document	0.3	0	0.6	0.3
Total	7.9	16.8	16	20.5

were recorded by the text mining algorithm in the extraction of buggy prone SATD tasks from the corpus. Similar to *RQ1* and *RQ2,* we computed the Welch's *t-test* and Cliff's δ effect size for comparing the manual and text mining extraction of the *vital few* and *trivial many* tasks. For example, with regards to the *vital few* task extraction from the ArgoUML, the *p-value* for the Welch's *t-test* was 0.2137 ($>$0.05) and δ = 0.128 ($<$0.33) implying no significant difference with a low effect size (Kampenes et al., 2007) between the manual and text mining process. Thus, we found no statistical significant difference between the manual and text mining exploratory of the *vital few* tasks from ArgoUML. The minimal effect size value (Kampenes et al., 2007) confirms that the statistical significance is very small or insignificant. We found similar results for the remaining projects with regards to the *vital few* tasks as well as the *trivial many* tasks.

4.4. RQ4: what is the extent of rework effort required to resolve the buggy prone (or vital few) tasks in open source projects?

Lastly, we estimate the rework effort (measured in commented LOC per SATD source file of each system) involved in addressing the *vital few* tasks in Table 9. From the perspective of considering all the five types of debts, it was realized that design debt required substantial rework effort as elaborated in Table 9. Thus, the rework effort for resolving design debt in ArgoUML is 5.1, Chromium is 12.3, Eclipse is 9.3 and lastly, Apache is 9.2 commented LOC on average per SATD source file. Similarly, test and defect debts were also of key interest in this study which required rework aside of design debts. These two debts even though known by the development team that it will lead to long-term bugs upon release were left unfixed. This we believe will be due to the time-to-market constraint as mentioned by Fernandez-Sanchez et al. (2015). It can be seen that the rework effort estimation of approximately 10–25 commented LOC per SATD source file across the selected projects could affect the quality of the software projects analysed. We also provide the rework effort estimation of resolving the extracted *trivial many* tasks in Table 10. Here, we found that approximately 8–21 commented LOC per SATD source file is required to address the *trivial many* tasks. It was realized that, design debts required much rework effort as compared to the remaining four types of SATD across all the four projects analysed. This was observed in both cases of the *vital few* and *trivial many tasks.*

Table 11
Plausible causes of SATD from source code comments.

Cause of SATD	Frequency	Percent
Code smells (CS)	60	23.2
Too complicated and complex (TCC)	57	22.0
Inadequate code testing (ICT)	55	21.2
Unexpected code performance (UCP)	45	17.4
Heedless failure to remember (HFR)	21	8.1
Time constraint (TC)	12	4.6
Inconsistent communication among developers (ICT)	9	3.5

Thus, more rework effort is needed to address buggy prone SATD tasks regarded as *vital few* tasks as compared to less buggy prone tasks (*trivial many*). It is of key interest for software developers to spend much time on *vital few* tasks in order to improve the software quality prior to software release.

4.5. RQ5: what are the plausible causes of SATD tasks from the source code comments analyzed?

Based on the distribution of buggy prone SATD tasks in Table 7, we explored from the corpus of source code comments, plausible causes of SATD resulting in software bugs during the software development process (step 5 of prioritization scheme). Seven key plausible causes were identified from the source code comments and the results are presented in Table 11. It should be noted that these plausible causes are highly dominant in the *vital few* SATD tasks.

Examples of the source code comments in relation to the plausible causes of SATD are presented as follows:

Too complicated and complex (Examples)

* TODO: ... says – "This is a really nasty horrible hack" *
* In the rather unlikely case that we have no name, give up *
* FIXME: this is killing at least SSE editors see bug *
* Cannot introspect, give up *

Time constraint (Examples)
* Bail out if we are before the onset of daylight savings time *
* TODO this isn't quite right but is ok for now to go *
* Have we reached the limit to go ... taking so long to write *

Unexpected/Inconsistent code performance (Examples)
* Silly: cause UI inconsistency *
*An inconsistency has been detected when saving the model *
* Yuck: see ... inconsistency is found in cloned object *
* Doesn't work on your system (i.e. Linux) as expected *

Inconsistent communication among developers (Examples)
* FIXME: We do not know if the execution time of operation*
* FIXME: why override if nobody uses? *

Code smells (Examples)
FIXME: the following is loading the model b4 anything else
* Name is set to the empty string (yuck!) by default - fix it *
* This is a kludge... *

