Detecting Technical Debt through Issue Trackers

Ke Dai and Philippe Kruchten
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
kedai, pbk@ece.ubc.ca

Abstract—**Managing technical debt effectively to prevent it from accumulating too quickly is of great concern to software stakeholders. To pay off technical debt regularly, software developers must be conscious of the existence of technical debt items. The first step is to make technical debt explicit; that is the identification of technical debt. Although there exist many kinds of static source code analysis tools to identify code-level technical debt, identifying non-code-level technical debt is very challenging and needs deep exploration. This paper proposed an approach to identifying non-code-level technical debt through issue tracking data sets using natural language processing and machine learning techniques and validated the feasibility and performance of this approach using an issue tracking data set recorded in Chinese from a commercial software project. We found that there are actually some common words that can be used as indicators of technical debt. Based on these key words, we achieved the precision of 0.72 and the recall of 0.81 for identifying technical debt items using machine learning techniques respectively.**

Keywords—technical debt; identification; issue tracking data sets; natural language processing; machine learning

I. Introduction

Technical debt refers to delayed tasks and immature artifacts that constitute a "debt" because they incur extra costs in the future in the form of increased cost of change during evolution and maintenance [1]. An appropriate amount of technical debt would accelerate the process of software development; however, too much of it would impede the progress and even abort the project [2]. Typically, some startup software companies tend to incur technical debt strategically to speed up the development at the early stage of development process for the purpose of capturing the market. But with the growth of the size and complexity of the software, it may become increasingly more difficult to maintain and evolve the product due to intertwined dependencies between modules or components without paying off technical debt regularly. As a result, software stakeholders need to pay off technical debt regularly to prevent it from accumulating too quickly. Different from bugs or defects existing in a software system, technical debt is invisible as the software often works well from users' perspective and even developers are often unconscious of the existence of technical debt. The invisibility of technical debt increases the risks of rigid software design and huge maintenance cost in the future significantly. Therefore, it is essentially critical for development teams to be able to identify technical debt items existing in the current software system at any point in time as it is the prerequisite to conduct other management activities of technical debt including measurement of technical debt, estimation of effort to be expended, payment of technical debt, risk evaluation, etc. Once technical debt can be identified systematically, software development teams would be able to estimate future budget, prioritize future tasks, allocate limited resources and evaluate potential risks. They could also make informed decisions about when technical debt should be paid off to maximize their profits.

Due to the importance of identification of technical debt, a number of studies empirically explored various approaches to detecting technical debt. Some of these researches focused on employing source code analysis techniques to detect technical debt. Code smells and automatic static analysis (ASA) are two most-used source code analysis techniques for the identification of technical debt. Code smells was first introduced by Fowler et al. to describe the violation of object-oriented design principles (e.g., abstract, encapsulation and inheritance) [3], whereas ASA techniques aim at identifying violations of recommended programming practices that might degrade some of software quality attributes (e.g., maintainability, efficiency).

Other studies aimed to identify technical debt of large granularity that's undetectable by source code analysis techniques, such as architecture and requirement technical debt [10] [11] [12] [13]. Compared to code-level technical debt, the identification of non-code-level technical debt is not studied sufficiently and the approaches are limited. To our knowledge, none of existing approaches can identify all types of technical debt.

As a complement to existing approaches, we try to identify non-code-level technical debt through issue trackers. We hope to acquire developers' points of view on technical debt and understand how they communicate technical debt in issue trackers since they use issue trackers to record, track, prioritize various kinds of issues in software projects. Further, developers' standpoints of technical debt will in turn help refine our understanding of technical debt and should be taken into consideration for an improved definition of technical debt.