Heedless failure to remember (Examples)
* There are other things to take care of, so ... *
* FIXME: In the future, we want to take ... into account *
* This is a temporary solution, until better one is decided*

Inadequate testing (Examples)
* This test is a bit stupid ... *
* We hang our heads in shame. There are still bugs in ... *
* FIXME: this is not 64-bit clean, needs more testing *
* FIXME: this code appears to be untested properly *

Code smells (violated methods and classes resulting in serious problems in a project[4]), too complicated and complex tasks (TCC), inadequate code testing (ICT) and unexpected code performance (UCP) were prominent factors found during the exploratory anal-

[4] http://martinfowler.com/bliki/CodeSmell.html.

ysis of the source code comments. Results from Table 11 show that most of the plausible causes of SATD within the projects under study are code smells (23.2%), complicated and complex tasks (22.0%), inadequate code testing (21.2%) and unexpected code performance (17.4%). These causes have to be given much attention during development before reaching a break-point for time-to-market release (step 6 of prioritization scheme).

5. Threats to validity

This study is constraint of a large number of open source projects for generalization of results obtained. The four selected open source projects might not be a general representative sample of the total population of all open source projects, thus the findings of this study cannot be confidently generalized. The selected projects used are popular and large in size. Therefore, the examination of all the developers' source code comments from the corpus of these projects with the intention of resolving the technical debt problem can form a good foundation for researchers to conduct more in-depth studies in this field. The list of Self-admitted technical debt (SATD) textual indicators identified from this study might not be a generalized representation for all SATD tasks in the software development and maintenance environment. However, we believe that by even identifying a subset of potential SATD tasks in source code comments and their subsequent management will lead to substantial improvement in released software projects. Our study employed a manual exploratory process to extract the textual indicators for the various categories of SATD prior to subjecting the extracted indicators to the text mining algorithm. Manual extraction of textual indicators can result in biasness which can affect the generalization of results from this study. Nothwithstanding, authors of this study engaged in an open discussion to address all conflicts that resulted in reading through the corpus of source code comments. Since our work focused on source code comment analysis, we were constraint of gathering more information especially from the industry to justify the sixth step of the prioritization scheme (break-point for release decision). Lastly, this study focused on well-commented open source projects which might not be confidently generalized on normally commented projects or less commented projects.

6. Conclusion

6.1. Summary of findings

In this study, we introduce a 6-step prioritization scheme to assist in the extraction of different classes of Self-admitted Technical Debt (SATD) and estimating the rework effort for buggy prone SATD (*vital few*) tasks. We implement a text mining algorithm to facilitate the analysis of source code comments of four large open source projects – ArgoUML, Chromium OS, Eclipse and Apache HTTP Server.

Results from the exploratory analysis show that, out of 15.5–31.5% of the source code comments forming instances of SATD, approximately 31–39% are *major tasks* and 58–69% are *minor tasks*. Most of these *major tasks* are complex and difficult to handle by developers prior to software release. We mine the corpus of source code comments and identify seven plausible causes of SATD tasks. The four prominent causes of SATD tasks are code smells (23.2%), complicated and complex tasks (22.0%), inadequate code testing (21.2%) and unexpected code performance (17.4%). These prominent factors are highly dominant among the *vital few* tasks (highly prioritized or buggy prone tasks). Our findings show that the rework effort needed to address the *vital few* tasks in the selected open source projects is approximately 10–25 commented LOC per SATD source file.

The 6-step prioritization scheme will aid in decision making prior to software release in an attempt to minimize SATD related tasks that would otherwise result in higher maintenance overheads. It is a novel and proven technique which can further be extended and exercised in software engineering.

6.2. Future direction

We intend to conduct an industrial case study to further validate the results obtained from the mined open source projects. More work can be done on the break-point for release of software projects to enhance the decision making process. We intend to extend the proposed prioritization scheme on source code comments of closed systems. Again, we intend to extend the prioritization scheme to resolve the debt metaphor in other software engineering related fields. Lastly, more work can be done on the need for resolving SATD tasks to software bugs in order to minimize maintenance overheads.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017), and the research funds of City University of Hong Kong (No. 7004683 and 7004474).