However, it is difficult and impractical to identify technical debt manually through issue trackers due to substantial effort involved, especially when a large project comprises a large number of issues. In this context, we exploited natural language processing (NLP) and machine learning techniques to automate the process. NLP techniques were applied to extract features from unstructured text data and machine learning techniques were used to decide whether a certain issue is an instance of technical debt or not. We performed an exploratory study on a commercial software project to validate the efficacy of our

Partially sponsored by a Canada NSERC grant, a MITACS grant and the Institute for Computing, Information and Cognitive System (ICICS) at UBC

approach to the identification of technical debt through issue trackers. Experimental results demonstrate that our approach is effective in identifying non-code-level technical debt, especially requirement debt, design debt, and UI debt, which cannot be detected by source code analysis techniques.

We address the following questions through this research:

- RQ1: How do software practitioners communicate technical debt issues in issue trackers?
- RQ2: Are there text patterns that are an indication that technical debt exist which can be used to identify potential technical debt using NLP and machine learning techniques automatically?

The rest of this paper is organized as follows: Section II discusses related work. Section III describes our approach. Section IV reports and analyzes experimental results of our exploratory study. Section V presents the threats to validity. Finally, we conclude our research and envision future work in Section VI.

II. Related Work

A. *Identification of Technical Debt*

Many researches have been done to identify code-level technical debt. This kind of technical debt can be detected using static program analysis tools based on the measurement of various source code metrics. Marinescu proposed metric-based detection strategies to help engineers directly localize classes or methods affected by the violation of object-oriented design principles and validated the approach on multiple large industrial case studies [4]. Munro et al. refined Marinescu's detection strategies by introducing some new metrics and justification for choosing the metrics and evaluated the performance of the approach in identifying two kinds of code smells (lazy class and temporary field) in two case studies [5]. Olbrich et al. investigated the relationship between two kinds of code smells (god class and shotgun surgery) and maintenance cost by analyzing the historical data of two major open source projects, Apache Lucene and Apache Xerces 2 J [6]. Wong et al. proposed a strategy to detect modularity violations and evaluated the approach using Hadoop and Eclipse [7]. Besides, some researchers explored identifying technical debt through comments in source code [8] [9].

Other researches aimed at exploring approaches to identifying other types of technical debt such as architecture technical debt. Brondum et al. proposed a modelling approach to visualizing architecture technical debt based on analysis of the structural code [10]. Li et al. proposed to use two modularity metrics, Index of Package Changing Impact(ICPI) and Index of Package Goal Focus(IPGF), as indicators of architecture technical debt [11]. Further they proposed an architecture technical debt identification approach based on architecture decisions and change scenarios [12]. The work closest to ours is the work by Bellomo et al., where manual examination was conducted on 1,264 issues in four issue trackers from open source and government projects and 109 examples of technical debt were identified using a categorization method they developed [13]. The major difference is that we partially automated the process of identification while they identified technical debt items manually. To our knowledge, our study is the first one that applies NLP and machine learning techniques to detect technical debt through issue trackers.

B. *Mining Issue Tracking Databases*

Issue tracking systems are widely used in open source projects as well as in software industry to record, triage and track different kinds of issues occurred during the lifecycle of software: bugs finding, defects fixing, adding new features, future tasks, requirements updating, etc. They play an important role in facilitating software development teams to manage development and maintenance activities and thus promoting the success of software projects. Some researches have focused on mining issue tracking databases to retrieve valuable information for improved definition, development management, quality evaluation, predictive models, etc.

Antoniol et al. applied NLP and machine learning techniques (alternating decision trees, naïve Bayes classifier, and logistic regression) to automate the process of distinguishing bugs from other kinds of issues, compared the performance of this approach with that of using regular expression matching and concluded machine learning techniques outperforms regular expression matching in terms of predictive accuracy [14].

Runeson et al. developed a prototype tool which detects duplicate defect reports in issue tracking systems using NLP techniques, evaluated the identification capabilities of this approach in a case study and concluded that about 2/3 of the duplicates can possibly be found using this approach [15]. Wang et al., Jalbert and Weimer, Sun et al., Sureka and Jalote performed similar research to address the same problem [16] [17] [18] [19].

Other work focused on concerned aspects of software quality attributes, say security. Cois et al. proposed an approach to detecting security-related text snippets in issue tracking systems using NLP and machine learning techniques [20].

III. Method

For this research, we cooperated with a local software company to access the issue data set of a commercial software product which have been in development for more than two years rather than just using issue data sets from open source software projects in order to make the classifier we developed more adaptable to the style of issue data from commercial software products. The issues are recorded mainly in Mandarin with a few English words as the developers of this product are Chinese.

A. ***Phase 0: Exporting issue data***

We first exported the issue data set and saved it in a spreadsheet which makes it easier for researchers to read the issues and to process the data. The fields or variables of the data set we used are id, type, priority, state, summary, description and label. We also tuned the character coding format so that Chinese characters can be displayed normally and removed issues with messy code to render the data set clean and tidy. Finally, we got 8,149 issues in total. Figure 1 shows an overview of our approach.

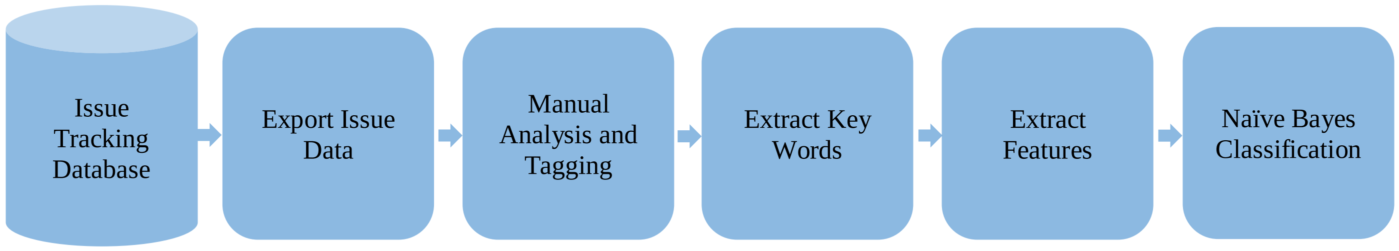

Fig. 1. Approach Overview

TABLE I. THE CLASSIFICATION CRITERIA OF ISSUES

Label	Subtype	Description
Not Technical Debt	Requirement Change	The request for requirement change from the client
	New Features	Tasks to add new functions or introduce new features
	Insufficient Description	The description is insufficient to make a decision
	Critical Defects	Critical functions or features are not implemented correctly
Technical Debt	Defect Debt	Temporarily tolerable defects that will be fixed in the future
	Requirement Debt	Requirements are not implemented accurately or implemented partially
	Design Debt	The violation of good object-oriented design principles such as god class and long method
	Code Debt	Bad coding practices such as dead code or no proper comments
	UI Debt	UI related issues such as inconsistent UI style or ugly UI elements
	Architecture Debt	Design limitation in architecture level such as the violation of modularity

B. *Phase 1: Tagging issues manually*

We tagged each issue or task in the issue data set as technical debt or not technical debt manually by reading the summary and description based on the following classification criteria:

1. Is it a request for requirement change from the client?

If yes, we definitely tag this issue as not technical debt.

2. Is it a task to add new functions or introduce new features to the product?

If yes, we also definitely tag this issue as not technical debt.

3. The description of the issue is too short or insufficient to decide whether the issue is a technical debt item.

In this case, we tag this issue as not technical debt.

4. Is it a defect that important and critical functions or features are not implemented correctly?

If yes, we tag this issue as not technical debt.

5. Is it a defect that is not critical from the client's perspective but weakens the performance and capabilities of the system and will be fixed in the future?

If yes, we tag this issue as defect debt.

6. Is it a task to redesign some function or feature as current design does not meet or meet the requirement partially?

If yes, we tag this issue as requirement debt.

7. Is it a limitation of design that may pose a threat to the performance of the system or to the evolution and maintenance of the system?