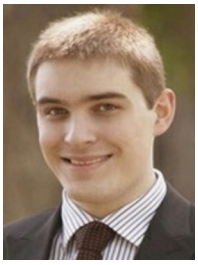
References

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250.
Baena-García, M., Carmona-Cejudo, J.M., Castillo, G., Morales-Bueno, R., 2011. TF-SIDF: Term frequency, sketched inverse document frequency. In: Int. Conf. Intell. Syst. Des. Appl. ISDA, pp. 1044–1049.
Bavota, G., Russo, B., 2016. A large-scale empirical study on self-admitted technical debt. In: Proc. 13th Int. Work. Min. Softw. Repos. - MSR '16, pp. 315–326.
Can, F., Kocberber, S., Balcik, E., Kaynak, C., Ocalan, H.C., Vursavas, O.M., 2008. Information retrieval on Turkish texts. J. Am. Soc. Inf. Sci. Technol. 59 (3), 407–421.
Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. 114 (3), 494.
Collard, M.L., Decker, M.J., Maletic, J.I., 2011. Lightweight transformation and fact extraction with the srcML toolkit. In: Proc. - 11th IEEE Int. Work. Conf. Source Code Anal. Manip. SCAM 2011, pp. 173–184.
Corbin, J., Strauss, A., 1990. Grounded theory research: procedures, canona and evaluative criteria. Zeitschrift fur Sociol 19 (6), 418–427.
Cunningham, W., 1993. The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger 4 (2), 29–30.
De los Santos, P.A., Burke, R.J., Tien, J.M., 2007. Progressive random sampling with stratification. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. 37 (6), 1223–1230.
Devroey, X., Cordy, M., Perrouin, G., Schobbens, P.-Y., Legay, A., Heymans, P., 2014. Towards Statistical Prioritization for Software Product Lines Testing. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, Vol. 14.
Farias, M.A.D.F., Neto, M.G.D.M., Da Silva, A.B., Spinola, R.O., 2015. A contextualized vocabulary model for identifying technical debt on code comments. In: 2015 IEEE 7th Int. Work. Manag. Tech. Debt, MTD 2015 - Proc., ii, pp. 25–32.
Fenton, N.E., Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE Trans. Softw. Eng. 26 (2000), 797–814 *Pp.*, vol. 797, no. 8.
Fernandez-Sanchez, C., Garbajosa, J., Yague, A., 2015. A framework to aid in decision making for technical debt management. In: 2015 IEEE 7th Int. Work. Manag. Tech. Debt, MTD 2015 - Proc., pp. 69–76.
Fluri, B., Wursch, M., Gall, H.C., 2007. Do code and comments co-evolve? On the relation between source code and comment changes. In: Proc. - Work. Conf. Reverse Eng. WCRE, pp. 70–79.
Fu, Z., Wu, X., Guan, C., Sun, X., 2016. Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. IEEE Trans 11 (12), 2706–2716.
Kamei, Y., Shihab, E., 2016. Defect prediction: accomplishments and future challenges. In: 2016 IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering, 1, pp. 33–45.
Kampenes, V.B., Dyba, T., Hannay, J.E., Sjoberg, D.I.K., 2007. A systematic review of effect size in software engineering experiments. Inf. Softw. Technol. 49 (11–12), 1073–1086.
Kitchenham, B., 2015. Robust statistical methods. In: Proc. 19th Int. Conf. Eval. Assess. Softw. Eng. - EASE '15, pp. 1–6.