If yes, we tag this issue as design debt.

8. Is it an issue related to bad coding practices such as dead code and no proper comments?

If yes, we tag this issue as code debt.

9. Is it a UI related issue such as inconsistent UI style or ugly UI elements that degrades user experience?

If yes, we tag this issue as UI debt.

10. Is it a limitation of design in architecture level that may exert a negative impact on the performance of the system or on the evolution and maintenance of the system such as the violation of modularity?

If yes, we tag this issue as architecture debt.

The 10 cases listed above are the typical cases we encountered when tagging the issues but do not cover all the types of issues existing in the issue tracker. Actually, some issues can be tagged as either technical debt or not technical debt, which, to a large extent, depends on your personal understanding of technical debt. Typically, there exist wide discrepancies regarding whether defects should be viewed as a type of technical debt among researchers and developers. In this study, we divided defects into two categories: 1. critical defects that may cause fatal errors occurring when using the software; 2. tolerable defects that may exert a marginal negative impact on the use of the software and are not fixed immediately after being detected. We tagged the first type of defects as not technical debt and the second type of defects as technical debt.

After we finished tagging all the issues, we asked a known expert in software engineering and technical debt external to our research team to validate the results of our manual classification. The expert classified a random subset of the issues independently. With respect to discrepancies in the classification of some issues, we exchanged our respective points of view about why we classified a certain issue as the category to solve our discrepancies. If we did not achieve agreements in the classification of a certain issue, we discussed the issue with developers to gain insight into the issue itself and their opinions on the classification.

Finally, we found 331 technical debt issues in total whose distribution is shown in Table II. Requirement debt and design debt are the main technical debt types, including 105 and 141 instances respectively.

TABLE II. THE NUMBER OF DIFFERENT TYPES OF TECHNICAL DEBT ISSUES

Technical Debt Type	Number
Requirement Debt	105
Architecture Debt	6
Design Debt	141
Defect Debt	15
UI Debt	35
Code Debt	20
other	9

C. *Phase 2: Extracting key words and phrases*

Different from English, Chinese is written without spaces between words. So before extracting key words from the Chinese texts, we have to convert each text to a word sequence using a Chinese text segmentation tool. For this research, we used Jieba (https://github.com/fxsjy/jieba/) [21], a popular open source Chinese text segmentation tool, to split Chinese texts into a sequence of words.

After conducting Chinese text segmentation, we extracted key words using Jieba. Jieba integrated two key word extraction algorithms: TF-IDF and TextRank. We used both of them to extract key words for detecting technical debt. We took the union of two sets of key words extracted using these two different algorithms, removed the key words referring to domain knowledge from the union set, and finally added some key words based on our intuition. To make this paper more readable, we only list the meaning of key words instead of original Chinese characters in the below:

"at present", "now", "current", "previously", "in the past", "in the future", "time", "actually", "in reality", "users", "clients", "strengthen", "change", "modify", "replace", "update", "delete", "cancel", "suggest", "optimize", "simplify", "perfect", "improve", "refactor", "decouple", "again", "re-", "replant", "tidy", "integrate", "merge", "adjust", "extend", "expect", "plan", "management", "maintenance", "function", "requirement", "design", "rule", "theory", "strategy", "mechanism", "algorithm", "data structure", "logic", "code", "structure", "architecture", "style", "format", "performance", "efficiency", "sufficiency", "security", "compatibility", "scalability", "maintainability", "stability", "generality", "usability", "readability", "real-time", "limitation", "more friendly", "more specialized", "more accurate", "problem", "configuration", "priority", "inconsistent", "unreasonable", "inconvenient", "convenient", "not clear", "inaccurate", 'not intuitive', "not pretty", "incongruous", "not smooth", "inconformity", "incomplete", "abnormity", "defect", "impact", "experience", "habit", "operation", "difficulty", "delay", "UI", "risk", "optimize", "refactor", "SonarQube"

There are 114 key words in total, among which 104 words are Chinese words and 10 words are English words. As some words express the similar or same meaning, we merged these words. All these words to some extent indicate or imply the concept of technical debt from different perspectives. To be specific,

- "at present", "now", "current", "previously", "in the past", "in the future", "time"

These words indicating time concept may imply accumulation.

- "strengthen", "change", "modify", "replace", "update", "delete", "cancel", "optimize", "simplify", "perfect", "improve", "refactor", "decouple", "again", "re-", "replant", "tidy", "integrate", "merge", "adjust", "extend"

These words indicate the modification of code, design or architecture, or the enhancement of functionality, capability, performance, efficiency, etc.

- "security", "compatibility", "scalability", "maintainability", "stability", "generality", "usability", "readability", "real-time", "limitation"

These words indicate concerned aspects of software quality attributes.

- "inconsistent", "unreasonable", "inconvenient", "convenient", "unclear", "inaccurate", 'not intuitive', "not pretty", "incongruous", "not smooth", "inconformity", "incomplete", "abnormity", "defect", "limit", "impact", "experience", "habit", "operation", "difficulty", "delay"

These words indicate defects or design limitation such as inconsistent UI style, unreasonable design, etc.

D. *Phase 3: Extracting features*

Once key words were extracted from the issue data set, features for text classification can be derived by checking the presence or absence of each key word in each issue text. Given the set of key words is [“users”, “change”, “modify”, “improve”, “refactor”, “decouple”, “priority”, “button”, “architecture”, “deploy”, “rules”], consider this issue description: “design change: to keep a consistent design with different pages, we are moving the clear-all-rules button to the front of the deploy rules table. (Consistent with event page)”. First, we tokenized the text into a sequence of words and removed stop words (words that are too common to indicate any semantic meaning for our classification). Thus, the text is converted to a string list: [“design”, “change”, “keep”, “consistent”, “design”, “different”, “pages”, “moving”, “clear-all-rules”, “button”, “front”, “deploy”, “rules”, “table”]. Then we could check whether this string list contains each of key words, i.e. [contain(“users”), contain(“change”), contain(“modify”), contain(“improve”), contain(“refactor”), contain(“decouple”), contain(“priority”), contain(“button”), contain(“architecture”), contain(“deploy”), contain(“rules”)]. This vector checking the presence or absence of each key word is called feature space. The dimension of feature space depends on the size of the set of key words. Finally, we got the feature vector of the issue sample based on the feature space: [false, true, false, false, false, false, false, true, false, true, true].

The feature space actually not only includes unigram features that are a single word like “design”, “decouple”, but also has bigram and trigram features which comprised adjacent word pair and triplet respectively, such as “design change” and “improve unit test”; that is to say, the feature space in the previous example can be extended to [contain(“users”), contain(“change”), contain(“modify”), contain(“improve”), contain(“refactor”), contain(“decouple”), contain(“priority”), contain(“button”), contain(“architecture”), contain(“deploy”), contain(“rules”), contain(“design change”), contain(“improve unit test”)]. Then the feature vector is turned into [false, true, false, false, false, false, false, true, false, true, true, true, false]. Figure 2 shows the process of feature extraction.

E. *Phase 4: Creating a binary Naïve Bayes classifier*

Naïve Bayes is a simple classification algorithm that is based on an assumption that the features are conditionally independent of each other given the category. It determines the category of a given sample with n-dimensional features ($x_1, \dots, x_n$) by calculating the probability that the sample belongs to each category and then assigning the most probable category c to it, which can be described as:

$$c = \arg \max_{k\in\{1,\dots,K\}} p(c_k \mid x_1, \dots, x_n),$$

where c_k is the kth category, and K is the size of the set of categories. Using Bayes’ theorem, the conditional probability $p(c_k \mid x_1, \dots, x_n)$ can be decomposed as:

$$p(c_k \mid x_1, \dots, x_n) = \frac{p(x_1,\dots,x_n \mid c_k)}{\sum_{h=1}^{K} p(x_1,\dots,x_n \mid c_h)}\, p(c_k).$$