Kitchenham, B., et al., 2016. Robust statistical methods for empirical software engineering. Empir. Softw. Eng. 21 (1), 212–259.
Kotsiantis, S.B., Kanellopoulos, D., Pintelas, P.E., 2006. Data preprocessing for supervised learning. Int. J. Comput. Sci. 1 (2), 111–117.
Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: from metaphor to theory and practice. IEEE Softw. 29 (6), 18–21.
Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220.
Maldonado, E.D.S., Shihab, E., 2015. Detecting and quantifying different types of self-admitted technical Debt. In: 2015 IEEE 7th Int. Work. Manag. Tech. Debt, MTD 2015 - Proc., pp. 9–15.
Manning, R., Prabhakar, C.D., Schütze, H., 2008. Introduction to Information Retrieval, 1. Cambridge University Press, Cambridge.
Martini, A., Bosch, J., Chaudron, M., 2014. Architecture technical debt: understanding causes and a qualitative model. In: Proc. - 40th Euromicro Conf. Ser. Softw. Eng. Adv. Appl. SEAA 2014, pp. 85–92.
Mensah, S., Keung, J., Bosu, M.F., Bennin, K.E., 2016. Rework effort estimation of self-admitted technical debt. In: Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ) and 1st International Workshop on Technical Debt Analytics (TDA) co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC), pp. 72–75.
Nakatsu, R.T., Grossman, E.B., Iacovou, C.L., 2014. A taxonomy of crowdsourcing based on task complexity. J. Inf. Sci. 40 (6), 823–834.
Potdar, A., Shihab, E., 2014. An exploratory study on self-admitted technical debt. In: Proc. - 30th Int. Conf. Softw. Maint. Evol. ICSME 2014, pp. 91–100.
Ramasubbu, N., 2013. Managing technical debt in enterprisesoftware packages. IEEE Trans. Softw. Eng. 40 (8), 1–15.
Tang, X., Wang, S., Mao, K., 2015. Will this bug-fixing change break regression testing? In: Int. Symp. Empir. Softw. Eng. Meas., 2015–Novem, pp. 215–224.
Team, R.C., 2016. R: A language and Environment for Statistical Computing. R Foundation For Statistical Computing, Vienna, Austria.
Wehaibi, S., Shihab, E., Guerrouj, L., 2016. Examining the impact of self-admitted technical debt on software quality. In: 2016 IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering, 1, pp. 179–188.
Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.
Ye, X., Bunescu, R., Liu, C., 2014. Learning to rank relevant files for bug reports using domain knowledge. In: Proc. 22Nd ACM SIGSOFT Int. Symp. Found. Softw. Eng., no. April, pp. 689–699.
Zazworka, N., Shaw, Ma., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: Work. Manag. Tech. Debt, pp. 17–23.
Zhang, Y., Jin, R., Zhou, Z.H., 2010. Understanding bag-of-words model: a statistical framework. Int. J. Mach. Learn. Cybern. 1 (1–4), 43–52.
Zhao, F., Tang, Y., Yang, Y., Lu, H., Zhou, Y., Xu, B., 2015. Is learning-to-rank cost–effective in recommending relevant files for bug localization? In: Proc. - 2015 IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS 2015, pp. 298–303.
Zimmermann, T., Nagappan, N., Zeller, A., 2008. Predicting bugs from history. Softw. Evol. 69–88.

Solomon Mensah received his B.Sc. degree in Computer Science and Statistics, and MEng in Computer Engineering from the University of Ghana in 2011 and 2014, respectively. He is a Ph.D. candidate at the Department of Computer Science, City University of Hong Kong under the supervision of Dr. Jacky Keung. His research interests include software effort estimation, technical debt, text mining and deep learning. He has experience in the application and development of statistical methods for Mathematical and prediction modelling. One of his works received the best paper award in the 2017 International Conference on Software Quality, Reliability & Security.

Jacky Keung received his B.Sc. (Hons) in Computer Science from the University of Sydney, and his Ph.D. in Software Engineering from the University of New South Wales, Australia. He is Assistant Professor in the Department of Computer Science, City University of Hong Kong. His main research area is in software effort and cost estimation, empirical modeling and evaluation of complex systems, and intensive data mining for software engineering datasets. He has published papers in prestigious journals including IEEE-TSE, IEEE-SOFTWARE, EMSE and many other leading journals and conferences.



Jeffrey Svajlenko is a Ph.D. candidate in the Department of Computer Science at University of Saskatchewan, Canada under the supervision of Dr. Chanchal Roy. He received his B.Sc. (Hons) degree in Computer Science and B.Sc. in Engineering Physics from the University of Saskatchewan in 2011. His research interests include code clones, clone detection and benchmarking.

Kwabena Ebo Bennin received his B.Sc. (Hons) degree in Computer Science and Statistics from the University of Ghana, in 2011. He is an HKPFS fellow and is working toward a Ph.D. degree in the Department of Computer Science, City University of Hong Kong. He is under the supervision of Dr. Jacky Keung. He was a visiting researcher with Okayama University, Japan working under the supervision of Professor Akito Monden. His research interests include improving the fundamentals and performances of software defect prediction models, software effort and cost estimation, data mining for software engineering datasets and techniques for bug localization.

Qing Mi is a Ph.D. candidate in the Department of Computer Science, City University of Hong Kong under the supervision of Dr. Jacky Keung. Prior to enrolling in CityU, she worked for two years as a Client Software Development Engineer in Tencent, Beijing, China. She received her M.Sc. in Computer Science and Technology from the Beijing Institute of Technology, Beijing, China in 2012, and her B.Sc. in Network Engineering from the Hebei University, Baoding, China in 2009. Her research interests primarily concentrate on code readability, data mining and analytics, deep learning and empirical experiments.