With the conditional independence assumptions, the conditional probability $p(c_k \mid x_1, \dots, x_n)$ can be transformed into:

$$p(c_k \mid x_1, \dots, x_n) = \frac{\prod_{j=1}^{n} p(x_j|c_k)}{\sum_{h=1}^{K}\prod_{j=1}^{n} p(x_j|c_h)p(c_h)}\, p(c_k).$$

To perform our experiments, we used a popular natural language toolkit for building Python programs to process human language data (NLTK http://www.nltk.org) [22]. We employed the implementations by NLTK instead of creating a binary Naïve Bayes classifier from scratch.

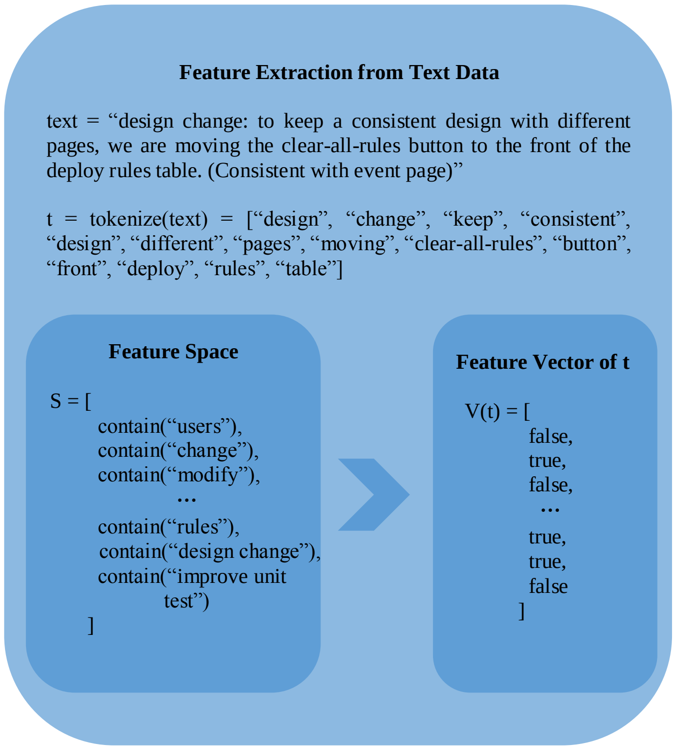
Feature Extraction from Text Data
text = “design change: to keep a consistent design with different pages, we are moving the clear-all-rules button to the front of the deploy rules table. (Consistent with event page)”
t = tokenize(text) = [“design”, “change”, “keep”, “consistent”, “design”, “different”, “pages”, “moving”, “clear-all-rules”, “button”, “front”, “deploy”, “rules”, “table”]
Feature Space
S = [contain(“users”), contain(“change”), contain(“modify”), … contain(“rules”), contain(“design change”), contain(“improve unit test”)]
Feature Vector of t
V(t) = [false, true, false, … true, true, false]

Fig. 2. Feature Extraction

IV. Experimental Results and Analysis

Repeated random sub-sampling validation was performed to validate our approach to the identification of technical debt by repeatedly splitting the full data set into 80/20% randomly distributed partitions, training and testing the classifier for each split, and recording performance results.

RQ1: How do software practitioners communicate technical debt issues in issue trackers?

We searched for the term “technical debt” and the corresponding Chinese term in the issue data set and found no positive results. All the technical debt instances in this issue tracker were implicitly expressed using other technical debt related words such as redesign, design change, refactor, cleanup, decouple, etc. By means of communication with developers of this product, we learned that they did not have strong awareness of technical debt. Some of them had even never heard about the concept of technical debt although they recognized that they had much experience in incurring technical debt when we explained what is technical debt. To track, prioritize and pay off technical debt effectively, we suggested they take technical debt as an issue type in the issue tracker to communicate technical debt explicitly.

TABLE III. 20 MOST INFORMATIVE FEATURES FOR DETECTING TECHNICAL DEBT

20 Most Informative Features for Detecting Technical Debt	
Features	**Likelihood Ratio (Technical Debt : not Technical Debt)**
协议识别优化(protocol identification optimization) = 1	155.2 : 1.0
增强 (strengthen) = 1	128.2 : 1.0
不方便 (inconvenient) = 1	128.2 : 1.0
提高 (improve) = 1	117.4 : 1.0
优化 (optimize) = 1	90.8 : 1.0
整改 (change or modify) = 1	87.7 : 1.0
风格 (style) = 1	65.2 : 1.0
体验 (experience) = 1	64.4 : 1.0
改进 (improve) = 1	60.7 : 1.0
不容易 (not easy) = 1	47.2 : 1.0
改善 (improve) = 1	44.5 : 1.0
效率 (efficiency) = 1	44.5 : 1.0
简化(simplify) = 1	38.2 : 1.0
解决方案(strategy) = 1	35.8 : 1.0
困难(difficulty) = 1	33.7 : 1.0
前期(previously) = 1	33.7 : 1.0
不美观(not pretty) = 1	33.7 : 1.0
risk = 1	33.7 : 1.0
算法(algorithm) = 1	31.8 : 1.0
习惯(habit) = 1	31.8 : 1.0

TABLE IV. THE RESULT FOR REPEATED RANDOM SUB-SAMPLING VALIDATION

Category	**Average Precision**	**Average Recall**	**Average F1-score**
Technical Debt	0.72	0.81	0.76

RQ2: Are there text patterns that are an indication that technical debt exist which can be used to identify potential technical debt using NLP and machine learning techniques automatically?

The experimental results demonstrate that text patterns indicating technical debt indeed exist and can be used to identify technical debt. In general, technical debt issues are characterized by two aspects of properties including rework whether it is code refactoring or feature redesign and accumulation which is implied by some words indicating time such as previously, at present, and in the future. 20 most informative features that are strongly correlated to technical debt are shown in Table III. Each of these features may contribute differently to the identification of different types of technical debt. Intuitively, the presence of "style" and "experience" may indicate UI debt while "simplify" and "efficiency" are more likely to be indicators of design debt.

To evaluate the performance of our classifier, the average precision and recall were calculated for 10 repeated random sub-sampling validations. Precision measures the fraction of technical debt instances identified by our classifier that were proved to be correct classification. Recall measures the fraction of correctly classified technical debt items out of the total number of technical debt issues. In our experiments, the average precision and recall were 72% and 81% respectively for 10 repeated random sub-sampling validations shown in Table IV.

V. THREATS TO VALIDITY

There are two main threats to the validity of our study: threats to internal validity and threats to external validity. Threats to internal validity can be caused by the level of subjectivity in manual analysis and classification of issues as we definitely have personal bias in the understanding of issue description. To counter the threats, we had an expert external to our research team classify random samplings of the issues and solved our discrepancies by discussion. We also had discussions with the developers of the product to gain insight into the issues that we were not sure we classified correctly. Threats to external validity concern the generalization of our findings. We performed a case study on an issue data set from a commercial software project. The data set of issues we used may not be representative; that is to say, we cannot guarantee the same results will be obtained when our approach is applied to other commercial software projects. In particular, our approach may not be applicable to those projects for which issue trackers are not used to record issues.

VI. CONCLUSION AND FUTURE WORK

This paper presents an exploratory study of applying NLP and machine learning techniques to identify technical debt issues through issue trackers. We have demonstrated that we can automate the process of detecting technical debt issues through issue trackers and achieve an acceptable performance using NLP and machine learning techniques. We found that some common words in software engineering are directly or indirectly related to technical debt and these words can be used as features to decide whether a certain issue is technical debt or not. We believe the performance of our classifier will improve further when more sophisticated feature extraction and classification techniques are applied.

This exploratory study was based on a rather limited data set of 8,149 issues. Our approach needs to be validated with issue data sets from a wider range of software projects. Furthermore, we will improve the performance of our classifier by exploring more sophisticated feature extraction techniques such as mapping phrases with regular expressions and extracting semantically meaningful information based on the context and applying other classification techniques such as random forest, SVM, and deep learning. In addition, we will also develop a multi-classifier to identify technical debt of a specific type.

ACKNOWLEDGMENT

We would like to acknowledge the support of Jean Su, Steven Zhu, Billy Liu for this research. We also thank Robert Nord, Rob Fuller, Randy Hsu for their valuable review comments. This research was partially supported by a Canada NSERC grant, a MITACS grant and the Institute for Computing, Information and Cognitive System (ICICS) at UBC.

REFERENCES

[1] Avgeriou, P., et al. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). in Dagstuhl Reports. 2016. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[2] Cunningham, W., The WyCash portfolio management system. SIGPLAN OOPS Mess., 1992. 4(2): p. 29-30.

[3] Fowler, M. and K. Beck, Refactoring: improving the design of existing code. 1999: Addison-Wesley Professional.

[4] Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. in Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. 2004. IEEE.

[5] Munro, M.J. Product metrics for automatic identification of" bad smell" design problems in java source-code. in Software Metrics, 2005. 11th IEEE International Symposium. 2005. IEEE.

[6] Olbrich, S., et al. The evolution and impact of code smells: A case study of two open source systems. in Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement. 2009. IEEE Computer Society.

[7] Wong, S., et al. Detecting software modularity violations. in Proceedings of the 33rd International Conference on Software Engineering. 2011. ACM.

[8] de Freitas Farias, M.A., et al. A Contextualized Vocabulary Model for identifying technical debt on code comments. in Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on. 2015. IEEE.

[9] Maldonado, E., E. Shihab, and N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering, 2017.

[10] Brondum, J. and L. Zhu, Visualising architectural dependencies, in Proceedings of the Third International Workshop on Managing Technical Debt. 2012, IEEE Press: Zurich, Switzerland. p. 7-14.

[11] Li, Z., et al., An empirical investigation of modularity metrics for indicating architectural technical debt, in Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures. 2014, ACM: Marcq-en-Bareul, France. p. 119-128.

[12] Li, Z., P. Liang, and P. Avgeriou. Architectural technical debt identification based on architecture decisions and change scenarios. in Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on. 2015. IEEE.

[13] Bellomo, S., et al. Got technical debt? Surfacing elusive technical debt in issue trackers. in Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on. 2016. IEEE.

[14] Antoniol, G., et al. Is it a bug or an enhancement?: a text-based approach to classify change requests. in Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds. 2008. ACM.

[15] Runeson, P., M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. in Proceedings of the 29th international conference on Software Engineering. 2007. IEEE Computer Society.

[16] Wang, X., et al. An approach to detecting duplicate bug reports using natural language and execution information. in Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. 2008. IEEE.

[17] Jalbert, N. and W. Weimer. Automated duplicate detection for bug tracking systems. in Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on. 2008. IEEE.

[18] Sun, C., et al. A discriminative model approach for accurate duplicate bug report retrieval. in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. 2010. ACM.

[19] Sureka, A. and P. Jalote. Detecting duplicate bug report using character n-gram-based features. in Software Engineering Conference (APSEC), 2010 17th Asia Pacific. 2010. IEEE.

[20] Cois, C.A. and R. Kazman. Natural Language Processing to Quantify Security Effort in the Software Development Lifecycle. in SEKE. 2015.

[21] Sun, J., ‘Jieba’Chinese word segmentation tool. 2012.

[22] Bird, S. NLTK: the natural language toolkit. in Proceedings of the COLING/ACL on Interactive presentation sessions. 2006. Association for Computational Linguistics.